pksilen publishing

Clean Code Principles And Patterns

Python Edition

Petri Silén

Clean Code Principles And Patterns

Python Edition

Petri Silen

This book is for sale at
http://leanpub.com/cleancodeprinciplesandpatternspythonedition

This version was published on 2024-01-09

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

© 2024 Petri Silen

Table of Contents

1: About the Author

2: Introduction

3: Architectural Principles and Patterns
- 3.1: Software Hierarchy
- 3.2: Single Responsibility Principle
- 3.3: Uniform Naming Principle
- 3.4: Encapsulation Principle
- 3.5: Service Aggregation Principle
- 3.6: High Cohesion, Low Coupling Principle
- 3.7: Library Composition Principle
- 3.8: Avoid Duplication Principle
- 3.9: Externalized Service Configuration Principle
  - 3.9.1: Environment Variables
  - 3.9.2: Kubernetes ConfigMaps
  - 3.9.3: Kubernetes Secrets
- 3.10: Service Substitution Principle
- 3.11: Inter-Service Communication Methods
  - 3.11.1: Synchronous Communication Method
  - 3.11.2: Asynchronous Communication Method
  - 3.11.3: Shared Data Communication Method
- 3.12: Strategic Domain-Driven Design Principle
  - 3.12.1: Design Example 1: Mobile Telecom Network Analytics Software System
  - 3.12.2: Design Example 2: Banking Software System
- 3.13: Autopilot Microservices Principle
  - 3.13.1: Stateless Microservices Principle
  - 3.13.2: Resilient Microservices Principle
  - 3.13.3: Horizontally Autoscaling Microservices Principle
  - 3.13.4: Highly-Available Microservices Principle
  - 3.13.5: Observable Microservices Principle
- 3.14: Software Versioning Principles
  - 3.14.1: Use Semantic Versioning Principle
  - 3.14.2: Avoid Using 0.x Versions Principle
  - 3.14.3: Don’t Increase Major Version Principle
  - 3.14.4: Implement Security Patches and Bug Corrections to All Major Versions Principle
  - 3.14.5: Avoid Using Non-LTS Versions in Production Principle
- 3.15: Git Version Control Principle
  - 3.15.1: Feature Branch

3.15.2: Feature Toggle
3.16: Architectural Patterns
3.16.1: Event Sourcing Pattern
3.16.2: Command Query Responsibility Segregation (CQRS) Pattern
3.16.3: Distributed Transaction Patterns
3.16.3.1: Saga Orchestration Pattern
3.16.3.2: Saga Choreography Pattern
3.17: Preferred Technology Stacks Principle
4: Object-Oriented Design Principles
4.1: Object-Oriented Programming Concepts
4.1.1: Classes/Objects
4.1.2: Encapsulation
4.1.3: Abstraction
4.1.4: Inheritance
4.1.5: Interface
4.1.5.1: Interface evolution
4.1.6: Polymorphism
4.2: Programming Paradigms
4.2.1: Imperative Programming
4.2.2: Functional Programming
4.3: Multi-Paradigm Programming Principle
4.4: Why is Object-Oriented Programming Hard?
4.5: SOLID Principles
4.5.1: Single Responsibility Principle
4.5.2: Open-Closed Principle
4.5.3: Liskov's Substitution Principle
4.5.4: Interface Segregation and Multiple Inheritance Principle
4.5.5: Program Against Interfaces Principle (Generalized Dependency Inversion Principle)
4.6: Clean Microservice Design Principle
4.6.1: Real-Life Example
4.7: Vertical Slice Design Principle
4.8: Class Organization Principle
4.9: Uniform Naming Principle
4.9.1: Naming Interfaces and Classes
4.9.2: Naming Functions
4.9.2.1: Preposition in Function Name
4.9.2.2: Naming Method Pairs
4.9.2.3: Naming Boolean Functions (Predicates)
4.9.2.4: Naming Builder Methods
4.9.2.5: Naming Methods with Implicit Verbs
4.9.2.6: Naming Lifecycle Methods
4.9.2.7: Naming Function Parameters
4.10: Encapsulation Principle
4.10.1: Immutable Objects
4.10.2: Don't Leak Modifiable Internal State Outside an Object Principle

4.10.3: Don’t Assign From a Method Parameter to a Modifiable Attribute
4.11: Prefer Composition Over Inheritance Principle
4.12: Tactical Domain-Driven Design Principle
4.12.1: Tactical DDD Concepts
4.12.1.1: Entities
4.12.1.2: Value Objects
4.12.1.3: Aggregates
4.12.1.4: Aggregate Roots
4.12.2: Actors
4.12.2.1: Factories
4.12.2.2: Repositories
4.12.2.3: Services
4.12.2.4: Events
4.12.2.5: Event Storming
4.12.3: Domain-Driven Design Example: Data Exporter Microservice
4.12.4: Domain-Driven Design Example 2: Anomaly Detection Microservice
4.13: Design Patterns
4.13.1: Design Patterns for Creating Objects
4.13.1.1: Factory Pattern
4.13.1.2: Abstract Factory Pattern
4.13.1.3: Static Factory Method Pattern
4.13.1.4: Builder Pattern
4.13.1.5: Singleton Pattern
4.13.1.6: Prototype Pattern
4.13.1.7: Object Pool Pattern
4.13.2: Structural Design Patterns
4.13.2.1: Composite Pattern
4.13.2.2: Facade Pattern
4.13.2.3: Bridge Pattern
4.13.2.4: Strategy Pattern
4.13.2.5: Adapter Pattern
4.13.2.6: Proxy Pattern
4.13.2.7: Decorator Pattern
4.13.2.8: Flyweight Pattern
4.13.3: Behavioral Design Patterns
4.13.3.1: Chain of Responsibility Pattern
4.13.3.2: Observer Pattern
4.13.3.3: Command/Action Pattern
4.13.3.4: Iterator Pattern
4.13.3.5: State Pattern
4.13.3.6: Mediator Pattern
4.13.3.7: Template Method Pattern
4.13.3.8: Memento Pattern
4.13.3.9: Visitor Pattern
4.13.3.10: Null Object Pattern
4.14: Don’t Ask, Tell Principle
4.15: Law of Demeter
4.16: Avoid Primitive Obsession Principle

4.17: Dependency Injection (DI) Principle
4.18: Avoid Code Duplication Principle
4.19: Inheritance in Cascading Style Sheets (CSS)

5: Coding Principles
5.1: Uniform Variable Naming Principle
5.1.1: Naming Integer Variables
5.1.2: Naming Floating-Point Number Variables
5.1.3: Naming Boolean Variables
5.1.4: Naming String Variables
5.1.5: Naming Enum Variables
5.1.6: Naming Collection (List and Set) Variables
5.1.7: Naming Dictionary Variables
5.1.8: Naming Pair and Tuple Variables
5.1.9: Naming Object Variables
5.1.10: Naming Optional Variables
5.1.11: Naming Function Variables (Callbacks)
5.1.12: Naming Class Attributes
5.1.13: General Naming Rules
5.1.13.1: Use Short, Common Names
5.1.13.2: Pick One Term And Use It Consistently
5.1.13.3: Avoid Obscure Abbreviations
5.1.13.4: Avoid Too Short Or Meaningless Names
5.2: Uniform Source Code Repository Structure Principle
5.3: Domain-Based Source Code Structure Principle
5.4: Avoid Comments Principle
5.4.1: Name Things Properly
5.4.2: Single Return Of Named Value At The End Of Function
5.4.3: Return Type Aliasing
5.4.4: Extract Constant for Boolean Expression
5.4.5: Extract Named Constant or Enumerated Type
5.4.6: Extract Function
5.4.7: Avoid Comments for Regular Expression
5.4.8: Avoiding Comments in Bash Shell Scripts
5.5: Function Single Return Principle
5.6: Use Type Annotations for Production Code Principle
5.6.1: Function Arguments Might Be Given in Wrong Order
5.6.2: Function Argument Might Be Given with Wrong Type
5.6.3: Function Return Value Type Might Be Misunderstood
5.6.4: Refactoring Code Is More Difficult
5.6.5: Forced to Write Public API Comments
5.6.6: Type Errors Are Not Found in Testing
5.7: Refactoring Principle
5.7.1: Rename
5.7.2: Extract Method
5.7.3: Extract Class
5.7.4: Extract Constant
5.7.5: Replace Conditionals with Polymorphism

5.7.6: Introduce Parameter Object
5.7.7: Invert If-Statement
5.7.8: Creating Rich Object
5.8: Static Code Analysis Principle
5.8.1: Common Static Code Analysis Issues
5.9: Error/Exception Handling Principle
5.9.1: Returning Errors
5.9.1.1: Returning Failure Indicator
5.9.1.2: Returning an Optional Value
5.9.1.3: Returning an Error Object
5.9.1.4: Adapt to Wanted Error Handling Mechanism
5.9.1.5: Functional Exception Handling
5.10: Avoid Off-By-One Errors Principle
5.11: Be Critical When Googling or Using Generative AI Principle
5.12: Make One Change At A Time Principle
5.13: Use Appropriate Data Structure Principle
5.13.1: List
5.13.2: Dictionary
5.13.3: Tuple
5.13.4: Set
5.13.5: String
5.13.6: Bytearray
5.13.7: Counter
5.13.8: OrderedDict
5.13.9: Deque (Double Ended Queue)
5.13.10: Customized Built-In Collections
5.13.11: Stack (LIFO Queue)
5.13.12: Queue (FIFO Queue)
5.13.13: Priority Queue
5.13.14: Synchronized Queues (LIFO/FIFO)
5.14: Optimization Principle
5.14.1: Optimization Patterns
5.14.1.1: Optimize Busy Loops Only Pattern
5.14.1.2: Remove Unnecessary Functionality Pattern
5.14.1.3: Object Pool Pattern
5.14.1.4: Use Optimal Data Structures Pattern
5.14.1.5: Algorithm Complexity Reduction Pattern
5.14.1.6: Cache Function Results Pattern
5.14.1.7: Buffer File I/O Pattern
5.14.1.8: Share Identical Objects a.k.a Flyweight Pattern
6: Testing Principles
6.1: Functional Testing Principles
6.1.1: Unit Testing Principle
6.1.1.1: Test-Driven Development (TDD)
6.1.1.2: Unit Specification-Driven Development (USDD)
6.1.1.3: Naming Conventions

6.1.1.4: Mocking
6.1.1.5: Web UI Component Unit Testing
6.1.2: Software Component Integration Testing Principle
6.1.2.1: Web UI Integration Testing
6.1.2.2: Setting Up Integration Testing Environment
6.1.3: Complete Example with BDD, ATDD, DDD, OOD and TDD
6.1.4: End-to-End (E2E) Testing Principle
6.2: Non-Functional Testing Principle
6.2.1: Performance Testing
6.2.2: Data Volume Testing
6.2.3: Stability Testing
6.2.4: Reliability Testing
6.2.5: Stress and Scalability Testing
6.2.6: Security Testing
6.2.7: Other Non-Functional Testing
6.2.7.1: Visual Testing
7: Security Principles
7.1: Shift Security to Left Principle
7.2: Have a Product Security Lead Principle
7.3: Use Threat Modelling Process Principle
7.3.1: Decompose Application
7.3.2: Determine and Rank Threats
7.3.2.1: STRIDE Threat Examples
7.3.3: Determine Countermeasures and Mitigation
7.3.4: Threat Modeling Example using STRIDE
7.3.4.1: Decompose Application
7.3.4.2: Determine and Rank Threats
7.3.5: Determine Countermeasures and Mitigation
7.3.6: Threat Modeling Example Using ASF
7.4: Security Features
7.4.1: Authentication and Authorization
7.4.1.1: OpenID Connect Authentication and Authorization in Frontend
7.4.1.2: OAuth2 Authorization in Backend
7.4.2: Password Policy
7.4.3: Cryptography
7.4.3.1: Encryption Key Lifetime and Rotation
7.4.4: Denial-of-service (DoS) Prevention
7.4.5: Database Security
7.4.6: SQL Injection Prevention
7.4.7: OS Command Injection Prevention
7.4.8: Security Configuration
7.4.9: Automatic Vulnerability Scanning
7.4.10: Integrity
7.4.11: Error Handling
7.4.12: Logging
7.4.13: Audit Logging
7.4.14: Input Validation
7.4.14.1: Validating Numbers

7.4.14.2: Validating Strings
7.4.14.3: Validating Timestamps
7.4.14.4: Validating Arrays
7.4.14.5: Validating Objects
7.4.14.6: Validating Files Uploaded to Server

8: API Design Principles
8.1: Frontend Facing API Design Principles
8.1.1: JSON-based RPC API Design Principle
8.1.2: REST API Design Principle
8.1.2.1: Creating a Resource
8.1.2.2: Reading Resources
8.1.2.3: Updating Resources
8.1.2.4: Deleting Resources
8.1.2.5: Executing Non-CRUD Actions on Resources
8.1.2.6: Resource Composition
8.1.2.7: HTTP Status Codes
8.1.2.8: HATEOAS and HAL
8.1.2.9: API Versioning
8.1.2.10: Documentation
8.1.2.11: Implementation Example
8.1.3: GraphQL API Design
8.1.4: Subscription-Based API Design
8.1.4.1: Server-Sent Events (SSE)
8.1.4.2: GraphQL Subscriptions
8.1.5: WebSocket Example
8.2: Inter-Microservice API Design Principles
8.2.1: Synchronous API Design Principle
8.2.1.1: gRPC-Based API Design Example
8.2.2: Asynchronous API Design Principle
8.2.2.1: Request-Only Asynchronous API Design
8.2.2.2: Request-Response Asynchronous API Design

9: Databases And Database Principles
9.1: Relational Databases
9.1.1: Structure of Relational Database
9.1.2: Use Object Relational Mapper (ORM) Principle
9.1.3: Entity/Table Relationships
9.1.3.1: One-To-One/Many Relationships
9.1.3.2: Many-To-Many Relationships
9.1.3.3: Sales Item Repository Example
9.1.4: Use Parameterized SQL Statements Principle
9.1.5: Normalization Rules
9.1.5.1: First Normal Form (1NF)
9.1.5.2: Second Normal Form (2NF)
9.1.5.3: Third Normal Form (3NF)
9.2: Document Databases
9.3: Key-Value Database Principle
9.4: Wide-Column Databases

9.5: Search Engines

10: Concurrent Programming Principles
10.1: Threading Principle
10.1.1: Parallel Executors
10.2: Thread Safety Principle
10.2.1: Use Locking for Mutual Exclusion (Mutex)
10.2.2: Atomic Variables
10.2.3: Concurrent Collections
10.3: Publish/Subscribe Shared State Change Principle

11: Teamwork Principles
11.1: Use Agile Framework Principle
11.2: Define the Done Principle
11.3: You Write Code for Other People Principle
11.4: Avoid Technical Debt Principle
11.5: Software Component Documentation Principle
11.6: Code Review Principle
11.6.1: Focus on Object-Oriented Design
11.6.2: Focus on Function Specification by Unit Tests
11.6.3: Focus on Proper and Uniform Naming
11.6.4: Don’t Focus on Premature Optimization
11.6.5: Detect Possible Malicious Code
11.7: Uniform Code Formatting Principle
11.8: Highly Concurrent Development Principle
11.8.1: Dedicated Microservices and Microlibraries
11.8.2: Dedicated Domains
11.8.3: Follow Open-Closed Principle
11.9: Pair Programming Principle
11.10: Well-Defined Development Team Roles Principle
11.10.1: Product Owner
11.10.2: Scrum Master
11.10.3: Software Developer
11.10.4: Test Automation Developer
11.10.5: DevOps Engineer
11.10.6: UI Designer
11.11: Competence Transfer Principle

12: DevSecOps
12.1: SecOps Lifecycle
12.2: DevOps Lifecycle
12.2.1: Plan
12.2.2: Code
12.2.3: Build and Test
12.2.4: Release

12.2.4.1: Example Dockerfile
12.2.4.2: Example Kubernetes Deployment
12.2.4.3: Example CI/CD Pipeline
12.2.5: Deploy
12.2.6: Operate
12.2.7: Monitor
12.2.7.1: Logging to Standard Input
12.2.7.2: Distributed Tracing
12.2.7.3: Metrics Collection
12.2.7.4: Metrics Visualization
12.2.7.5: Alerting
12.2.8: Software System Alerts Dashboard Example
12.2.9: Microservice Grafana Dashboard Example
12.2.9.1: Logging
12.2.9.2: OpenTelemetry Log Data Model
12.2.9.3: PrometheusRule Example
13: Conclusion
14: Appendix A
15: Appendix B

1: About the Author

Petri Silén is a seasoned software developer working at Nokia Networks in Finland with industry experience of almost 30 years. He has done both frontend and backend development with a solid competence in multiple programming languages, including C++, Java, Python and JavaScript/TypeScript. He started his career at Nokia Telecommunications in 1995. During his first years, he developed a real-time mobile networks analytics product called “Traffica” in C++ for major telecom customers worldwide, including companies like T-Mobile, Orange, Vodafone, and Claro. The initial product was for monitoring a 2G circuit-switched core network and GPRS packet-switched core network. Later, functionality to Traffica was added to cover new network technologies, like 3G circuit-switched and packet core networks, 3G radio networks, and 4G/LTE. He later developed new functionality for Traffica using Java and web technologies, including jQuery and React. During the last few years, he has developed cloud-native containerized microservices with Java and C++ for the next-generation Customer and Networks Insights (CNI) product used by major communications service providers like Verizon, AT&T, USCC, and KDDI. The main application areas he has contributed during the last years include KPI-based real-time alerting, anomaly detection for KPIs, and configurable real-time data exporting.

During his free time, he has developed a data visualization application using React, Redux, TypeScript, and Jakarta EE. He has also developed a security-first cloud-native microservice framework for Node.js in TypeScript. He likes to take care of his Kaapo cat, take walks, play tennis and badminton, ski in the winter, and watch soccer and ice hockey on TV.

2: Introduction

This book teaches you how to write clean code. It presents software design and development principles and patterns in a very practical manner. This book is suitable for both junior and senior developers. Some basic knowledge of programming in Python is required. All examples in this book are presented in Python, except some examples related to frontend code, which are in JavaScript/TypeScript. The content of this book is divided into ten chapters. All Python examples in this book require Python 3.11 or later. This is a book for primarily software developers. For that reason, some topics are not covered exhaustively. This includes topics related to architecture, DevSecOps, E2E, and non-functional testing. Those are topics most relevant to software architects, DevOps specialists and test/QA engineers. I want to cover them in this book because it is always good to have a basic understanding of topics closely related to software development.

This book presents a lot of principles, best practices, and patterns. It can be difficult to grasp them all on a single read, which is not the purpose. You should pick *the most relevant topics for yourself*, embrace them, and try to put them into use in everyday coding. You can always return to the book to learn additional topics. Some principles, patterns, and practices can be subjective and open for debate, but I have only put into this book such principles, patterns, and practices that I have used or would use myself. In the last *Conclusion* chapter, I will list the topics that I consider the most important based on my experience in real-life projects.

The *second chapter* is about *architectural design principles* that enable the development of true cloud-native microservices. The first architectural design principle described is the single responsibility principle, which defines that a piece of software should be responsible for one thing at its abstraction level. Then, a uniform naming principle for microservices, clients, APIs, and libraries is presented. The encapsulation principle defines how each software component

should hide its internal state behind a public API. The service aggregation principle is introduced with a detailed explanation of how a higher-level microservice can aggregate lower-level microservices. Architectural patterns like event sourcing, command query responsibility segregation (CQRS), and distributed transactions are discussed. Distributed transactions are covered with examples using the saga orchestration pattern and the saga choreography pattern. You get answers on how to avoid code duplication at the architectural level. The externalized configuration principle describes how service configuration should be handled in modern environments. We discuss the service substitution principle, which states that dependent services a microservice uses should be easily substitutable. The importance of autopilot microservices is discussed from the point of view of statelessness, resiliency, high availability, observability, and automatic scaling. Towards the end of the chapter, there is a discussion about different ways microservices can communicate. Several rules are presented on how to version software components. The chapter ends with discussing why limiting the number of technologies used in a software system is helpful.

The *third chapter* presents *object-oriented design principles*. We start the chapter with object-oriented programming concepts and programming paradigms followed by the SOLID principles: Single responsibility principle, open-closed principle, Liskov's substitution principle, interface segregation principle, and dependency inversion principle. Each SOLID principle is presented with realistic but simple examples. The uniform naming principle defines a uniform way to name interfaces, classes, functions, function pairs, boolean functions (predicates), builder, factory, conversion, and lifecycle methods. The encapsulation principle describes that a class should encapsulate its internal state and how immutability helps ensure state encapsulation. The encapsulation principle also discusses the importance of not leaking an object's internal state out. The object composition principle defines that composition should be preferred over inheritance. Tactical Domain-driven design (DDD) is presented with two real-world examples. All the design patterns from the *GoF's Design Patterns* book are presented with realistic yet straightforward

examples. The don’t ask, tell principle is presented as a way to avoid the feature envy design smell. The chapter also discusses avoiding primitive obsession and the benefits of using semantically validated function arguments. The chapter ends by presenting the dependency injection principle and avoiding code duplication principle.

The *fourth chapter* is about *coding principles*. The chapter starts with a principle for uniformly naming variables in code. A uniform naming convention is presented for integer, floating-point, boolean, string, enum, and collection variables. Also, a naming convention is defined for maps, pairs, tuples, objects, and callback functions. The uniform source code repository structure principle is presented with examples. Next, the avoid comments principle lists reasons why most comments are unnecessary and defines concrete ways to remove unnecessary comments from the code. The following concrete actions are presented: naming things correctly, returning a named value, return-type aliasing, extracting a constant for a boolean expression, extracting a constant for a complex expression, extracting enumerated values, and extracting a function. The chapter discusses the benefits of using type hints. We discuss the most common refactoring techniques: renaming, extracting a method, extracting a variable, replacing conditionals with polymorphism, introducing a parameter object, and making anemic objects rich objects. The importance of static code analysis is described, and the most popular static code analysis tools are listed. The most common static code analysis issues and the preferred way to correct them are listed. Handling errors and exceptions correctly in code is fundamental and can be easily forgotten or done wrong. This chapter instructs how to handle errors and exceptions and return errors by returning a boolean failure indicator, an optional value, or an error object. The chapter instructs how to adapt code to a wanted error-handling mechanism and handle errors functionally. Ways to avoid off-by-one errors are presented. Readers are instructed on handling situations where some code is copied from a web page found by googling or generated by AI. Advice is given on what data structure is the most appropriate for a given use

case. The chapter ends with a discussion about code optimization: when and how to optimize.

The *fifth chapter* is dedicated to *testing principles*. The chapter starts with the introduction of the functional testing pyramid. Then, we present unit testing and instruct how to use test-driven development (TDD) and behavior-driven development for individual functions. We give unit test examples with mocking. When introducing software component integration testing, we discuss behavior-driven development (BDD), acceptance test-driven development (ATDD), and the Gherkin language to specify features formally. Integration test examples are given using Behave and the Postman API development platform. The chapter also discusses the integration testing of UI software components. We end the integration testing section with an example of setting up an integration testing environment using Docker Compose. We give a complete example of applying multiple design approaches (BDD, ATDD, DDD, OOD, and TDD) in a small project. Lastly, the purpose of end-to-end (E2E) testing is discussed with some examples. The chapter ends with a discussion about non-functional testing. The following categories of non-functional testing are covered in more detail: performance testing, stability testing, reliability testing, security testing, stress, and scalability testing.

The *sixth chapter* handles *security principles*. The threat modeling process is introduced, and there is an example of how to conduct threat modeling for a simple API microservice. A full-blown frontend OpenID Connect/OAuth 2.0 authentication and authorization example for a SPA (single-page application) with TypeScript, Vue.js, and Keycloak is implemented. Then, we discuss how authorization by validating a JWT should be handled in the backend. The chapter ends with a discussion of the most important security features: password policy, cryptography, denial-of-service prevention, SQL injection prevention, security configuration, automatic vulnerability scanning, integrity, error handling, audit logging, and input validation.

The *seventh chapter* is about *API design principles*. First, we tackle design principles for frontend-facing APIs. We discuss

how to design JSON-based RPC, REST, and GraphQL APIs. Also, subscription-based and real-time APIs are presented with realistic examples using Server-Sent Events (SSE) and the WebSocket protocol. The last part of the chapter discusses inter-microservice API design and event-driven architecture. gRPC is introduced as a synchronous inter-microservice communication method, and examples of request-only and request-response asynchronous APIs are presented.

The *8th chapter* discusses *databases and related principles*. We cover the following types of databases: relational databases, document databases (MongoDB), key-value databases (Redis), wide-column databases (Cassandra), and search engines. For relational databases, we present how to use object-relational mapping (ORM), define one-to-one, one-to-many and many-to-many relationships, and parameterized SQL queries. Finally, we present three normalization rules for relational databases.

The *9th chapter* presents *concurrent programming principles* regarding threading and thread safety. For thread safety, we present several ways to achieve thread synchronization: locks, atomic variables, and thread-safe collections. We also discuss publishing changes to a shared state and subscribing to them from two different threads.

The *10th chapter* discusses *teamwork principles*. We explain the importance of using an agile framework and discuss the fact that a developer usually never works alone and what that entails. We discuss how to document a software component so that onboarding new developers is quick and easy. Technical debt in software is something that each team should avoid. Some concrete actions to prevent technical debt are presented. Code reviews are something teams should do, and this chapter gives guidance on what to focus on in code reviews. The chapter ends with a discussion of developer roles each team should have and provides hints on enabling a team to develop software as concurrently as possible.

The *11th chapter* is dedicated to *DevSecOps*. DevOps describes practices that integrate software development (Dev) and software operations (Ops). It aims to shorten the software

development life cycle through parallelization and automation and provides continuous delivery with high software quality. DevSecOps is a DevOps augmentation where security practices are integrated into the DevOps practices. This chapter presents the phases of the DevOps lifecycle: plan, code, build and test, release, deploy, operate, and monitor. The chapter gives an example of creating a microservice container image and how to specify the deployment of a microservice to a Kubernetes cluster. Also, a complete example of a CI/CD pipeline using GitHub Actions is provided. Guidance on how to implement microservice observability (logging, metrics, alerts) is given with some examples.

3: Architectural Principles and Patterns

This chapter describes architectural principles and patterns for designing clean, modern, cloud-native software systems and applications. Architectural design means designing a software system consisting of multiple software components. This chapter focuses on modern cloud-native microservices, but some principles can be used with a monolithic software architecture. This book does not cover monolithic software architecture design. Still, if you design a monolithic software system, you should consider implementing a so-called modular monolith, which is a monolith with modularity: Different functionalities are separated inside the monolith. This modular architecture makes it easier to dismantle the monolith to microservices or extract part(s) of the monolith into microservice(s) if needed later.

Cloud-native software is built of loosely coupled scalable, resilient, and observable services that can run in public, private, or hybrid clouds. Cloud-native software utilizes technologies like containers (e.g., Docker), microservices, serverless functions, and container orchestration (e.g., Kubernetes), and it can be automatically deployed using declarative code. Examples in this chapter assume microservices deployed in a Kubernetes environment. Kubernetes is a cloud provider agnostic way of running containerized microservices and has gained massive popularity in recent years. If you are new to Kubernetes, you can familiarize yourself with the most relevant Kubernetes concepts.

This chapter discusses the following architectural principles and patterns:

- Single responsibility principle
- Uniform naming principle

- Encapsulation principle
- Service aggregation principle
- High cohesion, low coupling principle
- Library composition principle
- Avoid duplication principle
- Externalized service configuration principle
- Service substitution principle
- Autopilot microservices principle
    - Stateless microservices principle
    - Resilient microservices principle
    - Horizontally autoscaling microservices principle
    - Highly-available microservices principle
    - Observable services principle
- Inter-service communication patterns
- Strategic domain-driven design principle
- Software versioning principles
- Git version control principle
- Architectural patterns
- Preferred technology stacks principle

3.1: Software Hierarchy

A *software system* consists of multiple computer programs and anything related to those programs to make them operable, including but not limited to configuration, deployment code, and documentation. A software system is divided into two parts: *the backend* and *the frontend*. Backend software runs on servers, and frontend software runs on client devices like PCs, tablets, and phones. Backend software consists of *services*. Frontend software consists of *clients* that use backend services and *standalone applications* that do not use any backend services. An example of a standalone application is a

calculator or a simple text editor. A service is something that provides a specific functionality and is running continuously.

There are a couple of other types of programs in the backend. Programs that run once or on-demand are called Jobs, and programs that run on a schedule are called CronJobs. Jobs and CronJobs are typically related to services and perform administrative tasks related to a specific service. According to the *single responsibility principle*, the service itself should not perform administrative tasks but should be dedicated to providing a particular service only, like *order-service* provides operations on orders. For example, after the *order-service* installation, an *order-db-init-job* could be triggered to initialize the *order-service*'s database. An *order-db-cleanup-cronjob* could be triggered on a schedule to perform cleanup actions on the *order-service*'s database.

A separate service can be created to control a service's deployment and lifecycle. In a Kubernetes environment, that service is called an operator.

A command line interface (CLI) program is an additional program type. CLI programs are typically used for administrative tasks. For example, an *admin-cli* could be created to install and upgrade the software system.

The term *application* is often used to describe a single program designated for a specific purpose. In general, a software application is some software applied to solve a specific problem. From an end user's point of view, all clients are applications. But from a developer's point of view, an application needs both a client and backend service(s) to be functional unless the application is a *standalone application*. In this book, I will use the term application to designate a logical grouping of program(s) and related artifacts, like configuration, to form a functional piece of the software system dedicated to a specific purpose. In my definition, a non-standalone application consists of one or more services and possibly a client or clients to fulfill an end user's need.

Let's say we have a software system for mobile telecom network analytics. That system provides data visualization functionality. We can call the data visualization part of the

software system a data visualization application. That application consists of, for example, a web client and two services, one for fetching data and one for configuration. Suppose we also have a generic data ingester microservice in the same software system. That generic data ingester is not an application without some configuration that makes it a specific service that we can call an application. For example, the generic data ingester can have a configuration to ingest raw data from the radio network part of the mobile network. The generic data ingester and the configuration together form an application: a radio network data ingester. Then, there could be another configuration for ingesting raw data from the core network part of the mobile network. That configuration and the generic data ingester make another application: a core network data ingester.

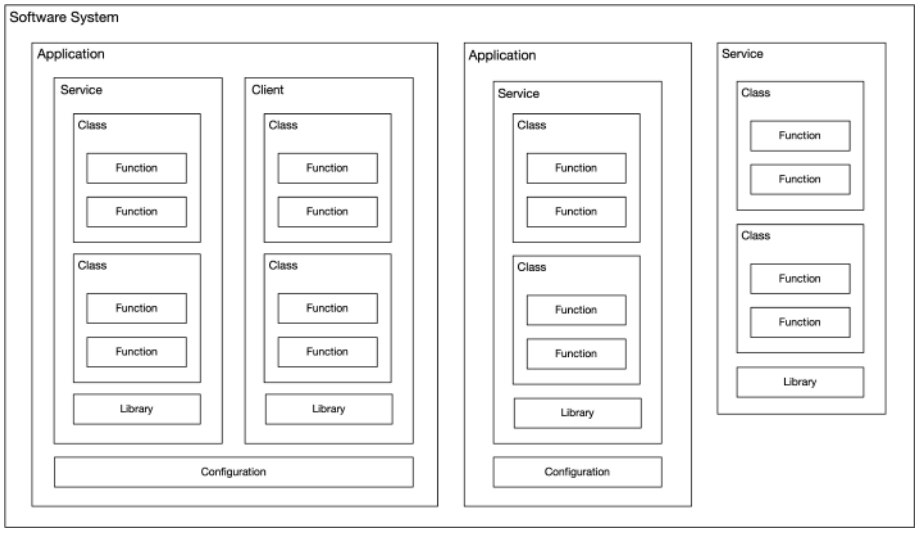

Figure 3.1. Software Hierarchy

Computer programs and *libraries* are *software components*. A *software component* is something that can be individually packaged, tested, and delivered. It consists of one or more classes, and a class consists of one or more functions (class methods). (There are no traditional classes in purely functional languages, but software components consist only of functions.) A computer program can also be composed of one or more libraries, and a library can be composed of other libraries.

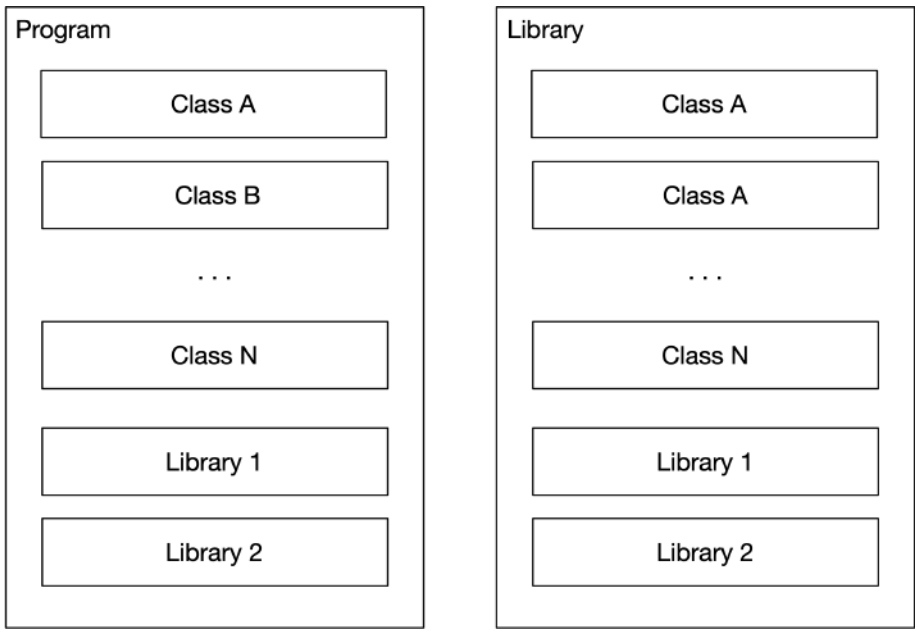

Figure 3.2. Software Components

3.2: Single Responsibility Principle

> ***A software entity should have only a single responsibility at its abstraction level.***

A software system at the highest level in the hierarchy should have a single dedicated purpose. For example, there can be an e-commerce or payroll software system. However, there should not be a software system that handles both e-commerce and payroll-related activities. If you were a software vendor and had made an e-commerce software system, selling that to clients wanting an e-commerce solution would be easy. But if you had made a software system encompassing both e-commerce and payroll functionality, it would be hard to sell that to customers wanting only an e-commerce solution because they might already have a payroll software system and, of course, don't want another one.

Let's consider the application level in the software hierarchy. Suppose we have designed a software system for telecom network analytics. This software system is divided into four different applications: Radio network data ingestion, core

network data ingestion, data aggregation, and data visualization. Each of these applications has a single dedicated purpose. Suppose we had coupled the data aggregation and visualization applications into a single application. In that case, replacing the data visualization part with a 3rd party application could be difficult. But when they are separate applications with a well-defined interface, replacing the data visualization application with a 3rd party application would be much easier, if needed.

A software component should also have a single dedicated purpose. A service type of software component with a single responsibility is called a *microservice*. For example, in an e-commerce software system, one microservice could be responsible for handling orders and another for handling sales items. Both of those microservices are responsible for one thing only. By default, we should not have a microservice responsible for orders and sales items. That would be against the single responsibility principle because order and sales item handling are different functionalities at the same level of abstraction. Combining two or more functionalities into a single microservice sometimes makes sense. The reason could be that the functionalities firmly belong together, and putting functionalities in a single microservice would diminish the drawbacks of microservices, like needing to use distributed transactions. Thus, the size of a microservice can vary and depends on the abstraction level of the microservice. Some microservices can be small, and others can be larger if they are at a higher level of abstraction. A microservice is always smaller than a monolith and larger than a single function. The higher the level of abstraction the microservice is, the fewer microservice benefits you get. Depending on the software system size and its design, the number of microservices in it can vary from a handful to tens or even hundreds of microservices.

Let’s have an example of an e-commerce software system that consists of the following functionality:

- sales items
- shopping cart

- orders

Let’s design how to split the above-described functionality into microservices. When deciding which functionality is put in the same microservice, we should ensure that the requirement of a single responsibility is met and that high functional and non-functional cohesion is achieved. High functional cohesion means that two functionalities depend on each other and tend to change together. Examples of low functional cohesion would be email sending and shopping cart functionality. Those two functionalities don’t depend on each other, and they don’t change together. Thus, we should implement email sending and shopping cart functionalities as separate microservices. Non-functional cohesion is related to all non-functional aspects like architecture, technology stack, deployment, scalability, resiliency, availability, observability, etc. We discuss cohesion and coupling more in a later section of this chapter.

We should not put all the e-commerce software system functionality in a single microservice because there is not high non-functional cohesion between sales items-related functionality and the other functionality. The functionality related to sales items should be put into its own microservice that can scale separately because the sales item microservice receives much more traffic than the shopping cart and order services. Also, we should be able to choose appropriate database technology for the sales item microservice. The database engine should be optimized for a high number of reads and a low number of writes. Later, we might realize that the pictures of the sales items should not be stored in the same database as other sales item-related information. We could then introduce a new microservice dedicated to storing/retrieving images of sales items.

Instead of implementing shopping cart and order-related functionality as two separate microservices, we could implement them as a single microservice. This is because shopping cart and order functionalities have high functional cohesion. For example, whenever a new order is placed, the items from the shopping cart should be read and removed.

Also, the non-functional cohesion is high. Both services can use the same technology stack and scale together. We eliminate distributed transactions by putting the two functionalities in a single microservice and can use standard database transactions. That simplifies the codebase and testing of the microservice. We should not name the microservice as *shopping-cart-and-order-service*, because that name does not denote a single responsibility. We should name the microservice using a term on a higher level of abstraction. For example, we could name it *purchase-service* because the microservice is responsible for functionality related to a customer purchasing item(s) from the e-commerce store. In the future, if we notice that the requirement of high functional and non-functional cohesion is no longer met, it is possible to split the *purchase-service* into two separate microservices: *shopping-cart-service* and *order-service*. When you first implement the *purchase-service*, you should put the code related to different subdomains in separate domain-specific source code directories: *shoppingcart* and *order*. It will be easier later to extract those two functionalities into separate microservices.

The initial division of a software system into microservices should not be engraved in stone. You can make changes to that in the future if seen as appropriate. You might realize that a particular microservice should be divided into two separate microservices due to different scaling needs, for example. Or you might realize that it is better to couple two or more microservices into a single microservice to avoid complex distributed transactions, for instance.

There are many advantages to microservices:

- Improved productivity
    - You can choose the best-suited programming language and technology stack
    - Microservices are easy to develop in parallel because there will be fewer merge conflicts
    - Developing a monolith can result in more frequent merge conflicts

- Improved resiliency and fault isolation
    - A fault in a single microservice does not bring other microservices down
    - A bug in a monolith can bring the whole monolith down
- Better scalability
    - Stateless microservices can automatically scale horizontally
    - Horizontal scaling of a monolith is complicated or impossible
- Better data security and compliance
    - Each microservice encapsulates its data, which can be accessed via a public API only
- Faster and easier upgrades
    - Upgrading only the changed microservice(s) is enough. There is no need to update the whole monolith every time
- Faster release cycle
    - Build the changed microservice only. There is no need to build the whole monolith when something changes
- Fewer dependencies
    - Lower probability for dependency conflicts
- Enables “open-closed architecture”, meaning architecture that is more open for extension and more closed for modification
    - New functionality not related to any existing microservice can be put into a new microservice instead of modifying the current codebase.

The main drawback of microservices is the complexity that a distributed architecture brings. Implementing transactions between microservices requires implementing distributed transactions, which are more complex than standard database

transactions. Distributed transactions require more code and testing. You can avoid distributed transactions by placing closely related services in a single microservice whenever possible. Operating and monitoring a microservice-based software system is complicated. Also, testing a distributed system is more challenging than testing a monolith. Development teams should focus on these "problematic" areas by hiring DevOps and test automation specialists.

A library-type software component should also have a single responsibility. Like calling single-responsibility services microservices, we can call a single-responsibility library a *microlibrary*. For example, there could be a library for handling YAML-format content and another for handling XML-format content. We shouldn't try to bundle the handling of both formats into a single library. If we did and needed only the YAML-related functionality, we would also always get the XML-related functionality. Our code would always ship with the XML-related code, even if never used. This can introduce unnecessary code bloat. We would also have to take any security patch for the library into use, even if the patch was only for the XML-related functionality we don't use.

3.3: Uniform Naming Principle

> ***Use a specific postfix to name different types of software components.***

When developing software, you should establish a naming convention for different kinds of software components: Microservices, clients, jobs, operators, command line interfaces (CLIs), and libraries. Next, I present my suggested way of naming different software components.

The preferred naming convention for microservices is *<service's purpose>-service* or *<service's purpose>-svc*. For example: *data-aggregation-service* or *email-sending-svc*. Use the microservice name systematically in different places. For example, use it as the Kubernetes Deployment name and the source code repository name (or directory name in case of a

monorepo). It is enough to name your microservices with the *service* postfix instead of a *microservice* postfix because each service should be a microservice by default. So, there would not be any real benefit in naming microservices with the *microservice* postfix. That would make the microservice name longer without any added value.

If you want to be more specific in naming microservices, you can name API microservices with an *api* postfix instead of the more generic *service* postfix, for example, *sales-item-api*. In this book, I am not using the *api* postfix but always use the *service* postfix only.

The preferred naming convention for clients is *<client's purpose>-<client type>-client*, *<client's purpose>-<ui type>-ui* or *<client's purpose>-<app type>-app*. For example: *data-visualization-web-ui*, *data-visualization-mobile-client*, *data-visualization-android-app* or *data-visualization-ios-app*. In this book, I mostly use the *client* postfix because it is the most generic term.

The preferred naming convention for jobs is *<job's purpose>-job*. For example, a job that initializes the database for orders could be named *order-db-init-job*.

The preferred naming convention for cron jobs is *<cron job's purpose>-cronjob*. For example, a cron job that performs order database backup regularly could be named *order-db-backup-cronjob*.

The preferred naming convention for operators is *<operated service>-operator*. For example, an operator for *order-service* could be named *order-service-operator*.

The preferred naming convention for a CLI is *<CLI's purpose>-cli*. For example, a CLI that is used to administer the software system could be named *admin-cli*.

The preferred naming convention for libraries is either *<library's purpose>-lib* or *<library's purpose>-library*. For example: *common-utils-lib* or *common-ui-components-library*.

When using these naming conventions, a clear distinction between a microservice, client, (cron) job, operator, CLI, and

library-type software component can be made only by looking at the name. Also, it is easy to recognize if a source code repository contains a microservice, client, (cron) job, operator, CLI, or library.

3.4: Encapsulation Principle

> ***Microservice must encapsulate its internal state behind a public API. Anything behind the public API is considered private to the microservice and cannot be accessed directly by other microservices.***

Microservices should define a public API that other microservices use for interfacing. Anything behind the public API is private and inaccessible from other microservices.

While microservices should be made stateless (the *stateless services principle* is discussed later in this chapter), a stateless microservice needs a place to store its state outside the microservice. Typically, the state is stored in a database or a cache. The database is the microservice's internal dependency and should be made private to the microservice, meaning that no other microservice can directly access the database. Access to the database happens indirectly using the microservice's public API.

It is discouraged to allow multiple microservices to share a single database because then there is no control over how each microservice will use the database and what requirements each microservice has for the database. (The architecture where multiple services use a shared database is usually called *service-based architecture* and is different from a microservice architecture.)

It can be possible to share a *physical* database with several microservices if each uses its own *logical* database. This requires that a specific database user is created for each microservice. Each database user can access only one logical database dedicated to a particular microservice. In this way, no microservice can directly access another microservice's data. However, this approach can still pose some problems because

the dimensioning requirements of all microservices for the shared physical database must be considered. Also, the deployment responsibility of the shared database must be decided. The shared database could be deployed as a platform or common service as part of the platform or common services deployment, for example.

If a database is shared between microservices, it is called *service-based architecture*, not microservice architecture per se. A service-based architecture's benefit is avoiding complex distributed transactions that the actual microservice architecture would entail. When having a service-based architecture, distributed transactions can be replaced with database transactions. The main problem with this architectural style is that each service is no longer necessarily single-responsibility, which can sometimes be an acceptable trade-off. (e.g. *shopping-cart-service* and *order-service* with a shared database, and creating an order with *order-service* will also read and empty the shopping cart in a single database transaction. Now, the *order-service* is no longer a single-responsibility service because it is doing some work that should be in the *shopping-cart service* in the case of a microservice architecture.

3.5: Service Aggregation Principle

> ***Service on a higher level of abstraction aggregates services on a lower level of abstraction.***

Service aggregation happens when one service on a higher level of abstraction aggregates services on a lower level of abstraction.

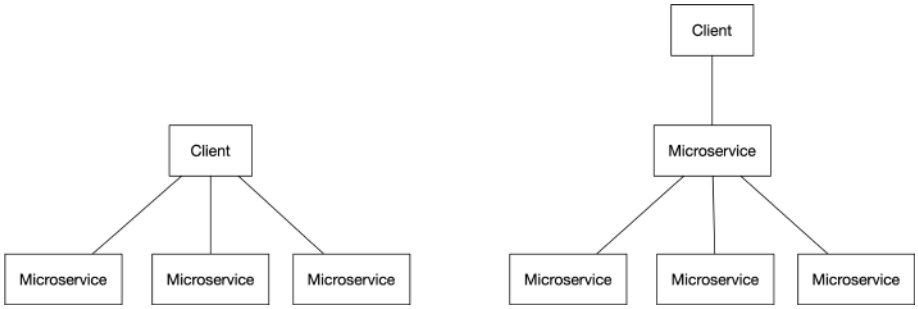

Figure 3.3. Architecture Without and With Service Aggregation

Let's have a service aggregation example with a second-hand e-commerce software system that allows people to sell their products online.

The problem domain of the e-commerce service consists of the following subdomains:

- User account domain
  - Create, modify, and delete a user account
  - View user account with sales items and orders
- Sales item domain
  - Add new sales items, modify, view, and delete sales items
- Shopping cart domain
  - Add/remove sales items to/from a shopping cart, empty a shopping cart
  - View the shopping cart with sales item details
- Order domain
  - Placing orders
    - Ensure payment
    - Create order
    - Remove ordered items from the shopping cart
    - Mark ordered sales items sold
    - Send order confirmation by email

    - View orders with sales item details
    - Update and delete orders

We should not implement all the subdomains in a single *ecommerce-service* because that would be too monolithic. We want to create microservices with a single responsibility. We can use service aggregation. We create a separate lower-level microservice for each subdomain. Then, we create a higher-level *ecommerce-service* microservice that aggregates those lower-level microservices.

We define that our *ecommerce-service* aggregates the following lower-level microservices:

- *user-account-service*
    - Create/Read/Update/Delete user accounts
- *sales-item-service*
    - Create/Read/Update/Delete sales items
- *shopping-cart-service*
    - View a shopping cart, add/remove sales items from a shopping cart, or empty a shopping cart
- *order-service*
    - Create/Read/Update/Delete orders
- *email-notification-service*
    - Send email notifications

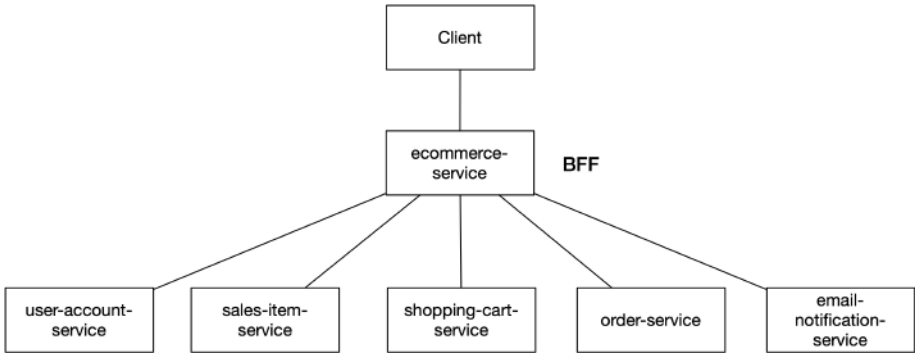

Figure 3.4. Service Aggregation in E-Commerce Software System

Most of the microservices described above can be implemented as REST APIs because they mainly contain basic CRUD (create, read, update, and delete) operations for which a REST API is a good match. We will handle API design in more detail in a later chapter dedicated to API design. Let's implement the *sales-item-service* as a REST API using Django and Django REST framework.

First, Create a directory for the Django project, and in that directory, we create a virtual environment:

Figure 3.5

```
python -m venv venv
```

Activate the virtual environment in Windows:

Figure 3.6

```
venv\Scripts\activate
```

Or activate the virtual environment in MacOS/Linux:

Figure 3.7

```
source venv/bin/activate
```

Install dependencies:

Figure 3.8

```
pip install django djangorestframework
```

Create new Django project and app:

Figure 3.9

```
django-admin startproject salesitemservice .
python manage.py startapp salesitems
```

We will first implement the `SalesItem` model class, which contains properties like `name` and `price`.

Figure 3.10

```
from django.core.validators import MaxValueValidator,
MinValueValidator
from django.db import models


class SalesItem(models.Model):
    user_account_id = models.BigIntegerField()
    name = models.CharField(max_length=512)
    price = models.IntegerField(
```

```
        validators=[MinValueValidator(1),
MaxValueValidator(2147483647)]
    )
```

Next, we will implement a serializer for the `SalesItem` model. In the serializer class, we list the wanted model fields to be serialized by their names. This is good from a security point of view. We should not use the `fields = '__all__'`, because if we added some internal or confidential fields to the model, they would be automatically serialized and sent to clients, exposing internal information to clients. It is safer to list the serialized fields explicitly.

Figure 3.11

```
from rest_framework import serializers
from .models import SalesItem


class SalesItemSerializer(serializers.ModelSerializer):
    class Meta:
        model = SalesItem
        fields = ['id', 'user_account_id', 'name', 'price']
```

Finally, we implement the `SalesItemViewSet` class, which defines API endpoints for creating, getting, updating, and deleting sales items. In the below *views.py* module, we need to implement only the `list` method. Other methods come from the `viewsets.ModelViewSet` base class.

Figure 3.12

```
from typing import Any

from rest_framework import viewsets
from rest_framework.request import Request
from rest_framework.response import Response

from .models import SalesItem
from .serializers import SalesItemSerializer


class SalesItemViewSet(viewsets.ModelViewSet):
    queryset = SalesItem.objects.all()
    serializer_class = SalesItemSerializer

    def list(
            self, request: Request, *args: tuple[Any], **kwargs:
dict[str, Any]
    ) -> Response:
        user_account_id =
request.query_params.get('userAccountId')

        queryset = (
```

```
            SalesItem.objects.all()
            if user_account_id is None
            else
SalesItem.objects.filter(user_account_id=user_account_id)
        )

        serializer = SalesItemSerializer(queryset, many=True)
        return Response(serializer.data)
```

We also need to update the *urls.py* file in the Django project to contain the following:

Figure 3.13

```
from django.urls import include, path
from rest_framework import routers
from salesitems.views import SalesItemViewSet

router = routers.DefaultRouter(trailing_slash=False)
router.register('sales-items', SalesItemViewSet)

urlpatterns = [
    path('', include(router.urls)),
]
```

Before running the microservice, the database needs to be set up:

Figure 3.14

```
python manage.py makemigrations
python manage.py migrate
```

You can run the *sales-item-service* with the following command:

Figure 3.15

```
python manage.py runserver
```

The REST API will be accessible at *http://127.0.0.1:8000/*

> In the above examples, I used idiomatic Django by defining models in *models.py* file, serializers in *serializers.py* file and views in *views.py* file. Instead, you could define each class in its own file and name the file according to the class name. That is the best approach to ensure a single responsibility for each module. For module names containing a class definition, I use *CapWords* (or *PascalCase*). This is against the PEP 8 style guide, but it is the only deviation from PEP 8 I am making in this book. You can, of course, follow the PEP 8, but there are two reasons for the approach I am using:

- The module name becomes more insightful and tells you that it contains a single public class definition, and the module name tells the name of the class. For example, if you have a module named `OrderService.py`, you can expect that a class named `OrderService` can be imported from it. And that there is nothing else public in that module.
- If you export an instance of the class from a module, that module should be named in the snake case. For example, If you have a module with a private `__OrderService` class and export of an `order_service` variable (singleton) that is an instance of the `__OrderService` class, you should name the module as *order_service.py*. Now the *order_service.py* module name tells everyone that a variable named `order_service` could be imported from that module. And that should be the only public thing exportable from the module.

If you want to name modules according to PEP 8, name modules containing a single class definition with the class name but converted to snake_case. For example, an *order_service.py* module contains an `OrderService` class definition.

Let's get back to the Django example. If you have several models and put them all into the *models.py* file, the file size will grow, and finding the wanted class is no longer easy. A better option is to create a *models* directory (a package) and put individual model classes into separate modules in that directory. Finding the wanted model would be easy because the models are automatically listed in alphabetical order in the file browser of your IDE. You don't get automatic alphabetical ordering of multiple class definitions if they are defined in a single file.

The same approach applies to modules that contain multiple functions. Suppose we have a *utils.py* module containing various utility functions. Once again, a better option is to create a directory named *utils* and put individual functions into their own files. You can then quickly locate the wanted utility function by looking at the *utils* directory contents. You can even create subdirectories to make the structure hierarchical, like a *string* directory under the *utils* directory for utility functions related to strings. A single file should contain a single public function, but it can additionally contain multiple private functions that the public function utilizes. The next chapter will discuss more about the *single responsibility principle*.

Below is defined how the higher-level *ecommerce-service* will orchestrate the use of the aggregated lower-level microservices:

- User account domain
  - Delegates CRUD operations to *user-account-service*
  - Delegates to *sales-item-service* to fetch information about user's sales items

    - Delegates to *order-service* to fetch information about user's orders
- Sales item domain
    - Delegates CRUD operations to *sales-item-service*
- Shopping cart domain
    - Delegates read/add/remove/empty operations to *shopping-cart-service*
    - Delegates to *sales-item-service* to fetch information about the sales items in the shopping cart
- Order domain
    - Ensures that the payment gateway confirms payment
    - Delegates CRUD operations to *order-service*
    - Delegates to *shopping-cart-service* to remove bought items from the shopping cart
    - Delegates to *sales-item-service* for marking sales items bought
    - Delegates to *email-notification-service* for sending order confirmation email
    - Delegates to *sales-item-service* to fetch information about order's sales items

The *ecommerce-service* is meant to be used by frontend clients, like a web clients. *Backend for Frontend* (BFF) term describes a microservice designed to provide an API for frontend clients. Service aggregation is a generic term compared to the BFF term, and there need not be a frontend involved. You can use service aggregation to create an aggregated microservice used by another microservice or microservices. There can even be multiple levels of service aggregation if you have a large and complex software system.

Clients can have different needs regarding what information they want from an API. For example, a mobile client might be limited to exposing only a subset of all information available

from the API. In contrast, a web client can fetch all information.

All of the above requirements are something that a GraphQL-based API can fulfill. Therefore, it is wise to implement the *ecommerce-service* using GraphQL. I have chosen the Ariadne library to implement a single GraphQL query in the *ecommerce-service*. Below is the implementation of the `user` query, which fetches data from three microservices. It fetches the user account information from the *user-account-service*, the user's sales items from the *sales-item-service*, and finally, the user's orders from the *order-service*.

Let's create a new Python project and install the following dependencies:

Figure 3.16

```
pip install ariadne httpx hypercorn
```

Next, we create the GraphQL server supporting a single `user` query:

Figure 3.17

```
from asyncio import gather
import os

from ariadne import QueryType, gql, make_executable_schema
from ariadne.asgi import GraphQL
from httpx import AsyncClient

query = QueryType()

type_defs = gql(
    """
  type UserAccount {
    id: ID!,
    userName: String!
    # Define additional properties...
  }

  type SalesItem {
    id: ID!,
    name: String!
    # Define additional properties...
  }

  type Order {
    id: ID!,
    userId: ID!
    # Define additional properties...
  }

  type User {
```

```
    userAccount: UserAccount!
    salesItems: [SalesItem!]!
    orders: [Order!]!
  }

  type Query {
    user(id: ID!): User!
  }
"""
)

USER_ACCOUNT_SERVICE_URL =
os.environ.get('USER_ACCOUNT_SERVICE_URL')
SALES_ITEM_SERVICE_URL =
os.environ.get('SALES_ITEM_SERVICE_URL')
ORDER_SERVICE_URL = os.environ.get('ORDER_SERVICE_URL')


@query.field('user')
async def resolve_user(_, info, id):
    async with AsyncClient() as client:
        [
            user_account_service_response,
            sales_item_service_response,
            order_service_response,
        ] = await gather(
            client.get(f'{USER_ACCOUNT_SERVICE_URL}/user-
accounts/{id}'),
            client.get(
                f'{SALES_ITEM_SERVICE_URL}/sales-items?
userAccountId={id}'
            ),
            client.get(f'{ORDER_SERVICE_URL}/orders?
userAccountId={id}'),
        )

    user_account_service_response.raise_for_status()
    sales_item_service_response.raise_for_status()
    order_service_response.raise_for_status()

    return {
        'userAccount': user_account_service_response.json(),
        'salesItems': sales_item_service_response.json(),
        'orders': order_service_response.json(),
    }


schema = make_executable_schema(type_defs, query)
app = GraphQL(schema, debug=True)
```

To start the GraphQL server, we need an ASGI web server (e.g., hypercorn). You can run the GraphQL server:

Figure 3.18

```
export SALES_ITEM_SERVICE_URL=http://127.0.0.1:8000
export USER_ACCOUNT_SERVICE_URL=...
export ORDER_SERVICE_URL=...
hypercorn app:app -b 127.0.0.1:5000
```

You can access the *GraphiQL* UI at *http://127.0.0.1:5000/graphql* On the left-hand side pane, you can specify a GraphQL query. For example, to query the user identified with an id 2:

Figure 3.19

```
{
  user(id: 2) {
    userAccount {
      id
      userName
    }
    salesItems {
      id
      name
    }
    orders {
      id
      userId
    }
  }
}
```

Because we have only implemented the *sales-item-service* lower-level microservice and not all the lower-level microservices, let's modify the *app.py* to return dummy static results instead of accessing the non-existent lower-level microservices:

Figure 3.20

```
from asyncio import gather
import os

from ariadne import QueryType, gql, make_executable_schema
from ariadne.asgi import GraphQL
from httpx import AsyncClient, Response

query = QueryType()

type_defs = gql(
    """
  type UserAccount {
    id: ID!,
    userName: String!
    # Define additional properties...
  }

  type SalesItem {
    id: ID!,
    name: String!
    # Define additional properties...
  }

  type Order {
    id: ID!,
```

```
    userId: ID!
    # Define additional properties...
  }

  type User {
    userAccount: UserAccount!
    salesItems: [SalesItem!]!
    orders: [Order!]!
  }

  type Query {
    user(id: ID!): User!
  }
"""
)

SALES_ITEM_SERVICE_URL =
os.environ.get('SALES_ITEM_SERVICE_URL')


async def getUserAccount(id):
    return Response(200, json={'id': id, 'userName': 'Petri'})


async def getOrders(id):
    return Response(200, json=[{'id': 1, 'userId': id}])


@query.field('user')
async def resolve_user(_, info, id):
    async with AsyncClient() as client:
        [
            user_account_service_response,
            sales_item_service_response,
            order_service_response,
        ] = await gather(
            getUserAccount(id),
            client.get(
                f'{SALES_ITEM_SERVICE_URL}/sales-items?
userAccountId={id}'
            ),
            getOrders(id),
        )

    return {
        'userAccount': user_account_service_response.json(),
        'salesItems': sales_item_service_response.json(),
        'orders': order_service_response.json(),
    }


schema = make_executable_schema(type_defs, query)
app = GraphQL(schema, debug=True)
```

We should see the result below if we now execute the previously specified query. We assume that *sales-item-service* returns a single sales item with id 1.

Figure 3.21

```
{
  "data": {
    "user": {
      "userAccount": {
        "id": "2",
        "userName": "Petri"
      },
      "salesItems": [
        {
          "id": "1",
          "name": "Sales item 1"
        }
      ],
      "orders": [
        {
          "id": "1",
          "userId": "2"
        }
      ]
    }
  }
}
```

We can simulate a failure by modifying the *app.py* to start the app with the wrong URL for the *sales-item-service* (port 8000 is changed to 8001):

Figure 3.22

```
export SALES_ITEM_SERVICE_URL=http://127.0.0.1:8001
hypercorn app:app -b 127.0.0.1:5000
```

Now, if we execute the query again, we will get the below error response because the GraphQL server cannot connect to a service at `localhost:8001` because no service runs at that address.

Figure 3.23

```
{
  "data": null,
  "errors": [
    {
      "message": "All connection attempts failed",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "user"
      ],
      "extensions": {
        "exception": {
          "stacktrace": [ ...
          ],
```

```
          "context": {
            "mapped_exc": "<class 'httpx.ConnectError'>",
            "from_exc": "<class 'httpc...rotocolError'>",
            "to_exc": "<class 'httpx...rotocolError'>",
            "message": "'All connecti...tempts failed'"
          }
        }
      }
    }
  ]
}
```

You can also query a user and specify the query to return only a subset of fields. The query below does not return identifiers and orders. The server-side GraphQL library automatically includes only requested fields in the response. You, as a developer, do not have to do anything. You can optimize your microservice to fetch only the requested fields from the database if you desire.

Figure 3.24

```
{
  user(id: 2) {
    userAccount {
      userName
    }
    salesItems {
      name
    }
  }
}
```

The result for the above query will be the following:

Figure 3.25

```
{
  "data": {
    "user": {
      "userAccount": {
        "userName": "pksilen"
      },
      "salesItems": [
        {
          "name": "sales item 1"
        }
      ]
    }
  }
}
```

The above example lacks some features like authorization that is needed for production. Authorization should ensure users

can only execute the `user` query to fetch their resources. The authorization should fail if a user tries to execute the `user` query using someone else's `id`. Security will be discussed more in the coming *Security principles* chapter.

The `user` query in the previous example spanned over multiple lower-level microservices: *user-account-service*, *sales-item-service*, and *order-service*. Because the query is not mutating anything, it can be executed without a distributed transaction. A distributed transaction is similar to a regular (database) transaction, with the difference that it spans multiple remote services.

The API endpoint for placing an order in the *ecommerce-service* needs to create a new order using the *order-service*, mark purchased sales items as bought using the *sales-item-service*, empty the shopping cart using the *shopping-cart-service*, and finally send order confirmation email using the *email-notification-service*. These actions need to be wrapped inside a distributed transaction because we want to be able to roll back the transaction if any of these operations fail. Guidance on how to implement a distributed transaction is given later in this chapter.

Service aggregation utilizes the facade pattern. The facade pattern allows for hiding individual lower-level microservices behind a facade (the higher-level microservice). The clients of the software system access the system through the facade. They don't directly contact the individual lower-level microservices behind the facade because it breaks the encapsulation of the lower-level microservices inside the higher-level microservice. A client accessing the lower-level microservices directly creates unwanted coupling between the client and the lower-level microservices, which makes changing the lower-level microservices hard without affecting the client.

Think about a post office counter as an example of a real-world facade. It serves as a facade for the post office and when you need to receive a package, you communicate with that facade (the post office clerk at the counter). You have a simple interface that just requires telling the package code, and the

clerk will find the package from the correct shelf and bring it to you. If you hadn't that facade, it would mean that you would have to do lower-level work by yourself. Instead of just telling the package code, you must walk to the shelves and try to find the proper shelf where your package is located, make sure that you pick the correct package, and then carry the package by yourself. In addition to requiring more work, this approach is more error-prone. You can accidentally pick someone else's package if you are not pedantic enough. And think about the case when you go to the post office next time and find out that all the shelves have been rearranged. This wouldn't be a problem if you used the facade.

Service aggregation, where a higher-level microservice delegates to lower-level microservices, also implements the bridge pattern. A higher-level microservice provides only some high-level control and relies on the lower-level microservices to do the actual work.

Service aggregation allows using more design patterns from the object-oriented design world. The most useful design patterns in the context of service aggregation are:

- Decorator pattern
- Proxy pattern
- Adapter pattern

We will discuss design patterns in the next chapter, but I want to give you some examples of the above three design patterns used in conjunction with the *ecommerce-service*.

Decorator pattern can be used to add functionality in a higher-level microservice for lower-level microservices. One example is adding audit logging in a higher-level microservice. For example, you can add audit logging for requests in the *ecommerce-service*. You don't need to implement the audit logging separately in all the lower-level microservices.

Proxy pattern can be used to control the access from a higher-level microservice to lower-level microservices. Typical examples of the proxy pattern are authorization and caching. For example, you can add authorization and caching for

requests in the *ecommerce-service*. Only after successful authorization will the requests be delivered to the lower-level microservices. If a request's response is not found in the cache, the request will be forwarded to the appropriate lower-level microservice. You don't need to implement authorization and caching separately in all the lower-level microservices.

Adapter pattern allows a higher-level microservice to adapt to different versions of the lower-level microservices while maintaining the API towards clients unchanged.

3.6: High Cohesion, Low Coupling Principle

> ***A software system should consist of services with high cohesion and low coupling.***

Cohesion refers to the degree to which classes inside a service belong together. Coupling refers to how many other services a service is interacting with. When following the *single responsibility principle*, it is possible to implement services as microservices with high cohesion. Service aggregation adds low coupling. Microservices and service aggregation together enable high cohesion and low coupling, which is the target of good architecture. If there were no service aggregation, lower-level microservices would need to communicate with each other, creating high coupling in the architecture. Also, clients would be coupled with the lower-level microservices. For example, in the e-commerce example, the *order-service* would be coupled with almost all the other microservices. And if the *sales-item-service* API changed, in the worst case, a change would be needed in three other microservices. When using service aggregation, lower-level microservices are coupled only to the higher-level microservice.

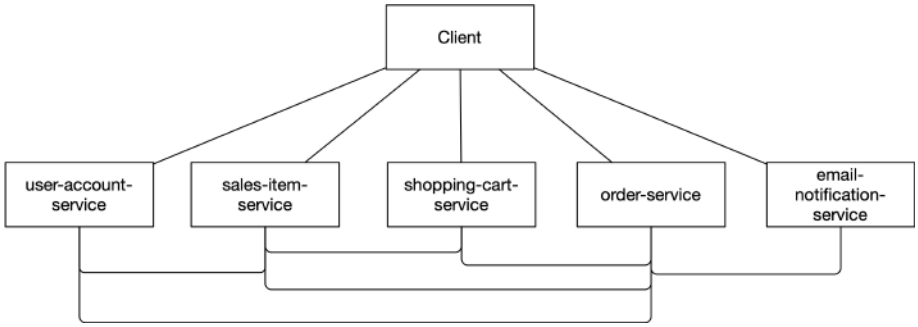

Figure 3.26. E-Commerce Software System With High Coupling

Low coupling means that the development of services can be highly parallelized. In the e-commerce example, the five lower-level microservices don't have coupling with each other. The development of each of those microservices can be isolated and assigned to a single team member or a group of team members. The development of the lower-level microservices can proceed in parallel, and the development of the higher-level microservice can start when the API specifications of the lower-level microservices become stable enough. The target should be to design the lower-level microservices APIs early on to enable the development of the higher-level microservice.

3.7: Library Composition Principle

> ***Higher-level libraries should be composed of lower-level libraries.***

Suppose you need a library for parsing configuration files (in particular syntax) in YAML or JSON format. In that case, you can first create the needed YAML and JSON parsing libraries (or use existing ones). Then, you can create the configuration file parsing library, which is composed of the YAML and JSON parsing libraries. You would then have three different libraries: one higher-level library and two lower-level libraries. Each library has a single responsibility: one for parsing JSON, one for parsing YAML, and one for parsing configuration files with a specific syntax, either in JSON or YAML. Software components can now use the higher-level library for parsing

configuration files, and they need not be aware of the JSON/YAML parsing libraries at all.

3.8: Avoid Duplication Principle

> ***Avoid software duplication at the software system and service level.***

Duplication at the software system level happens when two or more software systems use the same services. For example, two different software systems can both have a message broker, API gateway, identity and access management (IAM) application, and log and metrics collection services. You could continue this list even further. The goal of duplication-free architecture is to have only one deployment of these services. Public cloud providers offer these services for your use. If you have a Kubernetes cluster, an alternative solution is to deploy your software systems in different Kubernetes Namespaces and deploy the common services to a shared Kubernetes namespace, which can be called the *platform* or *common-services*, for example.

Duplication at the service level happens when two or more services have common functionality that could be extracted to a separate new microservice. For example, consider a case where both *user-account-service* and *order-service* have the functionality to send notification messages by email to a user. This email-sending functionality is duplicated in both microservices. Duplication can be avoided by extracting the email-sending functionality to a separate new microservice. The single responsibility of the microservices becomes more evident when the email-sending functionality is extracted to its own microservice. Another alternative is extracting the common functionality to a library. This is not the best solution because microservices become dependent on the library. When changes to the library are needed (e.g., security updates), you must change the library version in all the microservices using the library and then test all the affected microservices.

When a company develops multiple software systems in several departments, the software development typically happens in silos. The departments are not necessarily aware of what the other departments are doing. For example, it might be possible that two departments have both developed a microservice for sending emails. There is now software duplication that no one is aware of. This is not an optimal situation. A software development company should do something to enable collaboration between the departments and break the silos. One good way to share software is to establish shared folders or organizations in the source code repository hosting service that the company uses. For example, in GitHub, you could create an organization for sharing source code repositories for common libraries and another for sharing common services. Each software development department has access to those common organizations and can still develop its software inside its own GitHub organization. In this way, the company can enforce proper access control for the source code of different departments, if needed. When a team needs to develop something new, it can first consult the common source code repositories to find out if something is already available that can be reused as such or extended.

3.9: Externalized Service Configuration Principle

> ***The configuration of a service should be externalized. It should be stored in the environment where the service is running, not in the source code. The externalized configuration makes the service adaptable to different environments and needs.***

Service configuration means any data that varies between service deployments (different environments, different customers, etc.). The following are typical places where externalized configuration can be stored when software is running in a Kubernetes cluster:

- Environment variables

- Kubernetes ConfigMaps
- Kubernetes Secrets

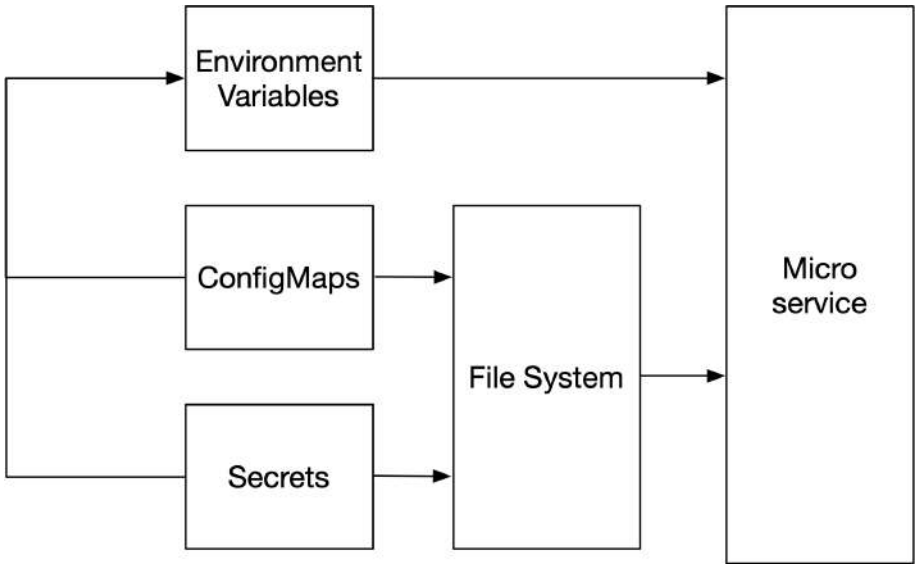

Figure 3.27. Configuration Storage Options

We will discuss these three configuration storage options in the following sections.

3.9.1: Environment Variables

Environment variables can be used to store configuration as simple key-value pairs. They are typically used to store information like connection information (how to connect to dependent services, like a database or message broker) or a microservice's logging level. Environment variables are available for the running process of the microservice, which can access the environment variable values by their names (keys).

You should not hardcode the default values of environment variables in the source code. This is because the default values are typically not for a production but for a development environment. Suppose you deploy a service to a production environment and forget to set all the needed environment variables. In that case, your service will have some environment variables with default values unsuitable for a production environment.

You can supply environment variables for a microservice in environment-specific *.env* files. For example, you can have a *.env.dev* file for storing environment variable values for a development environment and a *.env.ci* file for storing environment variable values used in the microservice's *continuous integration* (CI) pipeline. The syntax of *.env* files is straightforward. There is one environment variable defined per line:

Figure 3.28

```
NODE_ENV=development
HTTP_SERVER_PORT=3001
LOG_LEVEL=INFO
MONGODB_HOST=localhost
MONGODB_PORT=27017
MONGODB_USER=
MONGODB_PASSWORD=
```

Figure 3.29

```
NODE_ENV=integration
HTTP_SERVER_PORT=3001
LOG_LEVEL=INFO
MONGODB_HOST=localhost
MONGODB_PORT=27017
MONGODB_USER=
MONGODB_PASSWORD=
```

When a software component is deployed to a Kubernetes cluster using the Kubernetes package manager Helm, environment variable values should be defined in the Helm chart's *values.yaml* file:

Figure 3.30

```
nodeEnv: production
httpServer:
  port: 8080
database:
  mongoDb:
    host: my-service-mongodb
    port: 27017
```

The values in the above *values.yaml* file can be used to define environment variables in a Kubernetes Deployment using the following Helm chart template:

Figure 3.31

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-service
```

```
spec:
  template:
    spec:
      containers:
        - name: my-service
          env:
            - name: NODE_ENV
              value: {{ .Values.nodeEnv }}
            - name: HTTP_SERVER_PORT
              value: "{{ .Values.httpServer.port }}"
            - name: MONGODB_HOST
              value: {{ .Values.database.mongoDb.host }}
            - name: MONGODB_PORT
              value: {{ .Values.database.mongoDb.port }}
```

When Kubernetes starts a microservice Pod, the following environment variables will be made available for the running container:

Figure 3.32

```
NODE_ENV=production
HTTP_SERVER_PORT=8080
MONGODB_HOST=my-service-mongodb
MONGODB_PORT=27017
```

3.9.2: Kubernetes ConfigMaps

A Kubernetes ConfigMap can store a configuration file or files in various formats, like JSON or YAML. These files can be mounted to the filesystem of a microservice's running container. The container can then read the configuration files from the mounted directory in its filesystem.

For example, you can have a ConfigMap for defining the logging level of a *my-service* microservice:

Figure 3.33

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-service
data:
  LOG_LEVEL: INFO
```

The below Kubernetes Deployment manifest defines that the content of the *my-service* ConfigMap's key `LOG_LEVEL` will be stored in a volume named `config-volume`, and the value of the `LOG_LEVEL` key will be stored in a file named `LOG_LEVEL`. After mounting the `config-volume` to the `/etc/config`

directory in a *my-service* container, it is possible to read the contents of the `/etc/config/LOG_LEVEL` file, which contains the text: INFO.

Figure 3.34

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-service
spec:
  template:
    spec:
      containers:
        - name: my-service
          volumeMounts:
            - name: config-volume
              mountPath: "/etc/config"
              readOnly: true
      volumes:
        - name: config-volume
          configMap:
            name: my-service
            items:
              - key: "LOG_LEVEL"
                path: "LOG_LEVEL"
```

In Kubernetes, editing of a ConfigMap is reflected in the respective mounted file. This means you can listen to changes in the */etc/config/LOG_LEVEL* file. Below is shown how to do it using the watchdog library:

Figure 3.35

```
from watchdog.events import FileSystemEventHandler
from watchdog.observers import Observer

from update_log_level import update_log_level


class UpdateLogLevelFsEventHandler(FileSystemEventHandler):
    def on_modified(self, event):
        try:
            with open('/etc/config/LOG_LEVEL', 'r') as file:
                new_log_level = file.read()
                # Check here that 'new_log_level'
                # contains a valid log level
                update_log_level(new_log_level)
        except:
            # Handler errors


update_log_level_fs_event_handler =
UpdateLogLevelFsEventHandler()
observer = Observer()

observer.schedule(
    update_log_level_fs_event_handler,
```

```
    path='/etc/config/LOG_LEVEL',
    recursive=False
)

observer.start()

# ...

# observer.stop()
# observer.join()
```

3.9.3: Kubernetes Secrets

Kubernetes Secrets are similar to ConfigMaps except that they are used to store sensitive information, like passwords and encryption keys.

Below is an example of a *values.yaml* file and a Helm chart template for creating a Kubernetes Secret. The Secret will contain two key-value pairs: the database username and password. The Secret's data needs to be Base64-encoded. In the below example, the Base64 encoding is done using the Helm template function `b64enc`.

Figure 3.36

```
database:
  mongoDb:
    host: my-service-mongodb
    port: 27017
    user: my-service-user
    password: Ak9(lKt41uF==%lLO&21mA#gL0!"Dps2
```

Figure 3.37

```
apiVersion: v1
kind: Secret
metadata:
  name: my-service
type: Opaque
data:
  mongoDbUser: {{ .Values.database.mongoDb.user | b64enc }}
  mongoDbPassword: {{ .Values.database.mongoDb.password | b64enc
}}
```

After being created, secrets can be mapped to environment variables in a Deployment manifest for a microservice. In the below example, we map the value of the secret key `mongoDbUser` from the `my-service` secret to an environment variable named `MONGODB_USER` and the value of the secret key

`mongoDbPassword` to an environment variable named `MONGODB_PASSWORD`.

Figure 3.38

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-service
spec:
  template:
    spec:
      containers:
        - name: my-service
          env:
            - name: MONGODB_USER
              valueFrom:
                secretKeyRef:
                  name: my-service
                  key: mongoDbUser

            - name: MONGODB_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: my-service
                  key: mongoDbPassword
```

When a *my-service* pod is started, the following environment variables are made available for the running container:

Figure 3.39

```
MONGODB_USER=my-service-user
MONGODB_PASSWORD=Ak9(lKt41uF==%lLO&21mA#gL0!"Dps2
```

3.10: Service Substitution Principle

> ***Make substituting a service's dependency for another service easy by making the dependencies transparent. A transparent service is exposed to other services by defining a host and port. Use externalized service configuration principle (e.g., environment variables) in your microservice to define the host and port (and possibly other needed parameters like a database username/password) for a dependent service.***

Let's have an example where a microservice depends on a MongoDB service. The MongoDB service should expose itself by defining a host and port combination. For the microservice,

you can specify the following environment variables for connecting to a *localhost* MongoDB service:

Figure 3.40

```
MONGODB_HOST=localhost
MONGODB_PORT=27017
```

Suppose that in a Kubernetes-based production environment, you have a MongoDB service in the cluster accessible via a Kubernetes Service named *my-service-mongodb*. In that case, you should have the environment variables for the MongoDB service defined as follows:

Figure 3.41

```
MONGODB_HOST=my-service-mongodb.default.svc.cluster.local
MONGODB_PORT=8080
```

Alternatively, a MongoDB service can run in the MongoDB Atlas cloud. In that case, the MongoDB service could be connected to using the following kind of environment variable values:

Figure 3.42

```
MONGODB_HOST=my-service.tjdze.mongodb.net
MONGODB_PORT=27017
```

As shown with the above examples, you can easily substitute a different MongoDB service depending on your microservice's environment. If you want to use a different MongoDB service, you don't need to modify the microservice's source code but only change the *externalized configuration*.

3.11: Inter-Service Communication Methods

Services communicate with each other using the following communication methods: synchronous, asynchronous, and shared data.

3.11.1: Synchronous Communication Method

A synchronous communication method should be used when a service communicates with another service and wants an immediate response. Synchronous communication can be implemented using protocols like HTTP or gRPC (which uses HTTP under the hood).

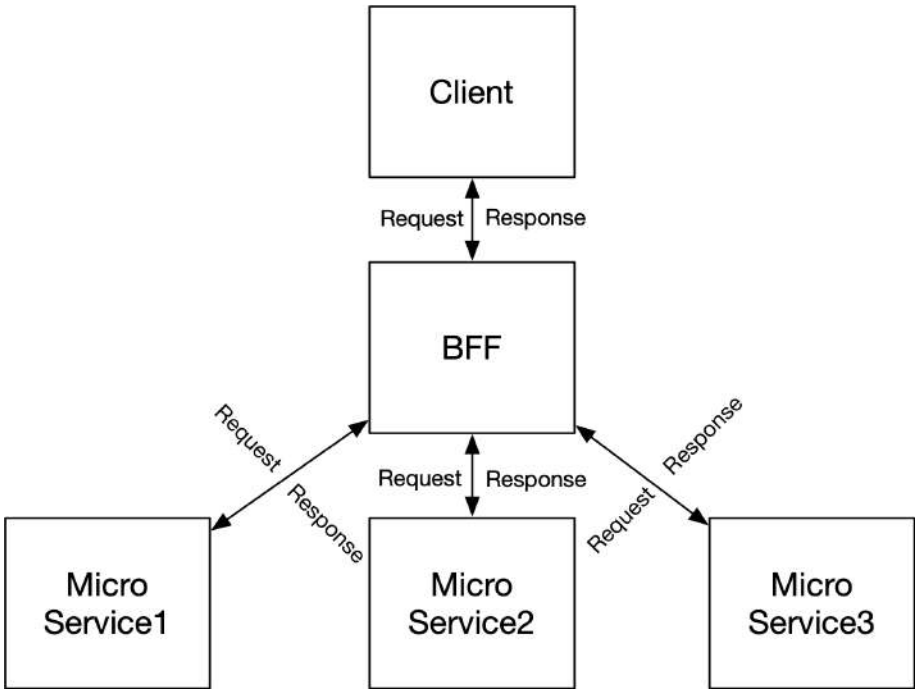

Figure 3.43. Synchronous Communication Method

In case of a failure when processing a request, the request processing microservice sends an error response to the requestor microservice. The requestor microservice can cascade the error up in the synchronous request stack until the initial request maker is reached. That initial request maker is often a client, like a web UI or mobile app. The initial request maker can then decide what to do. Usually, it will attempt to send the request again after a while (we are assuming here that the error is a transient server error, not a client error, like a bad request, for example)

3.11.2: Asynchronous Communication Method

When a service wants to deliver a request to another service but does not expect a response or at least not an immediate response, an asynchronous communication method should be

used. Some communication between services is asynchronous by nature. For example, a service might want to instruct an email notification service to email an end-user or to send an audit log entry to an audit logging service. Both examples can be implemented using an asynchronous communication method because no response for the operations is expected.

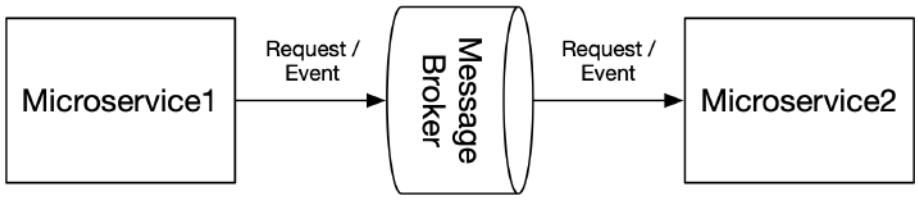

Figure 3.44. Request-Only Asynchronous Communication Method

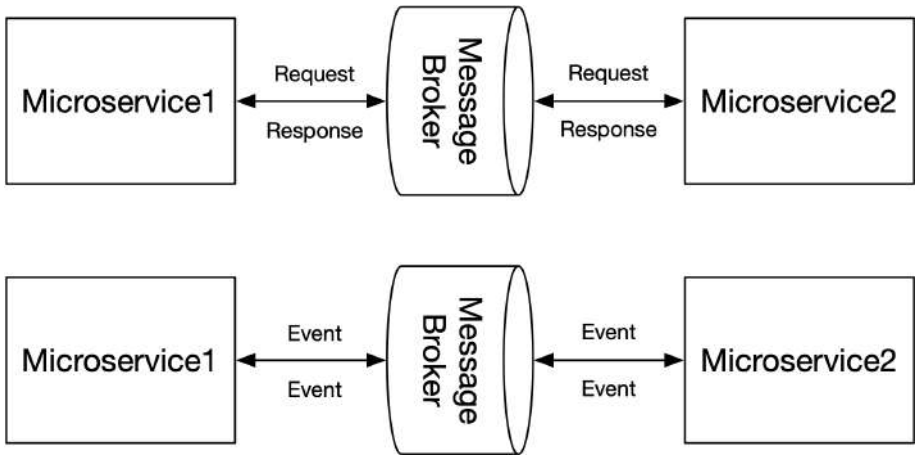

Figure 3.45. Request-Response Asynchronous Communication Method / Event Driven Architecture

Asynchronous communication can be implemented using a message broker. Services can produce messages to the message broker and consume messages from it. Several message broker implementations are available, like Apache Kafka, RabbitMQ, Apache ActiveMQ, and Redis. When a microservice produces a request to a message broker's topic, the producing microservice must wait for an acknowledgment from the message broker indicating that the request was successfully stored to multiple, or preferably all, replicas of the topic. Otherwise, there is no 100% guarantee that the request was successfully delivered in some message broker failure scenarios.

When an asynchronous request is of type fire-and-forget (i.e., no response is expected), the request processing microservice must ensure that the request will eventually get processed. If the request processing fails, the request processing microservice must reattempt the processing after a while. If a process termination signal is received, the request processing microservice instance must produce the request back to the message broker and allow some other microservice instance to fulfill the request. The rare possibility exists that the production of the request back to the message broker fails. Then, you could try to save the request to a persistent volume, for instance, but also that can fail. However, the likelihood of such a situation is very low.

The *API design principles* chapter describes designing APIs for inter-service communication in more detail.

3.11.3: Shared Data Communication Method

Service communication can also happen via shared data (e.g., a shared database). This method is useful with data-oriented services when storing the same data twice is not meaningful. Typically, one or more microservices produce the shared data, and other microservice(s) consume that data. The interface between these microservices is defined by the schema of the shared data, i.e., by the schemas of database tables. To secure the shared data, only the producing microservice(s) should have write access to the shared data, and the consuming microservice(s) should only have read access to the shared data.

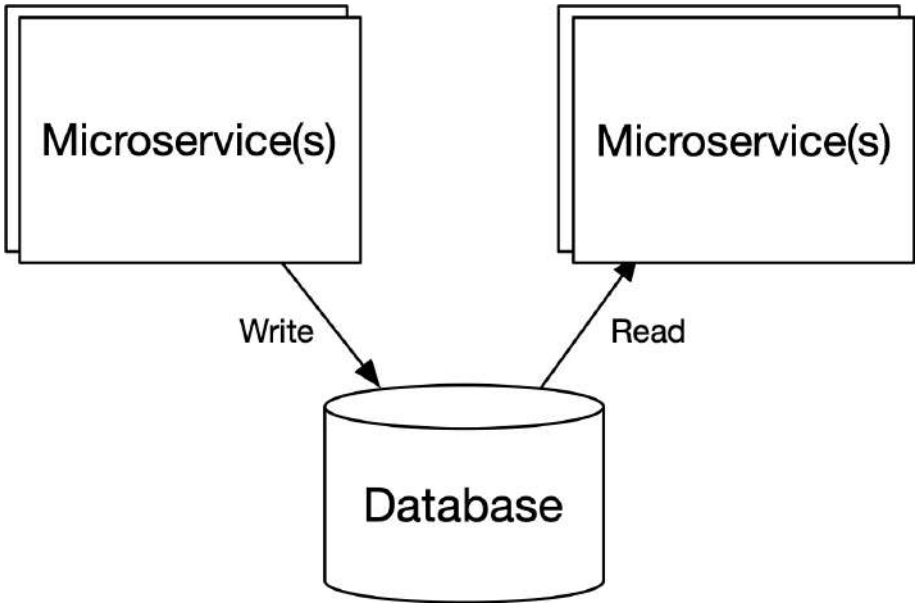

Figure 3.46. Shared Data Communication Method

3.12: Strategic Domain-Driven Design Principle

Design software system architecture by conducting domain-driven design (DDD) starting from the top of the software hierarchy (software system) and ending at the (micro)service (bounded context) level.

I often compare software system architectural design to the architectural design of a house. The house represents a software system. The entrances in the house's facade represent the software system's external interfaces. The rooms in the house are the microservices of the software system. Like a microservice, a single room usually has a dedicated purpose. The architectural design of a software system encompasses the definition of external interfaces, microservices, and their interfaces to other services.

The architectural design phase results in a ground plan for the software system. After the architectural design, you have the facade designed, and all the rooms are specified: the purpose of each room and how rooms interface with other rooms.

Designing an individual microservice is no longer architectural design, but it can be compared to the interior design of a single room. The microservice design is handled using object-oriented design principles, presented in the next chapter.

Domain-driven design (DDD) is a software design approach where software is modeled to match a problem/business domain according to input from the domain experts. Usually, these experts come from the business and specifically from product management. The idea of DDD is to transfer the domain knowledge from the domain experts to individual software developers so that everyone participating in software development can share a common language that describes the domain. The idea of the common language is that people can understand each other, and no multiple terms are used to describe a single thing. If you have a banking software system with an *account* entity, everyone should speak about *accounts*, not *money repositories*. This common language is also called the *ubiquitous language*.

The domain knowledge is transferred from product managers and architects to lead developers and product owners (POs) in development teams. The team's lead developer and PO share the domain knowledge with the rest of the team. This usually happens when the team processes backlog epics and features and splits them into user stories in planning sessions. A software development team can also have a dedicated domain expert or experts.

Strategical DDD starts from the top business/problem domain. The top domain is split into multiple *subdomains*, each on a lower abstraction level than the top domain. A domain should be divided into subdomains so there is minimal overlap between subdomains. A subdomain can further be divided into smaller subdomains. Subdomains will be interfacing with other subdomains when needed using well-defined interfaces. A subdomain implemented as a separate (micro)service is called a *bounded context*. You can also divide a bounded context into smaller pieces (subdomains). Those subdomains are manifested as software code placed in separate source code directories—more about that in the following chapters. Most DDD experts and books don't discuss multiple levels of

subdomains, but I like to use that term to describe what is inherent to any software system: hierarchy. For example, a *purchase-service* bounded context (a subdomain of an e-commerce software system) can have a subdomain for managing shopping carts and another subdomain that handles orders.

For example, a banking software system can have a bounded context for loan applications and another for making payments. The ubiquitous language in bounded contexts can differ. For example, consider an airline software system with the following bounded contexts: customers, ticketing, cargo, and airplane maintenance. The four bounded contexts can use the term *flight*, but the flights in the different bounded contexts differ. They are entities with the same name but with different attributes. Various strategies exist on how bounded contexts interface with each other. The interface between them can be jointly designed by two teams (a partnership) or one bounded context defines the interface. The interface provider can be sovereign (other bounded contexts must conform to the interface) or listen to the interface consumer needs. Perhaps the most modern and useful way to create an interface between two (or more) bounded contexts is for the interface provider to create an "open" API that can serve all interface consumers. The bounded context acting as an interface provider does not have to expose its model (entities and ubiquitous language) as such to other bounded contexts. It can use DTOs to map entities to clients' expected format and map the client format back to entities. Using DTOs enables better interfacing between different ubiquitous languages. Bounded context can create a so-called anticorruption layer for talking to another bounded context to translate its ubiquitous language into its own ubiquitous language. This is a good example of using the *adapter pattern*.

Strategical DDD has similarities with the microservice architecture. In both, something big is split into smaller, more manageable parts. DDD says that a single team should own a bounded context, and each bounded context should have its own source code repository (this will be automatically true if bounded contexts are microservices). A bounded context or a

microservice can become hard to manage and change if the responsibility is shared between multiple teams.

3.12.1: Design Example 1: Mobile Telecom Network Analytics Software System

Suppose an architecture team is assigned to design a mobile telecom network analytics software system. The team starts by defining the problem domain of the software system in more detail. When thinking about the system in more detail, they end up figuring out at least the following subdomains:

1. Ingesting raw data from various sources of the mobile telecom network
2. Transforming the ingested raw data into meaningful insights
3. Proper ways of presenting the insights to software system users

Let's pick up some keywords from the above definitions and formulate short names for the subdomains:

1. Ingesting raw data
2. Transforming raw data into insights
3. Presenting insights

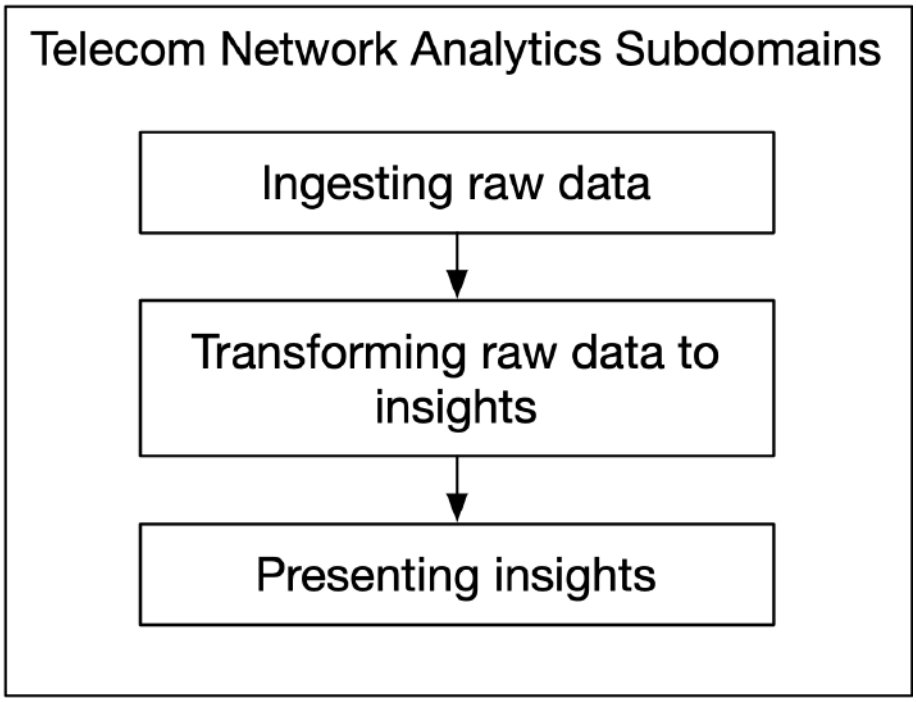

Figure 3.47. Subdomains

Let’s consider each of these three subdomains separately.

We know a mobile telecom network is divided into core and radio networks. From that, we can conclude that *Ingesting raw data* domain can be divided into further subdomains: *Ingesting radio network raw data* and *Ingesting core network raw data*. We can turn these two subdomains into applications for our software system: *Radio network data ingester* and *Core network data ingester*.

The *Transforming raw data to insights* domain should at least consist of an application aggregating the received raw data to counters and key performance indicators (KPIs). We can call that application *Data aggregator*.

The *Presenting insights* domain should contain a web application that can present insights in various ways, like using dashboards containing charts presenting aggregated counters and calculated KPIs. We can call this application *Insights visualizer*.

Now we have the following applications for the software system defined:

- Radio network data ingester
- Core network data ingester
- Data aggregator
- Insights visualizer

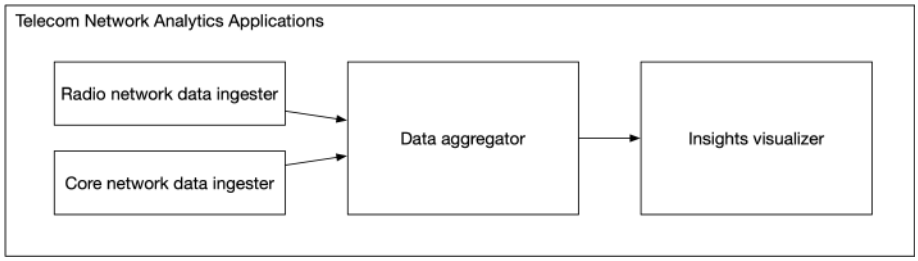

Figure 3.48. Applications

Next, we continue architectural design by splitting each application into one or more software components. (services, clients, and libraries). When defining the software components, we must remember to follow the *single responsibility principle*, *avoid duplication principle* and *externalized service configuration principle*.

When considering the *Radio network data ingester* and *Core network data ingester* applications, we can notice that we can implement them both using a single microservice, *data-ingester-service*, with different configurations for the radio and core network. This is because the data ingesting protocol is the same for radio and core networks. The two networks differ in the schema of the ingested data. If we have a single configurable microservice, we can avoid code duplication. The microservice and the two sets of configurations are our bounded contexts for the *Ingesting raw data* subdomain.

The *Data aggregator* application can be implemented using a single *data-aggregator-service* microservice that will be one more bounded context. We can use externalized configuration to define what counters and KPIs the microservice should aggregate and calculate from the raw data.

The *Insights visualizer* application consists of three different software components:

- A web client
- A service for fetching aggregated and calculated data (counters and KPIs)
- A service for storing the dynamic configuration of the web client

The dynamic configuration service stores information about what insights to visualize and how in the web client.

Microservices in the *Insights visualizer* application are:

- insights-visualizer-web-client
- insights-visualizer-data-service
- insights-visualizer-configuration-service

Now, we are ready with the microservice-level architectural design for the software system.

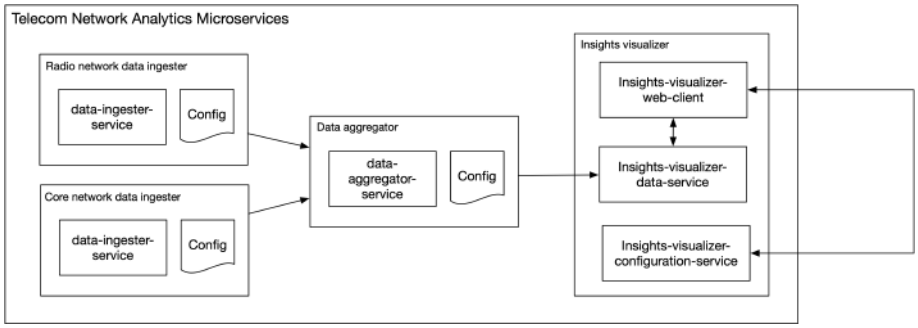

Figure 3.49. Microservices

The last part of architectural design is to define the inter-service communication methods. The *data-ingester-service* needs to send raw data to *data-aggregator-service*. Data is sent using asynchronous fire-and-forget requests and is implemented using a message broker. The communication between the *data-aggregator-service* and the *insights-visualizer-data-service* should use the *shared data* communication method because the *data-aggregator-service* generates aggregated data that the *insights-visualizer-data-service* uses. The communication between the *insights-visualizer-web-client* in the frontend and the *insights-visualizer-data-service* and *insights-visualizer-configuration-*

service in the backend is synchronous communication that can be implemented using an HTTP-based JSON-RPC, REST, or GraphQL API.

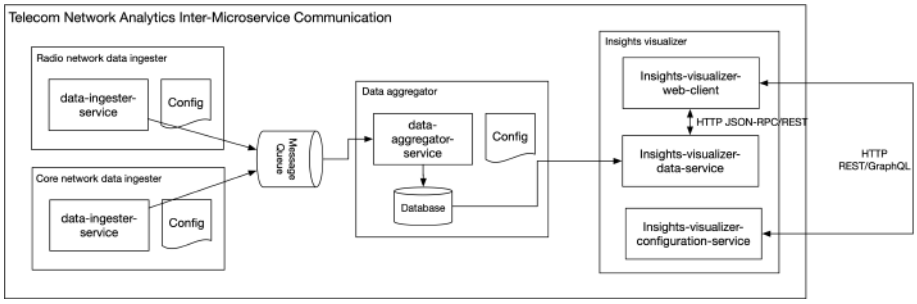

Figure 3.50. Inter-Microservice Communication

Next, design continues in development teams. Teams will specify the APIs between the microservices and conduct *tactical domain-driven design* and object-oriented design for the microservices. API design is covered in a later chapter, and object-oriented design, including tactical domain-driven design, is covered in the next chapter.

If the splitting of a software system into subdomains and bounded context is not straightforward, you can use *event storming* process to help you identify your software system's subdomains and bounded context. Event storming is particularly useful when the software system is extensive and its requirements are unclear. The basic idea of the event-storming process is to model your software system's domain events on a timeline and then group a set of domain events to form a subdomain or a bounded context. Below is an example of an event storming process result where *domain events* are listed as solid-line boxes along the time axis. After listing all the domain events, they are grouped into subdomains shown with dotted lines. Each subdomain can be further divided into one or more bounded contexts (not shown in the figure below). You can continue the event-storming process for each bounded context to figure out additional DDD concepts related to the domain events. The event-storming process is better described in the next chapter.

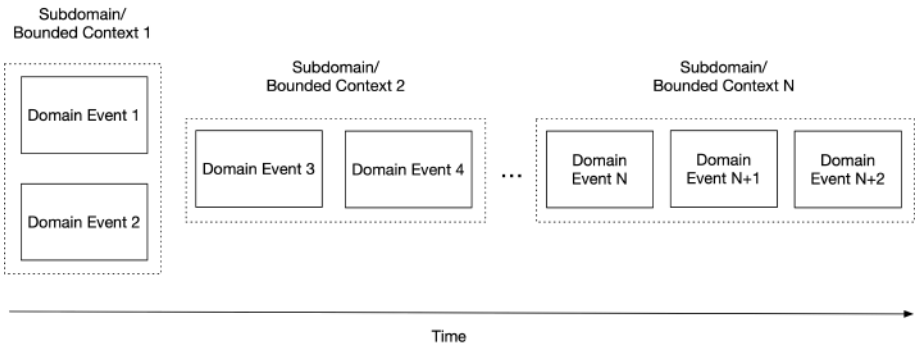

Figure 3.51. Event storming

3.12.2: Design Example 2: Banking Software System

Let's design a partial banking software system. The banking software system should be able to handle customers' loan applications and payments. The banking system problem domain can be divided into two subdomains:

1. Loan applications
2. Making payments

In the loan applications domain, a customer can submit a loan application. The eligibility for the loan will be assessed, and the bank can either accept the loan application and pay the loan or reject the loan application. In the making payments domain, a customer can make payments. Making a payment will withdraw money from the customer's account. It is also a transaction that should be recorded.

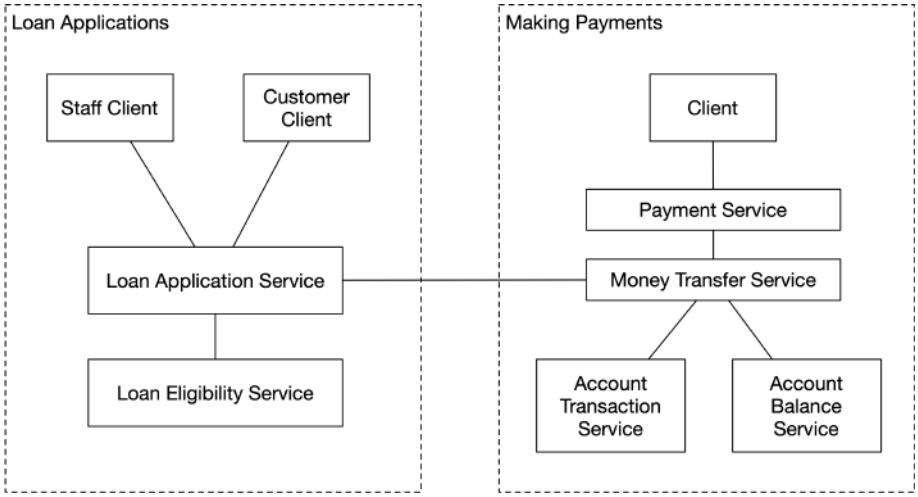

Figure 3.52. Banking Software System Bounded Contexts

Let's add a feature that a payment can be made to a recipient in another bank:

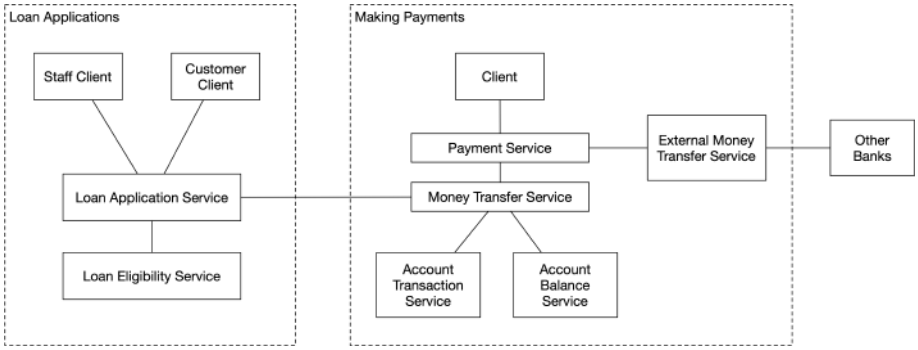

Figure 3.53. Banking Software System Bounded Contexts

Let's add another feature: money can be transferred from external banks to a customer's account.

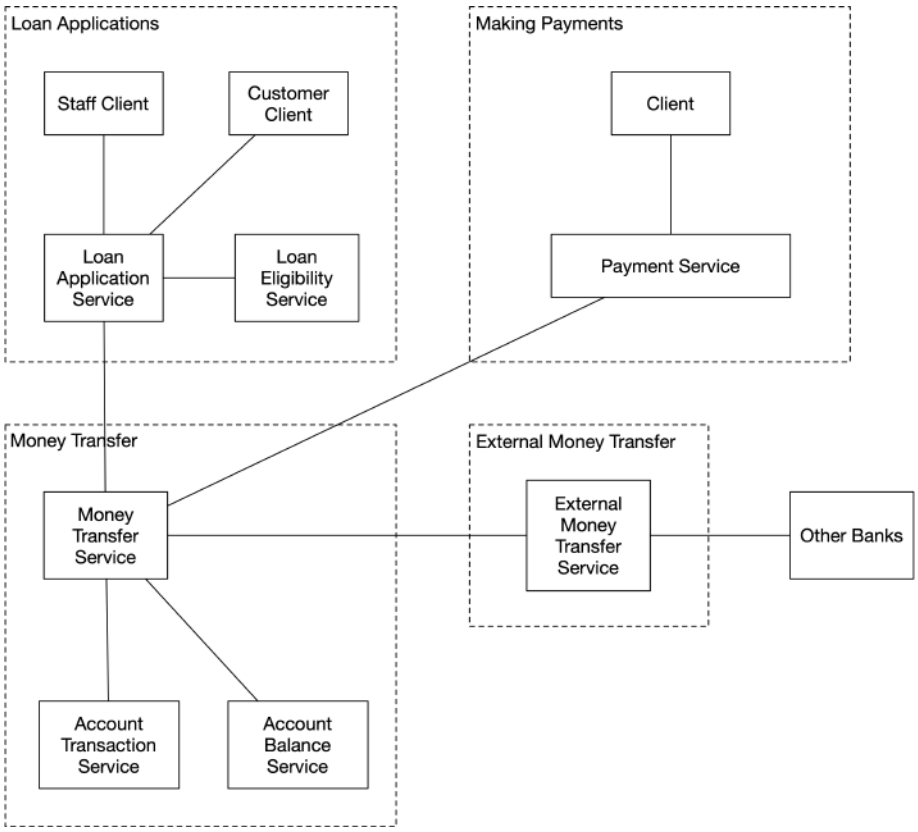

Figure 3.54. Banking Software System Bounded Contexts

As can be noticed from the above pictures, the architecture of the banking software system evolved when new features were introduced. For example, two new subdomains were created: money transfer and external money transfer. There was not so much change in the microservices (or bounded contexts) themselves, but how they are grouped logically to subdomains was altered.

3.13: Autopilot Microservices Principle

> ***Microservices should be architected to run on autopilot in their deployment environment.***

An autopilot microservice means a microservice that runs in a deployment environment without human interaction, except in

abnormal situations when the microservice should generate an alert to indicate that human intervention is required.

Autopilot microservices principle requires that the following sub-principles are followed:

- Stateless microservices principle
- Resilient microservices principle
- Horizontally autoscaling microservices principle
- Highly-available microservices principle
- Observable microservices principle

These principles are discussed in more detail next.

3.13.1: Stateless Microservices Principle

Microservices should be stateless to enable resiliency, horizontal scalability, and high availability.

A microservice can be made stateless by storing its state outside itself. The state can be stored in a data store that microservice instances share. Typically, the data store is a database or an in-memory cache (like Redis, for example).

3.13.2: Resilient Microservices Principle

Microservices should be resilient, i.e., quickly recover from failures automatically.

In a Kubernetes cluster, the resiliency of a microservice is handled by the Kubernetes control plane. If the computing node where a microservice instance is located needs to be decommissioned, Kubernetes will create a new instance of the microservice on another computing node and then evict the microservice from the node to be decommissioned.

What needs to be done in the microservice is to make it listen to Linux process termination signals, especially the *SIGTERM*

signal, which is sent to a microservice instance to indicate that it should terminate. Upon receiving a *SIGTERM* signal, the microservice instance should initiate a graceful shutdown. If the microservice instance does not shut down gracefully, Kubernetes will eventually issue a *SIGKILL* signal to terminate the microservice instance forcefully. The *SIGKILL* signal is sent after a termination grace period has elapsed. This period is, by default, 30 seconds, but it is configurable.

There are other reasons a microservice instance might be evicted from a computing node. One such reason is that Kubernetes must assign (for some reason which can be related to CPU/memory requests, for instance) another microservice to be run on that particular computing node, and your microservice won't fit there anymore and must be moved to another computing node.

If a microservice pod crashes, Kubernetes will notice that and start a new pod so that there is always the desired number of microservice replicas (pods/instances) running. The replica count can be defined in the Kubernetes Deployment manifest for the microservice.

But what if a microservice pod enters a deadlock and cannot serve requests? This situation can be remediated with the help of a liveness probe. You should always specify a liveness probe for each microservice Deployment. Below is an example of a microservice Deployment where an HTTP GET type liveness probe is defined:

Figure 3.55

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "microservice.fullname" . }}
spec:
  replicas: 1
  selector:
    matchLabels:
      {{- include "microservice.selectorLabels" . | nindent 6 }}
  template:
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.imageRegistry }}/{{
.Values.imageRepository }}:{{ .Values.im\
ageTag }}"
          livenessProbe:
```

```
      httpGet:
        path: /isAlive
        port: 8080
      initialDelaySeconds: 30
      failureThreshold: 3
      periodSeconds: 3
```

Kubernetes will poll the `/isAlive` HTTP endpoints of the microservice instances every three seconds (after the initial delay of 30 seconds reserved for the microservice instance startup). The HTTP endpoint should return the HTTP status code 200 *OK*. Suppose requests to that endpoint fail (e.g., due to a deadlock) three times in a row (defined by the `failureThreshold` property) for a particular microservice instance. In that case, the microservice instance is considered dead, and Kubernetes will terminate the pod and launch a new pod automatically.

The Kubernetes Deployment manifest should be modified when upgrading a microservice to a newer version. A new container image tag should be specified in the `image` property of the Deployment. This change will trigger an update procedure for the Deployment. By default, Kubernetes performs a rolling update, which means your microservice can serve requests during the update procedure with zero downtime.

Suppose you had defined one replica in the microservice Deployment manifest (as shown above with the `replicas: 1` property) and performed a Deployment upgrade (change the image to a newer version). In that case, Kubernetes would create a new pod using the new image tag, and only after the new pod is ready to serve requests will Kubernetes delete the pod running the old version. So, there is zero downtime, and the microservice can serve requests during the upgrade procedure.

If your microservice deployment had more replicas, e.g., 10, by default, Kubernetes would terminate a maximum of 25% of the running pods and start a maximum of 25% of the replica count new pods. The *rolling update* means updating pods in chunks, 25% of the pods at a time. The percentage value is configurable.

3.13.3: Horizontally Autoscaling Microservices Principle

> ***Microservices should automatically scale horizontally to be able to serve more requests.***

Horizontal scaling means adding new instances or removing instances of a microservice. Horizontal scaling of a microservice requires statelessness. Stateful services are usually implemented using sticky sessions so that requests from a particular client go to the same service instance. The horizontal scaling of stateful services is complicated because a client's state is stored on a single service instance. In the cloud-native world, we want to ensure even load distribution between microservice instances and target a request to any available microservice instance for processing.

Initially, a microservice can have one instance only. When the microservice gets more load, one instance cannot necessarily handle all the work. In that case, the microservice must be scaled horizontally (scaled out) by adding one or more new instances. When several microservice instances are running, the state cannot be stored inside the instances anymore because different client requests can be directed to different microservice instances. A stateless microservice must store its state outside the microservice in an in-memory cache or a database shared by all the microservice instances.

Microservices can be scaled manually, but that is rarely desired. Manual scaling requires someone to constantly monitor the software system and manually perform the needed scaling actions. Microservices should scale horizontally automatically. There are two requirements for a microservice to be horizontally auto-scalable:

- Microservice must be stateless
- There must be one or more metrics that define the scaling behavior

Typical metrics for horizontal autoscaling are CPU utilization and memory consumption. In many cases, using the CPU utilization metric alone can be enough. It is also possible to use a custom or external metric. For example, the Kafka consumer lag metric can indicate if the consumer lag is increasing and if a new microservice instance should be spawned to reduce the consumer lag.

In Kubernetes, you can specify horizontal autoscaling using the HorizontalPodAutoscaler (HPA):

Figure 3.56

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: my-service
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-service
  minReplicas: 1
  maxReplicas: 99
  metrics:
    - type: Resource
      resource:
        name: cpu
        targetAverageUtilization: 75
    - type: Resource
      resource:
        name: memory
        targetAverageUtilization: 75
```

In the above example, the *my-service* microservice is horizontally auto-scaled so that there is always at least one instance of the microservice running. There can be a maximum of 99 instances of the microservice running. The microservice is scaled out if CPU or memory utilization is over 75%, and it is scaled in (the number of microservice instances is reduced) when both CPU and memory utilization falls below 75%.

3.13.4: Highly-Available Microservices Principle

Business-critical microservices must be highly available.

If only one microservice instance runs in an environment, it does not make the microservice highly available. If something happens to that one instance, the microservice becomes temporarily unavailable until a new instance has been started and is ready to serve requests. For this reason, you should run at least two or more instances for all business-critical microservices. You should also ensure these two instances don't run on the same computing node. The instances should run in different availability zones of the cloud provider. Then, a catastrophe in availability zone 1 won't necessarily affect microservices running in availability zone 2.

You can ensure that no two microservice instances run on the same computing node by defining an anti-affinity rule in the microservice's Deployment manifest:

Figure 3.57

```
  .
  .
  .
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchLabels:
              app.kubernetes.io/name: {{ include
"microservice.name" . }}
          topologyKey: "kubernetes.io/hostname"
  .
  .
  .
```

For a business-critical microservice, we need to modify the horizontal autoscaling example from the previous section: The `minReplicas` property should be increased to 2:

Figure 3.58

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: my-service
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-service
  minReplicas: 2
  maxReplicas: 99
.
.
.
```

3.13.5: Observable Microservices Principle

> ***It should be possible to detect any abnormal behavior in deployed microservices immediately. Abnormal behavior should trigger an alert. The deployment environment should offer aids for troubleshooting abnormal behavior.***

A modern cloud-native software system consists of multiple microservices running simultaneously. No one can manually check the logs of tens or even hundreds of microservice instances. The key to monitoring microservices is automation. Everything starts with collecting relevant metrics from microservices and their execution environment. These metrics are used to define rules for automatic alerts that trigger when an abnormal condition occurs. Metrics are also used to create monitoring and troubleshooting dashboards, which can be used to analyze the state of the software system and its microservices after an alert is triggered.

In addition to metrics, to enable drill-down to a problem's root cause, distributed tracing should be implemented to log the communication between various microservices to troubleshoot inter-service communication problems. Each microservice must also log at least all errors and warnings. These logs should be fed to a centralized log collection system where querying the logs is made quick and easy.

3.14: Software Versioning Principles

In this section, the following principles related to software versioning will be presented:

- Use semantic versioning
- Avoid using 0.x versions
- Don't increase major versions
- Implement security patches and bug corrections to all major versions

- Avoid using *non*-LTS (Long Term Support) versions in production

3.14.1: Use Semantic Versioning Principle

Use semantic versioning for software components.

Semantic versioning means that given a version number in the format: `<MAJOR>.<MINOR>.<PATCH>`, increment the:

- *MAJOR* value when you make incompatible API changes
- *MINOR* value when you add functionality in a backward-compatible manner
- *PATCH* value when you make backward-compatible bug fixes or security patches

3.14.2: Avoid Using 0.x Versions Principle

If you are using 3rd party components, avoid or at least be thoughtful of using 0.x versioned components.

In semantic versioning, major version zero (0.x.y) is for initial development. Anything can change at any time. The public API should not be considered stable. Typically, software components with a zero major version are still in a proof of concept phase, and anything can change. When you want or need to take a newer version into use, you must be prepared for changes, and sometimes, these changes can be considerable, resulting in a lot of refactoring.

3.14.3: Don’t Increase Major Version Principle

In semantic versioning, you need to increase the major version when making backward-incompatible public API changes. But if possible and feasible, I advise not to make backward-incompatible changes, thus no major version increases.

If you need to make a backward-incompatible public API change, you should create a totally new software component with a different name. For example, suppose you have a *common-ui-lib* and must make backward-incompatible changes. In that case, I recommend adding the new major version number to the library name and publishing a new library with the name *common-ui-lib-2*. This protects developers from accidentally using a more recent non-compatible version when changing the used library version number. Library users don't necessarily know if a library uses semantic versioning properly or not. This information is not usually told in the library documentation, but it is a good practice to communicate it in the library documentation.

If a software component uses the *common-ui-lib*, the latest version of the library can always be safely taken into use because it won't contain any breaking changes, only new features, bug fixes, and security patches.

This would be always safe:

Figure 3.59

```
pip install --upgrade common-ui-lib
```

When you are ready to migrate to the new major version of the library, you can uninstall the old version and install the new major version in the following way:

Figure 3.60

```
pip uninstall common-ui-lib
pip install common-ui-lib-2
```

Consider when to create a new major version of a library. When you created the first library version, you probably did not get everything right in the public API. That is normal. It is challenging to create a perfect API the first time. Before releasing the second major version of the library, I suggest reviewing the new API with a team, collecting user feedback, and waiting long enough to get the API "close to perfect" the second time. No one wants to use a library with frequent backward-incompatible major version changes.

3.14.4: Implement Security Patches and Bug Corrections to All Major Versions Principle

If you have authored a library for others to use, do not force the users to take a new major version of the library into use just because it contains some bug corrections or security patches that are not available for the older major version(s). You should have a comprehensive set of automated tests to ensure that a bug fix or security patch doesn't break anything. Thus, making a security patch or bug fix in multiple branches or source code repositories should be straightforward.

Requiring library users to upgrade to a new major version to get some security patch or a bug correction can create a maintenance hell where the library users must refactor all software components using the library just to get a security patch or bug correction.

3.14.5: Avoid Using Non-LTS Versions in Production Principle

Some software is available as Long Term Support (LTS) and non-LTS versions. Always use only an LTS version in production. You are guaranteed long-term support through bug corrections and security patches. You can use a non-LTS version for proof of concept (PoC) projects where you want to use some new features unavailable in an LTS version. But you must remember that if the PoC succeeds, you can't just throw it into production. You need to productize it first, i.e., replace the non-LTS software with LTS software.

3.15: Git Version Control Principle

> ***Use trunk(= main branch) based development and develop software in feature branches merged into the main branch. Use feature toggles (or flags) when needed.***

Trunk-based development is suitable for modern software, which has an extensive set of automated functional and non-functional tests and can use feature toggles. There is also an older/legacy branching model called GitFlow, which can be used instead of trunk-based development to get better control of releasing software.

When you need to develop a new feature, it can be done using either of the following ways:

1. Using a feature branch
2. Using multiple feature branches and a feature toggle (or flag)

3.15.1: Feature Branch

The feature branch approach is enough for simple features encompassing a single program increment, team, and microservice. A new feature is developed in a feature branch created from the main branch, and when the feature is ready, the feature branch is merged back into the main branch, and the feature branch can be deleted if wanted. The feature branch should be merged using a merge or pull request that triggers a CI pipeline run that must succeed before the merge/pull request can be completed. The merge or pull request should also take care of the code review. There should also be a manual way to trigger a CI/CD pipeline run for the feature branch so developers can test an unfinished feature in a test environment during the development phase. The artifacts produced by a CI/CD pipeline run from a feature branch can be called *in-progress* artifacts, and they should be regularly cleaned (e.g., after 48 hours) from the artifact repository.

Below a sample workflow of creating and using a feature branch is depicted:

Figure 3.61

```
git clone <repository-url>
git checkout main
git pull --rebase --ff-only

# Create and checkout a feature branch for a feature with id
<feature-id>
# The feature id can be a JIRA id, for example
```

```
git checkout -b feature/<feature-id>

# Make your changes to code

# First commit
git commit -a -m "Commit message here..."

# Possibly more code changes and commits...

# Fetch the latest commits from remote main branch and
# fast forward your local main branch to match origin/main
git fetch origin main
git update-ref refs/heads/main refs/remotes/origin/main

# Rebase your feature branch on top of main
git rebase origin/main --autostash

# Push the feature branch to remote
# After successful push, in response to the below git command,
# you should receive a link to create a merge/pull request.
# Create a MR/PR, wait for the CI build to complete and
# perform the merge
git push -u origin feature/<feature-id>

# Other developers can now also use the feature branch
# because it is pushed to origin

# After successfully merging
git checkout main
git pull --rebase

# Now you are ready to create a new feature branch...
```

When the feature is ready, you can create a pull or merge request from the feature branch to the main branch. You can create the pull/merge request in your Git hosting service's web page or use the link in the output of the `git push` command. After creating the pull/merge request, a build pipeline should be started, and colleagues can review the code. The build started after creating the pull/merge request builds *candidate* artifacts, which are stored in the artifact repository but deleted after a certain period. If you need to change the code after making the pull/merge request, just modify the code, then use *git add*, *git commit —amend*, and *git push* commands to push the changes to the merge request. The merge can be completed after the code is reviewed and the build pipeline succeeds. After the merge, a build pipeline from the main branch should be run. This pipeline run should push the final *release* artifacts to the artifact repository.

3.15.2: Feature Toggle

A feature toggle works in a similar way to a feature license. In the case of a feature license, the feature is available only when a user has the respective license activated in their environment. A toggleable feature is available only when the feature toggle is switched on. Feature toggles should be used for complex features spanning multiple program increments, microservices, or teams. Feature toggles are part of the configuration of an environment. For example, feature toggles can be stored in a Kubernetes ConfigMap that any microservice can access. When using a feature toggle, the toggle is initially switched off. Development of the feature happens in multiple feature branches in different teams. Teams merge their part of the feature to the main branch. When all feature branches are merged into the main branch, the feature toggle can be switched on to activate the feature.

People who haven't used feature toggles may have some prejudice and misconceptions:

- Code becomes cluttered with feature toggles
    - Not all features need a toggle. Only those should have a toggle that need it. For example, if a feature is implemented but not yet 100% tested, a feature toggle is needed to keep the feature disabled until it is thoroughly tested
- Code becomes unreadable and cluttered with if-else statements
    - This can be true if the codebase is poorly designed and contains technical debt. (= When you need to apply *shotgun surgery* to *spaghetti code*)
    - Usually, implementing a feature toggle in well-designed code does not need changes in many places but just a single or few places
- Feature toggle causes performance degradation
    - Feature toggles can almost always be implemented with negligible performance degradation, e.g., using one or a few if-statements

- Dismantling feature toggles is extra effort and can cause bugs
    - First of all, do you need to remove them? Many times, feature toggles can be left in the codebase if they don't degenerate the readability or performance of the code
    - When the codebase has the correct design (e.g., *open-closed principle* is used), removing a feature toggle is a lot easier compared to situation where *shotgun surgery* needs to be applied to *spaghetti code*.
    - Comprehensive automated testing should make it relatively safe to remove feature toggles

3.16: Architectural Patterns

3.16.1: Event Sourcing Pattern

> ***Use event sourcing to capture state changes as a sequence of events.***

Event sourcing ensures that all changes to the state of a service are stored as an ordered sequence of events. Event sourcing makes it possible to query state changes. Also, the state change events act as an audit log. It is possible to reconstruct past states and rewind the current state to some earlier state. Unlike CRUD actions on resources, event sourcing is simpler and utilizes only CR (create and read) actions. It is only possible to create new events and read events. It is not possible to update an existing event or delete an event.

Let's have an example of using event sourcing to store orders in an e-commerce software system. The *order-service* should be able to store the following events:

- AbstractOrderEvent
    - Abstract base event for other concrete events containing timestamp and order id properties

- OrderCreatedEvent
  - Contains basic information about the order
- OrderPaymentEvent
  - Contains information about the order payment
- OrderModificationEvent
  - Contains information about modifications made by the customer to the order before packaging
- OrderPackagedEvent
  - Contains information about who collected and packaged the order
- OrderCanceledEvent
  - Describes that the customer has canceled the order and the order should not be shipped
- OrderShippedEvent
  - Contains information about the logistics partner and the tracking id of the order shipment
- OrderDeliveredEvent
  - Contains information about the pick-up point of the delivered order
- OrderShipmentReceivedEvent
  - Informs that the customer has received the shipment
- OrderReturnedEvent
  - Contains information about the returned order or order item(s)
- OrderReturnShippedEvent
  - Contains information about the logistics partner and the tracking id of the return shipment
- OrderReturnReceivedEvent
  - Contains information about who handled the order return and the status of returned items

- OrderReimbursedEvent
    - Contains information about the reimbursement for the returned order item(s) to the customer

3.16.2: Command Query Responsibility Segregation (CQRS) Pattern

Use the CQRS pattern if you want to use a different model for create/update (= command) operations compared to the model you want to use to query information.

Let's consider the previous *order-service* example that used event sourcing. In the *order-service*, all the commands are events. However, we want users to be able to query orders efficiently. We should have an additional representation of an order in addition to events because it is inefficient to always generate the current state of an order by replaying all the related events. For this reason, our architecture should utilize the CQRS pattern and divide the *order-service* into two different services: *order-command-service* and *order-query-service*.

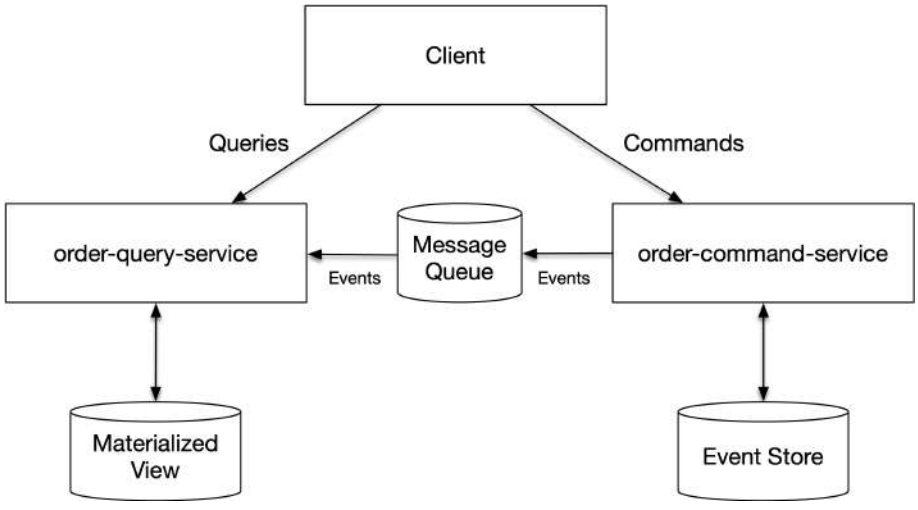

Figure 3.62. Order Services using CQRS

The *order-command-service* is the same as the original *order-service* that uses event sourcing, and the *order-query-service* is

a new service. The *order-query-service* has a database where it holds a materialized view of orders. The two services are connected with a message broker. The *order-command-service* sends events to a topic in the message broker. The *order-query-service* reads events from the topic and applies changes to the materialized view. The materialized view is optimized to contain basic information about each order, including its current state, to be consumed by the e-commerce company staff and customers. Because customers query orders, the materialized view should be indexed by the customer's `id` column to enable fast retrieval. Suppose that, in some particular case, a customer needs more details about an order available in the materialized view. In that case, the *order-command-service* can be used to query the events of the order for additional information.

3.16.3: Distributed Transaction Patterns

A distributed transaction is a transaction that spans multiple microservices. A distributed transaction consists of one or more remote requests. Distributed transactions can be implemented using the *saga pattern*. In the saga pattern, each request in a distributed transaction should have a respective compensating action defined. If one request in the distributed transaction fails, compensating requests should be executed for the already executed requests. The idea of executing the compensating requests is to bring the system back to the state where it was before the distributed transaction was started. So, the rollback of a distributed transaction is done via executing the compensating actions.

A failed request in a distributed transaction must be conditionally compensated if we cannot be sure whether the server successfully executed the request. This can happen when a request timeouts and we don't receive a response to indicate the request status.

Also, executing a compensating request can fail. For this reason, a microservice must persist compensating requests so they can be retried later until they all succeed. Persistence is needed because the microservice instance can be terminated

before it has completed all the compensating requests successfully. Another microservice instance can continue the work left by the terminated microservice instance.

Some requests in a distributed transaction can be such that they cannot be compensated. One typical example is sending an email. You can't get it unsent once it has been sent. There are at least two approaches to dealing with requests that cannot be compensated. The first one is to delay the execution of the request so that it can be made compensable. For example, instead of immediately sending an email, the email-sending microservice can store the email in a queue for later sending. The email-sending microservice can now accept a compensating request to remove the email from the sending queue.

Another approach is to execute non-compensable requests in the latest possible phase of the distributed transaction. You can, for example, issue the email-sending request as the last request of the distributed transaction. Then, the likelihood of needing to compensate for the email sending is lower than if the email was sent as the first request in the distributed transaction. You can also combine these two approaches. Sometimes, a request can be compensable even if you first think it is not. You should think creatively. Sending an email could be compensated by sending another email where you state that the email you sent earlier should be disregarded (for a specific reason).

3.16.3.1: Saga Orchestration Pattern

Orchestrator or controller microservice orchestrates the execution of a distributed transaction.

Let's have an example of a distributed transaction using the saga orchestration pattern with an online banking system where users can transfer money from their accounts. We have a higher-level microservice called *account-money-transfer-service*, which is used to make money transfers. The banking system also has two lower-level microservices called *account-*

balance-service and *account-transaction-service*. The *account-balance-service* holds accounts' balance information while the *account-transaction-service* keeps track of all account transactions. The *account-money-transfer-service* acts as a saga orchestrator and utilizes both of the lower-level microservices to make a money transfer to happen.

Let's consider a distributed transaction executed by the *account-money-transfer-service* when a user makes a withdrawal of $25.10:

1. The *account-money-transfer-service* tries to withdraw the amount from the user's account by sending the following request to the *account-balance-service*:

Figure 3.63

```
POST /account-balance-service/accounts/123456789012/withdraw
HTTP/1.1
Content-Type: application/json

{
  "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34",
  "amountInCents": 2510
}
```

The `sagaUuid` is a universally unique identifier (UUID) generated by the saga orchestrator before the saga begins. If there are not enough funds to withdraw the given amount, the request fails with the HTTP status code 400 *Bad Request*. If the request is successfully executed, the *account-balance-service* should store the saga UUID to a database table temporarily. This table stores information about successful sagas and should be cleaned regularly by deleting old enough saga UUIDs.

2. The *account-money-transfer-service* will create a new account transaction for the user's account by sending the following request to the *account-transaction-service*:

Figure 3.64

```
POST /account-transaction-
service/accounts/123456789012/transactions HTTP/1.1
Content-Type: application/json

{
  "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34",
```

```
  // Additional transaction information here...
}
```

The above-described distributed transaction has two requests, each of which can fail. Let's consider the scenario where the first request to the *account-balance-service* fails. If the first request fails due to a request timeout, we don't know if the recipient microservice successfully processed the request. We don't know because we did not get the response and status code. For that reason, we need to perform a conditional compensating action by issuing the following compensating request:

Figure 3.65

```
POST /account-balance-service/accounts/123456789012/undo-
withdraw HTTP/1.1
Content-Type: application/json

{
  "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34",
  "amountInCents": 2510
}
```

The *account-balance-service* will perform the `undo-withdraw` action only if a withdrawal with the given UUID was made earlier and that withdrawal has not been undone yet. Upon successful undoing, the *account-balance-service* will delete the row for the given saga UUID from the database table where the saga UUID was earlier temporarily stored. Further `undo-withdraw` requests with the same saga UUID will be no-op actions, making the `undo-withdraw` action idempotent.

Next, let's consider the scenario where the first request succeeds and the second request fails due to timeout. Now we have to compensate for both requests. First, we compensate for the first request as described earlier. Then, we will compensate for the second request by deleting the account transaction identified with the `sagaUuid`:

Figure 3.66

```
DELETE /account-transaction-
service/accounts/123456789012/transactions?sagaUuid=e8ab60b5-\
3053-46e7-b8da-87b1f46edf34 HTTP/1.1
```

If a compensating request fails, it must be repeated until it succeeds. Notice that the above compensating requests are both idempotent, i.e., they can be executed multiple times with the same result. Idempotency is a requirement for a compensating request because a compensating request may fail after the compensation has already been performed. That compensation request failure will cause the compensating request to be attempted again. The distributed transaction manager in the *account-money-transfer-service* should ensure that a distributed transaction is successfully completed or roll-backed by the instances of the *account-money-transfer-service*. You should implement a single distributed transaction manager library per programming language or technology stack and use that in all microservices that need to orchestrate distributed transactions. Alternatively, use a 3rd party library.

Let's have another short example with the *ecommerce-service* presented earlier in this chapter. The order-placing endpoint of the *ecommerce-service* should make the following requests in a distributed transaction:

1. Ensure payment
2. Create an order
3. Remove the ordered sales items from the shopping cart
4. Mark the ordered sales items sold
5. Enqueue an order confirmation email for sending

The respective compensating requests are the following:

1. Reimburse the payment
2. Delete the order using the saga UUID
3. Add the ordered sales items back to the shopping cart. (The shopping cart service must ensure that a sales item can be added only once to a shopping cart)
4. Mark the ordered sales items for sale
5. Dequeue the order confirmation email

3.16.3.2: Saga Choreography Pattern

> ***Microservices perform a distributed transaction in a choreography where one microservice initiates the distributed transaction, and the last microservice involved completes the distributed transaction by sending a completion message to the microservice that started the transaction.***

The saga choreography pattern utilizes asynchronous communication between microservices. Involved microservices send messages to each other in a choreography to achieve saga completion.

The saga choreography pattern has a couple of drawbacks:

- The execution of a distributed transaction is not centralized like in the saga orchestration pattern, and it can be hard to figure out how a distributed transaction is actually performed.
- It creates coupling between microservices, while microservices should be as loosely coupled as possible.

The saga choreography pattern works best in cases where the number of participating microservices is low. Then the coupling between services is low, and it is easier to reason how a distributed transaction is performed.

Let's have the same money transfer example as earlier, but now using the saga choreography pattern instead of the saga orchestration pattern.

1. The *account-money-transfer-service* initiates the saga by sending the following event to the message broker's *account-balance-service* topic:

Figure 3.67

```
{
  "event": "Withdraw",
  "data": {
    "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34",
    "amountInCents": 2510
```

```
  }
}
```

2. The *account-balance-service* will consume the `Withdraw` event from the message broker, perform a withdrawal, and if successful, send the same event to the message broker's *account-transaction-service* topic.
3. The *account-transaction-service* will consume the `Withdraw` event from the message broker, persist an account transaction, and if successful, send the following event to the message broker's *account-money-transfer-service* topic:

Figure 3.68

```
{
  "event": "Withdraw",
  "status": "Complete"
  "data": {
    "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34"
  }
}
```

If either step 2) or 3) fails, the *account-balance-service* or *account-transaction-service* will send the following event to message broker's *account-money-transfer-service* topic:

Figure 3.69

```
{
  "event": "Withdraw",
  "status": "Failure"
  "data": {
    "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34"
  }
}
```

If the *account-money-transfer-service* receives a `Withdraw` event with `Failure` status or does not receive a `Withdraw` event with `Complete` status during some timeout period, the *account-money-transfer-service* will initiate a distributed transaction rollback sequence by sending the following event to the message broker's *account-balance-service* topic:

Figure 3.70

```
{
  "event": "WithdrawRollback",
  "data": {
    "sagaUuid": "e8ab60b5-3053-46e7-b8da-87b1f46edf34",
    "amountInCents": 2510,
```

```
    // Additional transaction information here...
  }
}
```

Once the rollback in the *account-balance-service* is done, the rollback event will be produced to the *account-transaction-service* topic in the message broker. After the *account-transaction-service* successfully performs the rollback, it sends a `WithdrawRollback` event with a `Complete` status to the *account-money-transfer-service* topic. The withdrawal event is successfully rolled back once the *account-money-transfer-service* consumes that message. Suppose the *account-money-transfer-service* does not receive the `WithdrawRollback` event with a `Complete` status during some timeout period. In that case, it will restart the rollback choreography by resending the `WithdrawRollback` event to the *account-balance-service*.

3.17: Preferred Technology Stacks Principle

> ***Define preferred technology stacks for different purposes.***

The microservice architecture enables the use of the most suitable technology stack to develop each microservice. For example, some microservices require high performance and controlled memory allocation, and other microservices don't need such things. You can choose the used technology stack based on the needs of a microservice. For a real-time data processing microservice, you might pick C++ or Rust, and for a simple REST API, you might choose Node.js and Express, Java and Spring Boot, or Python and Django.

Even if the microservice architecture allows different teams and developers to decide what programming languages and technologies to use when implementing a microservice, defining preferred technology stacks for different purposes is still a good practice. Otherwise, you might find yourself in a situation where numerous programming languages and technologies are used in a software system. Some

programming languages and technologies like Clojure, Scala, or Haskell can be relatively niche. When software developers in the organization come and go, you might end up in situations where you don't have anyone who knows about some specific niche programming language or technology. In the worst case, a microservice needs to be reimplemented from scratch using some more mainstream technologies. For this reason, you should specify technology stacks that teams should use. These technology stacks should mainly contain mainstream programming languages and technologies.

For example, an architecture team might decide the following:

- Web clients should be developed using TypeScript, React, and Redux
- Non-API backend services should be developed by default in Go or C++ for performance reasons
- APIs should be developed with TypeScript, Node.js, and Nest.js or with Java and Spring Boot
- Integration tests should be implemented with Cucumber using the same language as is used for the implementation or, alternatively, with Python and Behave
- E2E tests should be implemented with Python and Behave
- Scripts should be implemented using Bash for small scripts and Python for larger scripts

The above technology stacks are pretty mainstream. Recruiting talent with needed knowledge and competencies should be effortless.

After you have defined the preferred technology stacks, you should create a utility or utilities that can be used to kick-start a new project using a particular technology stack quickly. This utility or utilities should generate the initial source code repository content for a new microservice, client, or library. The initial source code repository should contain at least the following items for a new microservice:

- Source code folder
- Unit test folder (if separate from source code folder)
- Integration test folder
- Build tools, like Gradle Wrapper for Java, for example
- Initial build definition file(s), like build.gradle for Java, CMakeLists.txt for C++ or package.json for Node.js
    - Initial dependencies defined in the build definition file
- .env file(s) to store environment variables for different environments (dev, CI)
- .gitignore
- Markdown documentation template file(s) (README.MD)
- Linting rules (e.g., .eslintrc.json)
- Code formatting rules (e.g., .prettier.rc)
- Initial code for integration tests, e.g., docker-compose.yml file for spinning up an integration testing environment
- Infrastructure code for the chosen cloud provider, e.g., code to deploy a managed SQL database in the cloud
- Code (e.g., Dockerfile) for building the microservice container image
- Deployment code (e.g., a Helm chart)
- CI/CD pipeline definition code

The utility should ask the following questions from the developer before creating the initial source code repository content for new microservice:

- What is the name of the microservice?
- To what cloud environment will microservice be deployed? (AWS, Azure, Google Cloud, etc.)

- What are the inter-service communication methods used? Based on the answer, the utility can add dependencies, e.g., a Kafka client library dependency
- Should microservice have a database, and what database?
- What are the other dependent microservices?

Of course, decisions about the preferred technology stacks are not engraved in stone. They are not static. As time passes, new technologies arise, and new programming languages gain popularity. At some point, a decision could be made that a new technology stack should replace an existing preferred technology stack. Then, new projects should use the new stack and old software components will be gradually migrated to use the new technology stack or eventually retired.

Many developers are keen on learning new things on a regular basis. They should be encouraged to work on hobby projects with technologies of their choice, and they should be able to utilize new programming languages and frameworks in selected new projects.

4: Object-Oriented Design Principles

This chapter describes principles related to object-oriented design. The following principles are discussed:

- Object-oriented programming concepts
- Programming paradigms
- Why is object-oriented programming hard?
- SOLID principles
- Clean microservice design (architecture) principle
- Vertical slice design (architecture) principle
- Class organization principle
- Uniform naming principle
- Encapsulation principle
- Prefer composition over inheritance principle
- Tactical domain-driven design principle
- Use the design patterns principle
- Don’t ask, tell principle
- Law of Demeter
- Avoid primitive type obsession principle
- Dependency injection principle
- Avoid duplication principle

We start the chapter by defining object-oriented programming (OOP) concepts and discussing different programming paradigms: object-oriented, imperative, and functional. We also analyze why OOP can be hard to master even though the concepts and fundamental principles are not so difficult to grasp.

In the coming code samples, I don't use the `Final` type hint for local variables, even though most of the local variables I create are constants. I made this decision to improve the readability of the examples. For production code, I recommend you add the `Final` type hints for local variables that should not be modified. By default, local variables should not be modified after the initial assignment. You should usually introduce a new variable instead of modifying an existing one. This rule applies, of course, to the function parameter variables, too. You shouldn't modify them in the function body.

Figure 4.1

```
from typing import Final


# Mutable variable
value = get_value()
value = get_another_value()

# Constant
value2: Final = get_value()

# Type checker reports an error
value2 = get_another_value()
```

For all you language designers, variables should be constant by default, and if you want a mutable variable, that should be denoted by a keyword like `Mutable`. This is how Python should look like:

Figure 4.2

```
from typing import Mutable


# Constant
value = get_value()

# Type checker reports an error
value = get_another_value()

# Mutable
value2: Mutable = get_value()
value2 = get_another_value()
```

4.1: Object-Oriented Programming Concepts

The following are the basic concepts related to OOP:

- Classes/Objects

    - Attributes and methods
    - Composition (= when attributes are other classes)
- Encapsulation
- Abstraction
- Inheritance
- Interfaces
    - Interface evolution
- Polymorphism
    - Dynamic dispatch (late binding)

Let's discuss each of the concepts next.

4.1.1: Classes/Objects

A class is a user-defined data type that acts as the blueprint for individual objects (instances of the class). An object is created using the class's `__init__` method, which sets the object's initial state. A class consists of *attributes* and *methods*, which can be either *class* or *instance* attributes/methods. Instance attributes define the state of an object. Instance methods act on instance attributes, i.e., they are used to query and modify the state of an object. Class attributes belong to the class, and class methods act on class attributes.

An object can represent either a concrete or abstract entity in the real world. For example, a circle and an employee object represent real-world entities, while an object representing an open file (a file handle) is an abstract entity. Objects can also be hybrid, representing something concrete and abstract.

Attributes of an object can contain other objects to create object hierarchies. This is called *object composition*, handled in more detail in the *prefer composition over inheritance principle* section.

In pure object-oriented languages like Java, you must always create a class where you can put functions. Even if you have only class methods and no attributes, you must create a class

in Java to host the class methods (static methods). In Python, you don't have to create classes for hosting functions; just put the functions into a single module or create a package (directory) and put each function in a separate module. Putting functions into classes has many benefits (e.g., dependency injection), which is why putting functions into classes is often a good idea.

4.1.2: Encapsulation

Encapsulation makes changing the internal state of an object directly outside of the object impossible. The idea of encapsulation is that the object's state is internal to the object and can be changed externally only by the object's public methods. Encapsulation contributes to better security and avoidance of data corruption. Unfortunately, the Python language does not support encapsulation, but some conventions can be used to simulate encapsulation. More about that in the *encapsulation principle* section.

4.1.3: Abstraction

Objects only reveal relevant internal mechanisms to other objects, hiding any unnecessary implementation code. Callers of the object methods don't need to know the object's internal workings. They adhere only to the public API of the object. This makes it possible to change the implementation details without affecting any external code.

4.1.4: Inheritance

Inheritance allows classes to be arranged in a hierarchy representing *is-a* relationships. For example, the `Employee` class might inherit from the `Person` class because an employee is also a person. All the attributes and methods in the parent (super) class also appear in the child (sub) class with the same names. For example, class `Person` might define attributes `name` and `birth_date`. These will also be available in the `Employee` class. Child class can add methods and attributes. Child class can also override a method in the parent class. For example,

the `Employee` might add attributes `employer` and `salary`. This technique allows easy re-use of the same functionality and data definitions, mirroring real-world relationships intuitively.

Python also supports multiple inheritance, where a child class can have multiple parent classes. The problem with multiple inheritance is that the child class can inherit different versions of a method with the same name. By default, multiple inheritance should be avoided whenever possible. Some languages, like Java, don't support multiple inheritance at all. In Python, inheriting from multiple so-called *mixin* classes can also be problematic because two mixin classes can also have clashing method names. Inheritance will cram additional functionality into a child class, making the class large and possibly not having a single responsibility. A better way to add functionality to a class is to compose the class of multiple other classes (the mixins). In that way, there is no need to worry about possible clashing of method names.

Multiple inheritance is always allowed for interfaces. Because Python does not have interfaces, you can use multiple inheritance with an abstract base class (ABC) or protocol serving as an interface. More about interfaces in the next section.

> Python 3.12 introduced the `@override` decorator that can be used for methods that are supposed to override a base class method. A similar kind of keyword or annotation is available in many other languages. Because the examples in this book are written for Python 3.11, I am not using the `@override` decorator, but you should take it into use and specify it for each method that is supposed to override a base class (or protocol) method. Using the decorator will reveal cases where you accidentally misspell a method name so that it does not override a base class method.

4.1.5: Interface

An interface specifies a contract that classes that implement the interface must obey. Interfaces are used to implement polymorphic behavior, which will be described in the next section. An interface consists of one or more methods that classes must implement. Python does not have interfaces but

has *abstract base classes* (ABCs) and *protocols*. Both of these can be used to create an interface. The ABC syntax is more verbose than protocol syntax because you must always denote a method in an ABC with the `@abstractmethod` decorator. You cannot instantiate an interface. It is just a contract specification.

Below are two interfaces implemented inheriting from the `ABC` and one class that implements both interfaces:

Figure 4.3

```
from abc import ABC, abstractmethod


class Drawable(ABC):
    @abstractmethod
    def draw(self) -> None:
        pass


class Clickable(ABC):
    @abstractmethod
    def click(self) -> None:
        pass


class Button(Drawable, Clickable):
    def draw(self) -> None:
        print("Button drawn")

    def click(self) -> None:
        print("Button clicked")


button = Button()
button.draw()
button.click()

# Output:
# Button drawn
# Button clicked
```

You can also combine the usage of ABCs and protocols. However, sticking to one way of defining interfaces is good practice. Below is an example where the `Window` class implements two interfaces, one that is defined extending from the `ABC` (in the above code listing) and one that is defined as extending from the `Protocol`:

Figure 4.4

```
from typing import Protocol
```

```
class Draggable(Protocol):
    def drag_to(self, x: int, y: int) -> None:
        pass


class Window(Drawable, Draggable):
    def draw(self) -> None:
        print("Window drawn")

    def drag_to(self, x: int, y: int) -> None:
        print(f"Window dragged to ({x}, {y})")


window = Window()
window.draw()
window.drag_to(200, 300)

# Output:
# Window drawn
# Window dragged to (200, 300)
```

> For the rest of the book, I will interchangeably use the terms *interface* and *protocol*.

4.1.5.1: Interface evolution

After an interface has been defined and is used by the implementing classes, and you would like to add method(s) to the interface, you might have to provide a default implementation in your interface because the classes that currently implement your interface don't implement the methods you are about to add to the interface. This is true in cases where the implementing classes are something you cannot or don't want to modify.

Let's imagine you have a `Message` interface with `get_data` and `get_length_in_bytes` methods, and you have classes implementing the `Message` interface, but you cannot modify the classes. You want to add `set_queued_at_instant` and `get_queued_at_instant` methods to the interface. You can add the methods to the interface but must provide a default implementation, like raising an error indicating the method is not implemented.

Figure 4.5

```
from typing import Protocol
```

```
class Message(Protocol):
    def get_data(self) -> bytearray:
        # ...

    def get_length_in_bytes(self) -> int:
        # ...

    def set_queued_at_instant(self, timestamp_in_ms: int) ->
None:
        raise NotImplementedError()

    def get_queued_at_instant(self) -> int:
        raise NotImplementedError()
```

4.1.6: Polymorphism

Polymorphism means that methods are polymorphic when the actual method to be called is decided during the runtime. For this reason, polymorphism is also called *late binding* (to a particular method) or *dynamic dispatch*. Polymorphic behavior is easily implemented using an interface variable. You can assign any object that implements the interface to the interface variable. When you call a method on the interface variable, that actual method to be called is decided based on what type of object is currently assigned to the interface variable. Below is an example of polymorphic behavior:

Figure 4.6

```
drawable: Drawable = Button()
drawable.draw()

# Output:
# Button drawn

drawable = Window()
drawable.draw()

# Output:
# Window drawn
```

Polymorphic behavior is also exhibited when you have a variable of the parent class type and assign a child class object to the variable, like in the below example:

Figure 4.7

```
class IconButton(Button):
    def draw(self) -> None:
        print("Button with icon drawn")
```

```
button: Button = Button()
button.draw()

# Output:
# Button drawn

button = IconButton()
button.draw()

# Output:
# Button with icon drawn
```

4.2: Programming Paradigms

The most popular programming languages, including Python, are multi-paradigm programming languages. Multi-paradigm languages support the following programming paradigms:

- Imperative programming
- Object-oriented programming
- Functional programming

4.2.1: Imperative Programming

Imperative programming is a programming paradigm that focuses on providing a sequence of explicit instructions or statements for the computer to follow to solve a problem or achieve a desired outcome. The program consists of a series of commands that modify the program state, typically using mutable variables and assignments. Imperative programming emphasizes how to achieve a result step by step, specifying the control flow and state changes explicitly. Typical imperative programming constructs are variable assignments, state mutations, match-case statements, if-statements, and different kinds of loops (for, while). Below is a code sample using imperative programming:

Figure 4.8

```
from typing import Final

numbers: Final = [1, 2, 3, 4, 5]

doubled_even_numbers: Final = []

for number in numbers:
```

```
    if number % 2 == 0:
        doubled_even_numbers.append(number**2)

print(doubled_even_numbers)

# Output:
# [4, 16]
```

In the above example, although the `doubled_even_numbers` variable is declared as `Final`, it is still a mutable list and we mutate the list inside the `for` loop.

4.2.2: Functional Programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. It emphasizes the use of immutable data and the composition of functions to solve problems. In functional programming, functions are first-class citizens, meaning they can be assigned to variables, passed as arguments to other functions, and returned as results. This enables the creation of higher-order functions and promotes modularity, code re-usability, and concise expression of complex operations in a declarative way. Functional programming avoids side effects, favoring pure functions that consistently produce the same output for a given input without causing side effects, making programs easier to reason about and test.

Unlike in imperative programming, in functional programming, you don't tell the computer *how* to do something but declare *what* you want, e.g., I want to filter even numbers from a sequence and calculate their squares.

> In mathematics and computer science, a higher-order function (HOF) is a function that does at least one of the following: 1. Takes one or more functions as arguments 2. Returns a function as its result.

Below is functional code that uses Python's list comprehension:

Figure 4.9

```
from typing import Final

numbers: Final = [1, 2, 3, 4, 5]

print([number**2 for number in numbers if number %2 == 0])

# Output:
# [4, 16]
```

As you can see, the above code is much safer, shorter, and more straightforward than the earlier imperative code. There are no variable assignments or state modifications.

Let's implement the above code using `map` and `filter` functions:

Figure 4.10

```
from typing import Final

numbers: Final = [1, 2, 3, 4, 5]

is_even: Final = lambda number: number % 2 == 0
doubled: Final = lambda number: number**2

print(list(map(doubled, filter(is_even, numbers))))

# Output:
# [4, 16]
```

In the above example, we assigned a *lambda* (a single statement anonymous function) to a variable. This practice is not according to PEP 8. We should use `def` to define a function instead:

Figure 4.11

```
from typing import Final

numbers: Final = [1, 2, 3, 4, 5]

def is_even(number: int):
    return number % 2 == 0

def doubled(number: int):
    return number**2

print(list(map(doubled, filter(is_even, numbers))))

# Output:
# [4, 16]
```

Both the `is_even` and `doubled` are pure functions because they return the same output for the same input without any side

effects. The above expression could be easier to read. Let's use a variable (= constant) to store an intermediate value:

Figure 4.12

```
even_numbers: Final = filter(is_even, numbers)
print(list(map(doubled, even_numbers)))

# Output:
# [4, 16]
```

There is another way to implement the above code using a composition of functions. We can define reusable functions and compose more specific ones from general-purpose ones. Below is an example of function composition using the `compose` function from the toolz library. The example also uses the `partial` function from the `functools` module to create partially applied functions. For example, the `filter_even` function is a partially applied `filter` function where the first parameter is bound to the `is_even` function. Similarly, the `map_doubled` function is a partially applied `map` function where the first parameter is bound to the `doubled` function. The `compose` function composes two or more functions in the following way: `compose(f, g)(x)` is the same as `f(g(x))` and `compose(f, g, h)(x)` is same as `f(g(h(x)))` and so on. You can compose as many functions as you need/want.

Figure 4.13

```
from functools import partial
from typing import Final

from toolz import compose

numbers: Final = [1, 2, 3, 4, 5]


def is_even(number: int):
    return number % 2 == 0


def doubled(number: int):
    return number**2


filter_even: Final = partial(filter, is_even)
map_doubled: Final = partial(map, doubled)
doubled_even: Final = compose(list, map_doubled, filter_even)
print(doubled_even(numbers))

# Output:
# [4, 16]
```

In the above example, all the following functions can be made re-usable and put into a library:

- `is_even`
- `doubled`
- `filter_even`
- `map_doubled`

Modern code should favor functional programming over imperative programming when possible. As compared to functional programming, imperative programming comes with the following disadvantages:

1. *Mutable state*: Imperative programming relies heavily on mutable state, where variables can be modified throughout the program's execution. This can lead to subtle bugs and make the program harder to reason about, as the state can change unpredictably. In functional programming, immutability is emphasized, reducing the complexity of state management and making programs more reliable.
2. *Side effects*: Imperative programming often involves side effects, where functions or operations modify the state or interact with the external world. Side effects make the code harder to test, reason about, and debug. On the other hand, functional programming encourages pure functions with no side effects, making the code more modular, reusable, and testable.
3. *Concurrency and parallelism*: Imperative programming can be challenging to parallelize and reason about in concurrent scenarios. Since mutable state can be modified by multiple threads or processes, race conditions and synchronization issues can occur. Functional programming, with its emphasis on immutability and pure functions, simplifies concurrency and parallelism by eliminating shared mutable state.
4. *Lack of referential transparency*: Imperative programming tends to rely on assignments and statements

that modify variables in place. This can lead to code that is difficult to reason about due to implicit dependencies and hidden interactions between different parts of the code. In functional programming, referential transparency is a key principle where expressions can be replaced with their values without changing the program's behavior. This property allows for easier understanding, debugging, and optimization.

Pure imperative programming also quickly leads to code duplication, lack of modularity, and abstraction issues. These are issues that can be solved using object-oriented programming.

4.3: Multi-Paradigm Programming Principle

> ***You should not use a single programming paradigm only.***

To best utilize both object-oriented and functional programming (FP) when developing software, you can leverage the strengths of each paradigm in different parts of your codebase. Use domain-driven design (DDD) and object-oriented design to design the application: interfaces and classes. Implement classes by encapsulating related behavior and (possibly mutable, but aim for immutable) state in the classes. Apply OOP principles like *SOLID principles* and *design patterns*. These principles and patterns make code modular and easily extensible without accidentally breaking existing code. Use FP as much as possible when implementing class and instance methods. Embrace functional composition by creating pure functions that take immutable data as input and always produce the same output for the same input without side effects. Use higher-order functions to compose functions and build complex operations from simpler ones. For example, utilize higher-order functions in OOP by passing functions as arguments to methods or using them as callbacks. This allows for greater flexibility and modularity, enabling

functional-style operations within an OOP framework. Also, remember to use functional programming libraries, either the standard or 3rd party libraries. Consider using functional techniques for error handling, such as *Either* or *Maybe/Optional* types. This helps you manage errors without exceptions, promoting more predictable and robust code. This is because function signatures don't tell if they can raise an error. You must remember to consult the documentation and check if a function can raise an error.

Aim for immutability within your codebase, regardless of the paradigm. Immutable data reduces complexity, avoids shared mutable state, and facilitates reasoning about your code. Favor creating new objects or data structures instead of modifying existing ones.

4.4: Why is Object-Oriented Programming Hard?

The basic concepts of OOP are not complex to understand, so why is it hard to master OOP? Below are listed things that can make OOP hard:

- You cannot rush into coding. You must have patience and perform object-oriented design (OOD) first
- You cannot get the OOD right on the first try. You need to have discipline and time reserved for refactoring.
- The difference between object composition and inheritance is not correctly understood, and inheritance is used in place of object composition, making the OOD flawed
- SOLID principles are not understood or followed
    - It can be challenging to create optimal-sized classes and functions with a single responsibility
        - For example, you might have a single-responsibility class, but the class is too big. You must realize that you must split the class into smaller classes the original class is composed

        of. Each of these smaller classes has a single responsibility on a lower level of abstraction compared to the original class
    - Understanding and following the open-closed principle can be challenging
        - The idea of the open-closed principle is to avoid modifying existing code and thus avoid breaking any existing working code. For example, If you have a collection class and need a thread-safe collection class, don't modify the existing one, e.g., by adding a constructor flag to tell if a collection should be thread-safe. Instead, create a totally new class for thread-safe collections.
    - Liskov's substitution principle is not as simple as it looks
        - For example, suppose you have a base class `Circle` with a `draw` method. If you derive a `FilledCircle` class from the `Circle` class, you must implement the `draw` function so that it first calls the base class method. In some cases, it is possible to override the base class method with the derived class method
    - Interface segregation is usually left undone if it is not immediately needed. This might hinder the extensibility of the codebase in the future
    - In many texts, the dependency inversion principle is explained in complicated terms. The dependency inversion principle generally means programming against interfaces instead of concrete class types.
- You don't understand the value of dependency injection and are not using it
    - Dependency injection is a requirement for effectively utilizing some other principles, like the open-closed principle

    - Dependency injection makes unit testing a breeze because you can create mock implementations and inject them into the tested code
- You don't know/understand design patterns and don't know when and how to use them
    - Familiarize yourself with the design patterns
    - Some design patterns are more useful than others. You use some patterns basically in every codebase, and some patterns you rarely use
    - Many design patterns help make code more modular and extensible and help avoid modifying existing code. Modifying existing code is always a risk. You can introduce bugs in already working code. These bugs are sometimes very subtle and hard to discover.
    - Learning the design patterns takes time. It can take years to master them, and mastery is only achieved by repeatedly using them in real-life codebases.

Mastering OOD and OOP is a life-long process. You are never 100% ready. The best way to become better in OOD and OOP, as in any other thing in your life, is practicing. I have been practicing OOD and OOP for 29 years and am still improving and learning something new regularly. Start a non-trivial (hobby/work) project and try to make the code 100% clean. Whenever you think you are ready with it, leave the project for some time and later come back to the project, and you might be surprised to notice that there are several things still needing improvement!

4.5: SOLID Principles

All five SOLID principles are covered in this section. The *dependency inversion principle* is generalized as a *program against interfaces principle*. The five SOLID principles are the following:

- Single responsibility principle

- Open-closed principle
- Liskov's substitution principle
- Interface segregation principle
- Dependency inversion principle (Generalization: program against interfaces principle)

4.5.1: Single Responsibility Principle

> ***Classes should have one responsibility: representing a thing or providing a single functionality. Functions should do one thing only.***

Single responsibility should be at a particular abstraction level. A class or function can become too large if the abstraction level is too high. Then, split the class or function into multiple classes or functions on a lower level of abstraction. The single responsibility principle is akin to the separation of concerns principle. In the *separation of concerns* principle, you divide a software component into distinct "sections". A section can be any size, e.g., a subdomain (package), module, class, or function.

Suppose you need to implement configuration reading and parsing for your software component. Reading and parsing are two different concerns and should be implemented in separate "sections", which in practice means implementing them in different class hierarchies: a `ConfigReader` interface with various implementation classes, like `FileSystemConfigReader`, `DatabaseConfigReader`, `RestApiConfigReader`, and a `ConfigParser` interface with various implementation classes like `JsonConfigParser`, `YamlConfigParser`, `TomlConfigParser`, `XmlConfigParser`, etc.

Another great example of the separation of concerns principle is the *clean microservice design (or architecture) principle* where you separate the microservice's business logic from the microservice's input and output. In this way, it is easy to make the microservice support various inputs and outputs without

modifying the business logic "section". The clean microservice design (or architecture) principle is discussed later in this chapter.

Let's get back to the single responsibility principle. Each class should have a single dedicated purpose. A class can represent a single thing, like a bank account (`Account` class) or an employee (`Employee` class), or provide a single functionality like parsing a configuration file (`ConfigFileParser` class) or calculating tax (`TaxCalculator` class).

We should not create a class representing a bank account and an employee. It is simply wrong. Of course, an employee can *have* a bank account. But that is a different thing. It is called object composition. In object composition, an `Employee` class object contains an `Account` class object. The `Employee` class still represents one thing: An employee (who can have a bank account). Object composition is covered in more detail later in this chapter.

At the function level, each function should perform a single task. The function name should describe what task the function performs, meaning each function name should contain a verb. The function name should not contain the word *and* because it can mean that the function is doing more than one thing or you haven't named the function on a correct abstraction level. You should not name a function according to the steps it performs (e.g., do_this_and_that_and_then_some_third_thing) but instead, use wording on a higher level of abstraction.

When a class represents something, it can contain multiple methods. For example, an `Account` class can have methods like `deposit` and `withdraw`. It is still a single responsibility if these methods are simple enough and if there are not too many methods in the class.

Below is a real-life code example where the *and* word is used in the function name:

Figure 4.14

```
def delete_page_and_all_references(page: Page):
    delete_page(page)
    registry.delete_reference(page.name)
    config_keys.delete_key(page.name.make_key())
```

In the above example, the function does two things: delete a page and remove all the references to that page. But if we look at the code inside the function, we can realize that it is also doing a third thing: deleting a page key from configuration keys. So should the function be named `delete_page_and_all_references_and_config_key`? It does not sound reasonable. The problem with the function name is that it is at the same level of abstraction as the function statements. The function name should be at a higher level of abstraction than the statements inside the function.

How should we then name the function? I cannot say for sure because I don't know the context of the function. We could name the function just `delete`. This would tell the function caller that a page will be deleted. The caller does not need to know all the actions related to deleting a page. The caller just wants a page to be deleted. The function implementation should fulfill that request and do the needed housekeeping tasks, like removing all the references to the page being deleted and so on.

Let's consider another example with React Hooks. React Hooks has a function named `useEffect`, which can be used to enqueue functions to be run after component rendering. The `useEffect` function can be used to run some code after the initial render (after the component mount), after every render, or conditionally. This is quite a responsibility for a single function. Also, the function's name does not reveal its purpose; it sounds abstract. The word *effect* comes from the fact that this function is used to enqueue other functions with side effects to be run. The term side effect might be familiar to functional programmers. It indicates that a function is not pure because it causes side effects.

Below is an example of a React functional component:

Figure 4.15

```
import { useEffect } from "react";

export default function MyComponent() {
  useEffect(() => {
    function startFetchData() {
      // ...
    }
```

```
    function subscribeToDataUpdates() {
      // ...
    }

    function unsubscribeFromDataUpdates() {
      // ...
    }

    startFetchData();
    subscribeToDataUpdates();
    return function cleanup() { unsubscribeFromDataUpdates() };
  }, []);

  // JSX to render
  return ...;
}
```

In the above example, the `useEffect` call makes calls to functions `startFetchData` and `subscribeToDataUpdates` to happen after the initial render because of the supplied empty array for dependencies (the second parameter to the `useEffect` function). The cleanup function returned from the function supplied to `useEffect` will be called before the effect will be rerun or when the component is unmounted and in this case, only on unmount because the effect will only run once after the initial render.

Let's imagine how we could improve the `useEffect` function. We could split the rather abstract-sounding `useEffect` method into multiple methods on a lower level of abstraction. The functionality related to mounting and unmounting could be separated into two different functions: `afterMount` and `beforeUnmount`. Then, we could change the above example to the following piece of code:

Figure 4.16

```
export default function MyComponent() {
  function startFetchData() {
    // ...
  }

  function subscribeToDataUpdate() {
    // ...
  }

  function unsubscribeFromDataUpdate() {
    // ...
  }

  afterMount(startFetchData, subscribeToDataUpdates);
  beforeUnmount(unsubscribeFromDataUpdates)
```

```
  // JSX to render
  return ...;
}
```

The above example is cleaner and much easier for a reader to understand than the original example. There are no multiple levels of nested functions. You don't have to return a function to be executed on component unmount, and you don't have to supply an array of dependencies.

Let's have another example of a React functional component:

Figure 4.17

```
import { useEffect, useState } from "react";

export default function ButtonClickCounter() {
  const [clickCount, setClickCount] = useState(0);

  useEffect(() => {
    function updateClickCountInDocumentTitle() {
      document.title = `Click count: ${clickCount}`;
    }

    updateClickCountInDocumentTitle();
  });
}
```

In the above example, the effect is called after every render (because no dependencies array is supplied for the `useEffect` function). Nothing in the above code clearly states what will be executed and when. We still use the same `useEffect` function, but now it behaves differently than in the previous example. It seems like the `useEffect` function is doing multiple things. How to solve this? Let's think hypothetically again. We could extract functionality from the `useEffect` function and introduce yet another new function called `afterEveryRender` that can be called when we want something to happen after every render:

Figure 4.18

```
export default function ButtonClickCounter() {
  const [clickCount, setClickCount] = useState(0);

  afterEveryRender(function updateClickCountInDocumentTitle() {
    document.title = `Click count: ${clickCount}`;
  });
}
```

The intentions of the above React functional component are pretty clear: It will update the click count in the document title after every render.

Let's optimize our example so that the click count update happens only if the click count has changed:

Figure 4.19

```
import { useEffect, useState } from "react";

export default function ButtonClickCounter() {
  const [clickCount, setClickCount] = useState(0);

  useEffect(() => {
    function updateClickCountInDocumentTitle() {
      document.title = `Click count: ${clickCount}`;
    }

    updateClickCountInDocumentTitle();
  }, [clickCount]);
}
```

Notice how `clickCount` is now added to the dependencies array of the `useEffect` function. This means the effect is not executed after every render but only when the click count is changed.

Let's imagine how we could improve the above example. We could once again extract functionality from the `useEffect` function and introduce a new function that handles dependencies: `afterEveryRenderIfChanged`. Our hypothetical example would now look like this:

Figure 4.20

```
export default function ButtonClickCounter() {
  const [clickCount, setClickCount] = useState(0);

  afterEveryRenderIfChanged(
    [clickCount],
    function updateClickCountInDocumentTitle() {
      document.title = `Click count: ${clickCount}`;
  });
}
```

Making functions do a single thing at an appropriate level of abstraction also helped make the code more readable. Regarding the original examples, a reader must look at the end of the `useEffect` function call to figure out in what circumstances the effect function will be called. Understanding

and remembering the difference between a missing and empty dependencies array is cognitively challenging.

> ***Good code is such that it does not make the code reader think. At best, the code should read like beautifully written prose.***

In the above example, we can read like prose: *after every render if changed click count, update click count in document title*.

One idea behind the single responsibility principle is that it enables software development using the *open-closed principle* described in the next section. When you follow the single responsibility principle and need to add functionality, you add it to a new class, which means you don't need to modify an existing class. You should avoid modifying existing code but extend it by adding new classes, each with a single responsibility. Modifying existing code always poses a risk of breaking something that works.

4.5.2: Open-Closed Principle

> ***Software code should be open for extension and closed for modification. Functionality in existing classes should not be modified, but new classes that implement a new or existing interface or extend an existing class should be introduced.***

Any time you find yourself modifying some method in an existing class, you should consider if this principle could be followed and if the modification could be avoided. Every time you modify an existing class, you can introduce a bug in the working code. The idea of this principle is to leave the working code untouched so it does not get accidentally broken.

Let's have an example where this principle is *not* followed. We have the following existing and working code:

Figure 4.21

```
from typing import Protocol


class Shape(Protocol):
    # ...


class RectangleShape(Shape):
    def __init__(self, width: int, height: int):
        self.__width = width
        self.__height = height

    @property
    def width(self) -> int:
       return self.__width

    @property
    def height(self) -> int:
       return self.__height

    @width.setter
    def width(self, width: int) -> None:
        self.__width = width

    @height.setter
    def height(self, height: int) -> None:
        self.__height = height
```

Suppose we get an assignment to introduce support for square shapes. Let's try to modify the existing `RectangleShape` class, because a square is also a rectangle:

Figure 4.22

```
class RectangleShape(Shape):
    # Constructor for creating rectangles
    def __init__(self, width: int, height: int):
        self.__width = width
        self.__height = height

    # Factory method for creating squares
    @classmethod
    def create_square(cls, side_length: int) ->
'RectangleShape':
        return cls(side_length, side_length)

    @property
    def width(self) -> int:
        return self.__width

    @property
    def height(self) -> int:
        return self.__height

    @width.setter
    def width(self, width: int) -> None:
        if self.__height == self.__width:
            self.__height = width
        self.__width = width
```

```
    @height.setter
    def height(self, height: int) -> None:
        if self.__height == self.__width:
            self.__width = height
        self.__height = height
```

We needed to add a factory method for creating squares and modify two methods in the class. Everything works okay when we run tests. But we have introduced a subtle bug in the code: If we create a rectangle with an equal height and width, the rectangle becomes a square, which is probably not what is wanted. This is a bug that can be hard to find in unit tests. This example showed that modifying an existing class can be problematic. We modified an existing class and accidentally broke it.

A better solution to introduce support for square shapes is to use the *open-closed principle* and create a new class that implements the `Shape` protocol. Then, we don't have to modify any existing class, and there is no risk of accidentally breaking something in the existing code. Below is the new `SquareShape` class:

Figure 4.23

```
class SquareShape(Shape):
    def __init__(self, side_length: int):
        self.__side_length = side_length

    @property
    def side_length(self) -> int:
        return self.__side_length

    @side_length.setter
    def side_length(self, side_length: int) -> None:
        self.__side_length = side_length
```

An existing class can be safely modified by adding a new method in the following cases:

1. The added method is a pure function, i.e., it always returns the same value for the same arguments and does not have side effects, e.g., it does not modify the object's state.
2. The added method is read-only and tread-safe, i.e., it does not modify the object's state and accesses the object's state in a thread-safe manner in the case of multithreaded

code. An example of a read-only method in the `Shape` class would be a method that calculates a shape's area.

3. Class is immutable, i.e., the added method (or any other method) cannot modify the object's state

There are a couple of cases where the modification of existing code is needed. One example is factories. When you introduce a new class, you need to modify the related factory to be able to create an instance of that new class. For example, if we had a `ShapeFactory` class, we would need to modify it to support the creation of `SquareShape` objects. Fortunately, this modification is simple: Just adding a new case branch. The probability of introducing a bug is very low. Factories are discussed later in this chapter.

Another case is adding a new enum constant. You typically need to modify existing code to handle the new enum constant. If you forget to add the handling of the new enum constant somewhere in the existing code, typically, a bug will arise. For this reason, You should always safeguard match-case statements with a _ case that raises an exception and safeguard if/elif structures with an else branch that raises an exception. You can also enable your static code analysis tool to report an issue if a match statement's _ case or an else branch is missing from an if/elif structure. Also, some static code analysis tools can report an issue if you miss handling an enum constant in a match-case statement.

Here is an example of safeguarding an if/elif structure:

Figure 4.24

```
from enum import Enum
from typing import Protocol, Final


class FilterType(Enum):
    INCLUDE = 1
    EXCLUDE = 2


class Filter(Protocol):
    def is_filtered_out(self) -> bool:
        pass


class FilterImpl(Filter):
    def __init__(self, filter_type: FilterType):
```

```
        self.__filter_type: Final = filter_type

    def is_filtered_out(self) -> bool:
        if self.__filter_type == FilterType.INCLUDE:
            # ...
        elif self.__filter_type == FilterType.EXCLUDE:
            # ...
        else:
            # Safeguarding
            raise ValueError('Invalid filter type')
```

Safeguarding might be needed for a literal type union also:

Figure 4.25

```
from typing import Literal

FilterType = Literal['include', 'exclude']

filter_type: FilterType = # ...

if filter_type == 'include':
    # ...
elif filter_type == 'exclude':
    # ...
else:
    # Safeguarding
    raise ValueError('Invalid filter type')
```

In the future, if a new literal is added to the `FilterType` type and you forget to update the if-statement, you get an exception raised instead of silently passing through the if-statement without any action.

We can notice from the above examples that if/elif structures could be avoided with a better object-oriented design. For instance, we could create a `Filter` protocol and two separate classes, `IncludeFilter` and `ExcludeFilter`. The classes implement the `Filter` protocol. Using object-oriented design allows us to eliminate the `FilterType` enum and the if/elif structure. This is known as the *replace conditionals with polymorphism* refactoring technique. Refactoring is discussed more in the next chapter. Below is the above example refactored to be more object-oriented:

Figure 4.26

```
from typing import Protocol


class Filter(Protocol):
    def is_filtered_out(self) -> bool:
        pass
```

```
class IncludeFilter(Filter):
    # ...

    def is_filtered_out(self) -> bool:
        # ...


class ExcludeFilter(Filter):
    # ...

    def is_filtered_out(self) -> bool:
        # ...
```

4.5.3: Liskov's Substitution Principle

> ***Objects of a superclass should be replaceable with objects of its subclasses without breaking the application. I.e., objects of subclasses behave the same way as the objects of the superclass.***

Following *Liskov's substitution principle* guarantees semantic interoperability of types in a type hierarchy.

Let's have an example with a `RectangleShape` class and a derived `SquareShape` class:

Figure 4.27

```
from typing import Protocol


class Shape(Protocol):
    def draw(self) -> None:
        pass


class RectangleShape(Shape):
    def __init__(self, width: int, height: int):
        self.__width = width;
        self.__height = height

    def draw(self):
        # ...

    @property
    def width(self) -> int:
        return self.__width

    @property
    def height(self) -> int:
        return self.__height

    @width.setter
    def width(self, width: int) -> None:
```

```
            self.__width = width

    @height.setter
    def height(self, height: int) -> None:
            self.__height = height


class SquareShape(RectangleShape):
    def __init__(self, side_length: int):
        super().__init__(side_length, side_length)

    @RectangleShape.width.setter
    def width(self, width: int) -> None:
        RectangleShape.width.fset(self, width)
        RectangleShape.height.fset(self, width)

    @RectangleShape.height.setter
    def height(self, height: int) -> None:
        RectangleShape.width.fset(self, height)
        RectangleShape.height.fset(self, height)
```

The above example does not follow Liskov's substitution principle because you cannot set a square's width and height separately. This means that a square is not a rectangle from an object-oriented point of view. Of course, mathematically, a square is a rectangle. But when considering the above public API of the `RectangleShape` class, we can conclude that a square is not a rectangle because a square cannot fully implement the API of the `RectangleShape` class. We cannot substitute a square object for a rectangle object. What we need to do is to implement the `SquareShape` class without deriving from the `RectangleShape` class:

Figure 4.28

```
class SquareShape(Shape):
    def __init__(self, side_length: int):
        self.__side_length = side_length

    def draw(self):
        # ...

    @property
    def side_length(self) -> int:
        return self.__side_length

    @side_length.setter
    def side_length(self, side_length: int) -> None:
        self.__side_length = side_length
```

Liskov's substitution principle requires the following:

- A subclass must implement the superclass API and retain (or, in some cases, replace) the functionality of the superclass.
- A superclass should not have protected attributes because it allows subclasses to modify the state of the superclass, which can lead to incorrect behavior in the superclass.

Below is an example where a subclass extends the behavior of a superclass in the `do_something` method. The functionality of the superclass is retained in the subclass making a subclass object substitutable for a superclass object.

Figure 4.29

```
class SuperClass:
    # ...

    def do_something(self):
        # ...


class SubClass(SuperClass):
    # ...

    def do_something(self):
        super().do_something()

        # Some additional behaviour...
```

Let's have a concrete example of using the above strategy. We have the following `CircleShape` class defined:

Figure 4.30

```
from typing import Protocol


class Shape(Protocol):
    def draw(self) -> None:
        pass


class CircleShape(Shape):
    def draw(self) -> None:
        # Draw the circle stroke here
```

Next, we introduce a class for filled circles:

Figure 4.31

```
class FilledCircleShape(CircleShape):
    def draw(self) -> None:
        super().draw() # Draws the circle stroke
        # Fill the circle here...
```

The `FilledCircleShape` class fulfills the requirements of Liskov's substitution principle. We can use an instance of the `FilledCircleShape` class everywhere where an instance of the `CircleShape` class is wanted. The `FilledCircleShape` class does all that the `CircleShape` class does, plus adds some behavior (= fills the circle).

You can also completely replace the superclass functionality in a subclass:

Figure 4.32

```
from collections import UserList


class ReverseList(UserList):
    def __iter__(self):
        return ReverseListIterator(self)
```

The above subclass implements the superclass API and retains its behavior: The `iterator` method still returns an iterator. It just returns a different iterator compared to the superclass.

4.5.4: Interface Segregation and Multiple Inheritance Principle

> ***Segregate a larger interface to microinterfaces with a single capability/behavior and construct larger interfaces by inheriting multiple microinterfaces.***

We will use the Python-specific term *protocol* instead of *interface* for the rest of this section. Let's have an example with several automobile classes:

Figure 4.33

```
from typing import Protocol

from Location import Location


class Automobile(Protocol):
    def drive(self, start: Location, destination: Location) ->
None:
        pass

    def carry_cargo(
        self,
        volume_in_cubic_meters: float,
```

```
        weight_in_kgs: float
    ) -> None:
        pass


class PassengerCar(Automobile):
    # Implement drive and carry_cargo


class Van(Automobile):
    # Implement drive and carry_cargo


class Truck(Automobile):
    # Implement drive and carry_cargo


class ExcavatingAutomobile(Automobile):
    def excavate(self) -> None:
        pass


class Excavator(ExcavatingAutomobile):
    # Implement drive, carry_cargo and excavate
```

Notice how the `Automobile` protocol has two methods declared. This can limit our software if we later want to introduce other vehicles that could be just driven but unable to carry cargo. We should segregate two microprotocols from the `Automobile` protocol in an early phase. A microprotocol defines a single capability or behavior. After segregation, we will have the following two microprotocols:

Figure 4.34

```
from typing import Protocol

from Location import Location


class Drivable(Protocol):
    def drive(self, start: Location, destination: Location) ->
None:
        pass


class CargoCarrying(Protocol):
    def carry_cargo(
        self,
        volume_in_cubic_meters: float,
        weight_in_kgs: float
    ) -> None:
        pass
```

Now that we have two protocols, we can use these protocols separately in our codebase. For example, we can have a list of

drivable objects or a list of objects that can carry cargo. We still want to have a protocol for automobiles, though. We can use *protocol multiple inheritance* to redefine the `Automobile` protocol to extend the two microprotocols:

Figure 4.35

```
class Automobile(Drivable, CargoCarrying):
    pass
```

If we look at the `ExcavatingAutomobile` protocol, we can notice that it extends the `Automobile` protocol and adds excavating behavior. Once again, we have a problem if we want an excavating machine that is not auto-mobile. The excavating behavior should be segregated into its own microprotocol:

Figure 4.36

```
class Excavating(Protocol):
    def excavate(self) -> None:
        pass
```

We can once again use the protocol multiple inheritance to redefine the `ExcavatingAutomobile` protocol as follows:

Figure 4.37

```
class ExcavatingAutomobile(Excavating, Automobile):
    pass
```

The `ExcavatingAutomobile` protocol now extends three microprotocols: `Excavating`, `Drivable`, and `CargoCarrying`. Where-ever you need an excavating, drivable, or cargo-carrying object in your codebase, you can use an instance of the `Excavator` class there.

Let's have another example with a generic collection protocol. We should be able to traverse a collection and also be able to compare two collections for equality. First, we define a generic `Iterator` protocol for iterators. It has two methods, as described below:

Figure 4.38

```
from typing import Protocol, TypeVar

T = TypeVar('T')


class Iterator(Protocol[T]):
```

```
    def has_next_elem(self) -> bool:
        pass

    def get_next_elem(self) -> T:
        pass
```

Next, we can define the collection protocol:

Figure 4.39

```
class Collection(Protocol[T]):
    def create_iterator(self) -> Iterator[T]:
        pass

    def equals(self, another_collection: 'Collection[T]') ->
bool:
        pass
```

`Collection` is a protocol with two unrelated methods. Let's segregate those methods into two microprotocols: `Iterable` and `Equatable`. The `Iterable` protocol is for objects that you can iterate over. It has one method for creating new iterators. The `Equatable` protocol's `equals` method is more generic than the `equals` method in the above `Collection` protocol. You can equate an `Equatable[T]` object with another object of type `T`:

Figure 4.40

```
class Iterable(Protocol[T]):
    def create_iterator(self) -> Iterator[T]:
        pass


class Equatable(Protocol[T]):
    def equals(self, another_object: T) -> bool:
        pass
```

We can use protocol multiple inheritance to redefine the `Collection` protocol as follows:

Figure 4.41

```
class Collection(Iterable[T], Equatable['Collection[T]']):
    pass
```

We can implement the `equals` method by iterating elements in two collections and checking if the elements are equal:

Figure 4.42

```
from abc import abstractmethod


class AbstractCollection(Collection[T]):
    @abstractmethod
```

```
    def create_iterator(self) -> Iterator[T]:
        pass

    def equals(self, another_collection: Collection[T]):
        iterator = self.create_iterator()
        another_iterator = another_collection.create_iterator()

        collections_are_equal = self.__are_equal(
            iterator, another_iterator
        )

        return (
            False
            if another_iterator.has_next_elem()
            else collections_are_equal
        )

    @staticmethod
    def __are_equal(iterator: Iterator[T], another_iterator:
Iterator[T]):
        while iterator.has_next_elem():
            if another_iterator.has_next_elem():
                if (
                    iterator.get_next_elem()
                    != another_iterator.get_next_elem()
                ):
                    return False
            else:
                return False
        return True
```

Collections can also be compared. Let's introduce support for such collections. First, we define a generic `Comparable` protocol for comparing an object with another object:

Figure 4.43

```
from typing import Protocol, Literal

ComparisonResult = Literal['isLessThan', 'areEqual',
'isGreaterThan', 'unspecified']


class Comparable(Protocol[T]):
    def compare(self, another_object: T) -> ComparisonResult:
        pass
```

Now, we can introduce a comparable collection protocol that allows comparing two collections of the same type:

Figure 4.44

```
class ComparableCollection(Comparable[Collection[T]],
Collection[T]):
    pass
```

Let's define a generic sorting algorithm for collections whose elements are comparable:

Figure 4.45

```
U = TypeVar('U', bound=ComparableCollection)

def sort(collection: U) -> U:
    # ...
```

Let's create two protocols, `Inserting` and `InsertingIterable` for classes whose instances elements can be inserted into:

Figure 4.46

```
class Inserting(Protocol[T]):
    def insert(self, element: T) -> None:
        pass


class InsertingIterable(Inserting[T], Iterable[T]):
    pass
```

Let's redefine the `Collection` protocol to extend the `InsertingIterable` protocol because a collection is iterable, and you can insert elements into a collection.

Figure 4.47

```
class Collection(InsertingIterable[T]):
    pass
```

Next, we introduce two generic algorithms for collections: `map` and `filter`. We can realize that those algorithms work with more abstract objects than collections. We benefit from protocol segregation because instead of the `Collection` protocol, we can use the `Iterable` and `InsertingIterable` protocols to create generic `map` and `filter` algorithms. Later, it is possible to introduce additional non-collection iterable objects that can utilize the algorithms. Below are the implementations of the `map` and `filter` functions:

Figure 4.48

```
from collections.abc import Callable
from typing import TypeVar

T = TypeVar('T')
U = TypeVar('U')


def map(
```

```
    source: Iterable[T],
    mapped: Callable[[T], U],
    destination: InsertingIterable[U],
) -> InsertingIterable[U]:
    source_iterator = source.create_iterator()
    while source_iterator.has_next_elem():
        source_element = source_iterator.get_next_elem()
        destination.insert(mapped(source_element))
    return destination


def filter(
    source: Iterable[T],
    is_included: Callable[[T], bool],
    destination: InsertingIterable[T],
) -> InsertingIterable[T]:
    source_iterator = source.create_iterator()
    while source_iterator.has_next_elem():
        source_element = source_iterator.get_next_elem()
        if is_included(source_element):
            destination.insert(source_element)
    return destination
```

Let's define the following concrete collection classes:

Figure 4.49

```
class List(Collection[T]):
    def __init__(self, *args: T):
        # ...

    # ...


class Stack(Collection[T]):
    # ...


class Set(Collection[T]):
    # ...
```

Now, we can use the `map` and `filter` algorithms with the above-defined collection classes:

Figure 4.50

```
numbers = List(1, 2, 3, 3, 3, 50, 60)
is_less_than_10 = lambda number: number < 10
unique_less_than_10_numbers = filter(numbers, is_less_than_10,
Set())

doubled = lambda number: 2 * number
stack_of_doubled_numbers = map(numbers, doubled, Stack())
```

Let's create an asynchronous version of the `map` algorithm:

Figure 4.51

```
from collections.abc import Callable
from typing import Protocol, TypeVar

T = TypeVar('T')
U = TypeVar('U')


class Closeable(Protocol):
    def close(self) -> None:
        pass


class MaybeInserting(Protocol[T]):
    class InsertError(Exception):
        pass

    async def try_insert(self, value: T) -> None:
        pass


class CloseableMaybeInserting(Closeable, MaybeInserting[T]):
    pass


class MapError(Exception):
    pass


async def try_map(
    source: Iterable[T],
    mapped: Callable[[T], U],
    destination: CloseableMaybeInserting[U],
) -> None:
    source_iterator = source.create_iterator()
    try:
        while source_iterator.has_next_elem():
            source_element = source_iterator.get_next_elem()
            await destination.try_insert(mapped(source_element))
    except destination.InsertError as error:
        raise MapError(error)
    finally:
        destination.close()
```

Let's create a `FileLineInserter` class that implements the `CloseableMaybeInserting` protocol using the aiofiles library:

Figure 4.52

```
from typing import Final

from aiofiles import open


class FileLineInserter(CloseableMaybeInserting[T]):
    def __init__(self, file_path_name: str):
        self.__file = None
        self.__file_path_name: Final = file_path_name

    async def try_insert(self, value: T):
        if self.__file is None:
```

```
            self.__file = await open(self.__file_path_name,
mode='w')
        line = str(value) + '\n'
        await self.__file.write(line)

    def close(self):
        self.__file.close()
```

Let's use the above-defined `try_map` algorithm and the `FileLineInserter` class to write doubled numbers (one number per line) to a file named *file.txt*:

Figure 4.53

```
from asyncio import run

numbers = [1, 2, 3, 2, 1, 50, 60]
doubled = lambda number: 2 * number


async def my_func():
    try:
        await try_map(numbers, doubled,
FileLineInserter('file.txt'))
    except MapError as error:
        print(str(error))


run(my_func())
```

Python's standard library utilizes interface segregation and multiple interface inheritance in an exemplary way. For example, the Python standard library defines the below listed abstract base classes (or interfaces) that implement a single method only. I.e., they are microinterfaces.

Abstract base class	Method
Container	__contains__
Hashable	__hash__
Iterable	__iter__
Sized	__len__
Callable	__call__
Awaitable	__await__
AsyncIterable	__aiter__

Python standard library also contains the below abstract base classes that inherit from multiple (micro)interfaces:

Abstract base class	Inherits from
`Collection`	`Sized`, `Iterable`, `Container`
`Sequence`	`Collection`, `Reversible`

4.5.5: Program Against Interfaces Principle (Generalized Dependency Inversion Principle)

> ***Do not write programs where internal dependencies are concrete object types—instead, program against interfaces. An exception to this rule is data classes with no behavior (not counting simple getters/setters)***.

An interface is used to define an abstract base type. Various implementations that implement the interface can be introduced. When you want to change the behavior of a program, you create a new class that implements an interface and then use an instance of that class. In this way, you can practice the *open-closed principle*. You can think of this principle as a prerequisite for using the *open-closed principle* effectively. The *program against interfaces principle* was presented by the *Gang of Four* in their book *Design Patterns* and can be seen as a generalization of the *dependency inversion principle* from the SOLID principles:

> The *dependency inversion principle* is a methodology for loosely coupling software classes. When following the principle, the conventional dependency relationships from high-level classes to low-level classes are reversed, thus making the high-level classes independent of the low-level implementation details.

The *dependency inversion principle* states:

1. High-level classes should not import anything from low-level classes
2. Abstractions (= interfaces) should not depend on concrete implementations (classes)

3. Concrete implementations (classes) should depend on abstractions (= interfaces)

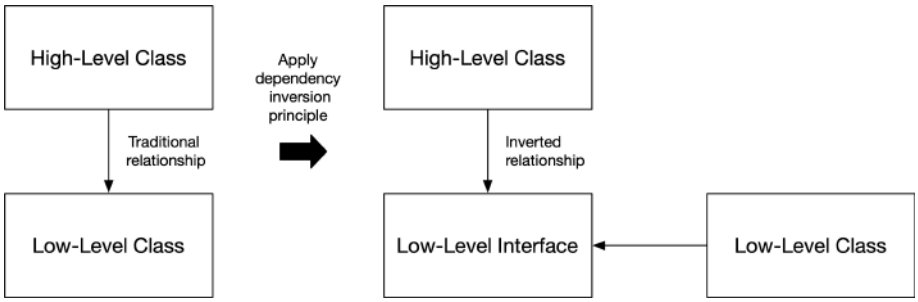

Figure 4.54. Dependency Inversion Principle

An interface is always an abstract type and cannot be instantiated. Below is an example of an interface:

Figure 4.55

```
from typing import Protocol


class Shape(Protocol):
    def draw(self) -> None:
        pass

    def calculate_area(self) -> float:
        pass
```

The name of an interface describes something abstract, which you cannot create an object of. In the above example, `Shape` is something abstract. You cannot create an instance of `Shape` and then draw it or calculate its area because you don't know what shape it is. But when a class implements an interface, a concrete object of the class representing the interface can be created. Below is an example of three different classes that implement the `Shape` interface:

Figure 4.56

```
from math import pi
from typing import Final


class CircleShape(Shape):
    def __init__(self, radius: int):
        self.__radius: Final = radius

    def draw(self) -> None:
        # ...

    def calculate_area(self) -> float:
```

```
        return pi * self.__radius**2


class RectangleShape(Shape):
    def __init__(self, width: int, height: int):
        self.__width: Final = width
        self.__height: Final = height

    def draw(self) -> None:
        # ...

    def calculate_area(self) -> float:
        return self.__width * self.__height


class SquareShape(RectangleShape):
    def __init__(self, side_length: int):
        super().__init__(side_length, side_length)
```

We should program against the `Shape` interface when using shapes in code. In the below example, we make a high-level class `Canvas` dependent on the `Shape` interface, not on any of the low-level classes (`CircleShape, RectangleShape` or `SquareShape)`. Now, the high-level `Canvas` class and all the low-level shape classes depend on abstraction only, the `Shape` interface. We can also notice that the high-level class `Canvas` does not import anything from the low-level classes. Also, the abstraction `Shape` does not depend on concrete implementations (classes).

Figure 4.57

```
from typing import Final


class Canvas:
    def __init__(self):
        self.__shapes: Final[list[Shape]] = []

    def add(self, shape: Shape) -> None:
        self.__shapes.append(shape)

    def draw_shapes(self) -> None:
        for shape in self.__shapes:
            shape.draw()
```

A `Canvas` object can contain any shape and draw any shape. It can handle any of the currently defined concrete shapes and any new ones defined in the future.

If you did not program against interfaces and did not use the dependency inversion principle, your `Canvas` class would look

like the following:

Figure 4.58

```
from typing import Final


class Circle:
    def draw(self) -> None:
        # ...


class Rectangle:
    def draw(self) -> None:
        # ...


class Square:
    def draw(self) -> None:
        # ...


class Canvas:
  def __init__(self):
      self.__shapes: Final[list[Circle | Rectangle | Square]] =
[]

  def add(self, shape: Circle | Rectangle | Square) -> None:
      self.__shapes.append(shape)

  def draw_shapes(self) -> None:
      for shape in self.__shapes:
          shape.draw()
```

The above high-level `Canvas` class is coupled with all the low-level classes (`Circle`, `Rectangle`, and `Square`). The type annotations in the `Canvas` class must be modified if a new shape type is needed. If something changes in the public API of any low-level class, the `Canvas` class needs to be modified accordingly. In the above example, we implicitly specify the protocol for the `draw` method: it does not take arguments and returns `None`. Relying on implicit protocols is not a good solution, especially in non-trivial applications. It is better to program against interfaces and make protocols explicit.

Let's have another example. If you have read books or articles about object-oriented design, you may have encountered something similar as is presented in the below example:

Figure 4.59

```
class Dog:
    def walk(self) -> None:
        # ...
```

```
    def bark(self) -> None:
        # ...


class Fish:
    def swim(self) -> None:
        # ...


class Bird:
    def fly(self) -> None:
        # ...

    def sing(self) -> None:
        # ...
```

Three concrete implementations are defined above, but no interface is defined. Let's say we are making a game that has different animals. The first thing to do when coding the game is to remember to program against interfaces and thus introduce an `Animal` protocol that we can use as an abstract base type. Let's try to create the `Animal` protocol based on the above concrete implementations:

Figure 4.60

```
from typing import Protocol


class Animal(Protocol):
    def walk(self) -> None:
        pass

    def bark(self) -> None:
        pass

    def swim(self) -> None:
        pass

    def fly(self) -> None:
        pass

    def sing(self) -> None:
        pass


class Dog(Animal):
    def walk(self) -> None:
        # ...

    def bark(self) -> None:
        # ...

    def swim(self) -> None:
        raise NotImplementedError()

    def fly(self) -> None:
        raise NotImplementedError()
```

```
    def sing(self) -> None:
        raise NotImplementedError()


# Fish class ...
# Bird class ...
```

The above approach is wrong. We declare that the `Dog` class implements the `Animal` protocol, but it does not do that. It implements only methods `walk` and `bark` while other methods throw an exception. We should be able to supply any concrete animal implementation where an animal is required. But it is impossible because if we have a `Dog` object, we cannot safely call `swim`, `fly`, or `sing` methods because they will always raise an error.

The problem is that we defined the concrete classes before defining the interface. That approach is wrong. We should specify the interface first and then the concrete implementations. What we did above was the other way around.

When defining an interface, we should remember that we are defining an abstract base type, so we must think in abstract terms. We must consider what we want the animals to do in the game. If we look at the methods `walk`, `fly`, and `swim`, they are all concrete actions. But what is the abstract action common to these three concrete actions? It is *move*. Walking, flying, and swimming are all ways of moving. Similarly, if we look at the `bark` and `sing` methods, they are also concrete actions. What is the abstract action common to these two concrete actions? It is *make sound*. And barking and singing are both ways to make a sound. When we use these abstract actions, our `Animal` protocol becomes the following:

Figure 4.61

```
from typing import Protocol


class Animal(Protocol):
    def move(self) -> None:
        pass

    def make_sound(self) -> None:
        pass
```

We can now redefine the animal classes to implement the new `Animal` protocol:

Figure 4.62

```
class Dog(Animal):
    def move(self) -> None:
        # Walk ...

    def make_sound(self) -> None:
        # Bark ...


class Fish(Animal):
    def move(self) -> None:
        # Swim ...

    def make_sound(self) -> None:
        # Intentionally no operation
        # (Fishes typically don't make sounds)
        pass


class Bird(Animal):
    def move(self) -> None:
        # Fly ...

    def make_sound(self) -> None:
        # Sing ...
```

Now, we have the correct object-oriented design and can program against the `Animal` interface. We can call the `move` method when we want an animal to move and the `make_sound` method when we want an animal to make a sound.

After realizing that some birds don't fly at all, we can easily enhance our design. We can introduce two different implementations:

Figure 4.63

```
from abc import abstractmethod


class AbstractBird(Animal):
    @abstractmethod
    def move(self) -> None:
        pass

    def make_sound(self) -> None:
        # Sing ...


class FlyingBird(AbstractBird):
    def move(self) -> None:
        # Fly ...
```

```
class NonFlyingBird(AbstractBird):
    def move(self) -> None:
        # Walk ...
```

We might also later realize that not all birds sing but make different sounds. Ducks quack, for example. Instead of using inheritance as was done above, an even better alternative is to use *object composition*. We compose the `Bird` class of behavioral classes for moving and making sounds. This is called the *strategy pattern*, and is discussed later in this chapter. We can give different moving and sound-making strategies for bird objects upon construction.

Figure 4.64

```
class Mover(Protocol):
    def move(self) -> None:
        pass


class SoundMaker(Protocol):
    def make_sound(self) -> None:
        pass


class Bird(Animal):
    def __init__(self, mover: Mover, sound_maker: SoundMaker):
        self.__mover = mover
        self.__sound_maker = sound_maker

    def move(self):
        self.__mover.move()

    def make_sound(self):
        self.__sound_maker.make_sound()
```

I don’t advocate adding a design pattern name to code entity names, but for demonstration purposes, we could make an exception here, and I can show how the code would look when making the *strategy pattern* explicit:

Figure 4.65

```
class MovingStrategy(Protocol):
    def move(self) -> None:
        pass


class SoundMakingStrategy(Protocol):
    def make_sound(self) -> None:
        pass


class Bird(Animal):
    def __init__(
```

```
        self,
        moving_strategy: MovingStrategy,
        sound_making_strategy: SoundMakingStrategy,
    ):
        self.__moving_strategy = moving_strategy
        self.__sound_making_strategy = sound_making_strategy

    def move(self):
        self.__moving_strategy.move()

    def make_sound(self):
        self.__sound_making_strategy.make_sound()
```

Now, we can create birds with various behaviors for moving and making sounds. We can use the *factory pattern* to create different birds. The *factory pattern* is described in more detail later in this chapter. Let's introduce three different moving and sound-making behaviors and a factory to make three kinds of birds: goldfinches, ostriches, and domestic ducks.

Figure 4.66

```
from enum import Enum


class Flyer(Mover):
    def move(self) -> None:
        # fly


class Runner(Mover):
    def move(self) -> None:
        # run


class Walker(Mover):
    def move(self) -> None:
        # walk


class GoldfinchSoundMaker(SoundMaker):
    def make_sound(self) -> None:
        # Sing goldfinch specific songs


class OstrichSoundMaker(SoundMaker):
    def make_sound(self) -> None:
        # Make ostrich specific sounds like whistles,
        # hoots, hisses, growls, and deep booming growls
        # that sound like the roar of a lion


class Quacker(SoundMaker):
    def make_sound(self) -> None:
        # quack


class BirdType(Enum):
```

```
    GOLDFINCH = 1
    OSTRICH = 2
    DOMESTIC_DUCK = 3


class BirdFactory:
    def create_bird(self, bird_type: BirdType) -> Bird:
        match bird_type:
            case BirdType.GOLDFINCH:
                return Bird(Flyer(), GoldfinchSoundMaker())
            case BirdType.OSTRICH:
                return Bird(Runner(), OstrichSoundMaker())
            case BirdType.DOMESTIC_DUCK:
                return Bird(Walker(), Quacker())
            case _:
                raise ValueError('Unsupported bird type')
```

4.6: Clean Microservice Design Principle

> ***The clean microservice design promotes object-oriented design with the separation of concerns achieved by dividing software into layers using the dependency inversion principle (programming against interfaces).***

Clean microservice design focuses on creating a microservice core (the model, business logic) that is devoid of technological concerns, pushing those to an outer input/output interface adaptation layer that includes, e.g., the persistence mechanism (an output interface adapter) and controller (an input interface adapter), which can be considered as technological details that have nothing to do with the microservice core. The benefit of this approach is that you can modify technological details without affecting the microservice core.

Robert C. Martin a.k.a. Uncle Bob uses the term *clean architecture* in his book *Clean Architecture* for this same principle. I do not use the term *architecture* because I have reserved that term to designate the design of something larger (i.e., a software system) than a single service. Clean microservice design focuses on designing a single (micro)service conducting OOD in a particular fashion.

Clean microservice design is a relatively simple concept. If you have used the single responsibility principle, divided a

microservice into layers, and been programming against interfaces (using the dependency inversion principle), you may have applied the clean microservice design without knowing it. The clean microservice design principle also employs the *adapter pattern* from design patterns discussed later in this chapter. The adapter pattern is used in input and output interface adapters. We can create separate adapter classes for various input sources and output destinations. Many resources (books, websites, etc.) can explain the clean architecture in rather complex terms.

> If you are interested, there are similar concepts to *clean architecture* called hexagonal architecture and *onion architecture*. They have the same basic idea of separating technology-related code from the business logic code, making it easy to change technological aspects of the software without modifications to the business logic part. They can use different terms, and in hexagonal architecture, you have co-centric hexagons instead of circles like in clean architecture, but the basic idea in all of them is the same.

Clean microservice design comes with the following benefits for the service:

- Not tied to any single framework
- Not tied to any single API technology like REST or GraphQL
- Unit testable
- Not tied to a specific client (works with web, desktop, console, and mobile clients)
- Not tied to a specific database or other storage technology
- Not dependent on any specific external service implementation

A clean API microservice design consists of the following layers:

- Input and output interface adapters (e.g., controllers, repositories, etc.)

- Use cases (i.e., the features the microservice exposes outside)
- (Business) Entities

Use cases and entities together form the *core* or *model* of the service, also called the *business logic*. The outermost layer of the use case layer usually contains service classes that implement the use cases. For example, one method in a service class implements one use case. The use case layer can contain other layers of software needed to implement the use cases. The service classes serve as a facade to those other layers. The service classes (i.e., the facade) are meant to be used by the *input interface adapters*, i.e., the *controllers*. A controller is a common term used to describe an input interface adapter. Similarly, a repository is a common term used to describe an output interface adapter that stores information, usually in a persistent storage.

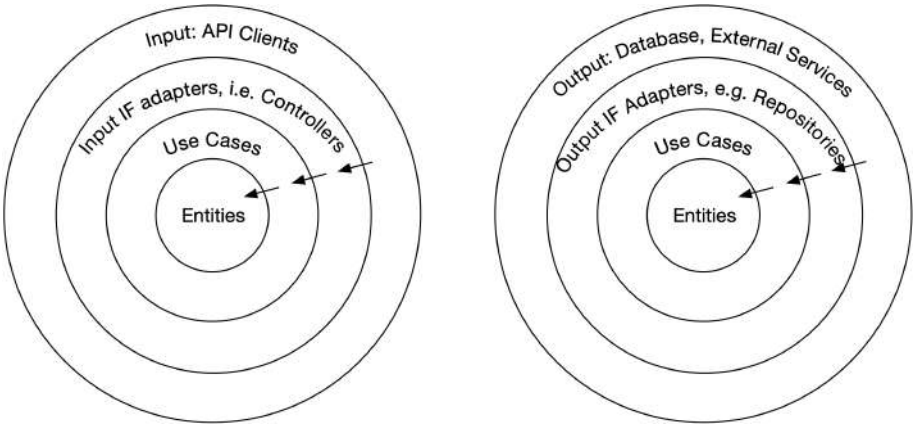

Figure 4.67. Clean Microservice Design

The direction of dependencies in the above diagrams is shown with arrows. We can see that the microservice's clients depend on the input interface adapter or *controller* we create. The controller depends on the use cases. The use case layer depends on (business) entities. The purpose of the use case layer is to orchestrate operations on the (business) entities. In the above figure, the parts of software that tend to change most often are located at the outer layers (e.g., controller technology like REST, GraphQL, and database) The most stable part of the software is located at the center (entities).

Let's have an example of an entity: a bank account. We know it is something that doesn't change often. It has a couple of key attributes: owner, account number, interest rate, and balance (and probably some other attributes), but what a bank account is or does has remained the same for tens of years. But we cannot say the same for API technologies or database technologies. Those are things that change at a much faster pace compared to bank accounts. Because of the direction of dependencies, changes in the outer layers do not affect the inner layers. The clean microservice design allows for easy API technology and database change, e.g., from REST to gRPC or SQL to NoSQL database. All these changes can be made without affecting the business logic (use case and entities layers).

Put entity-related business rules into entity classes. A `BankAccount` entity class should have a method for withdrawing money from the account. That method should enforce a business rule: Withdrawal is possible only if the account has enough funds. Don't put the withdrawal functionality into a service class and use `BankAccount's get_balance` and `set_balance` accessors to perform the withdrawal procedure in the service class. That would be against the *don't ask, tell principle* discussed later in this chapter. Also, don't allow service classes to access sub-entities (and their methods) a `BankAccount` possibly contains. That would be against DDD principles and the *law of demeter* (discussed later in this chapter). DDD states that you should access an aggregate (`BankAccount`) by its root only, not directly accessing any sub-entities.

Services orchestrate operations on one or more entities, e.g., a `BacklogItemService` can have a `setSprint` method that will fetch a `Sprint` entity and pass it to the `BacklogItem` entity's `setSprint` method, which verifies if the sprint for the backlog item can be set (only non-closed sprint can be set, i.e., current or future, not past sprints; a new sprint for a closed backlog item cannot be set. The closed backlog item has to remain in the sprint where it was closed.)

Let's assume that the `BacklogItem` entity class is large and the `setSprint` would contain a lot of validation code, making it

even larger. In that case, you should extract a new class for backlog item sprint validation business rules, and the `BacklogItem` class should be composed of that new behavioral class: `BacklogItemSprintValidator`. In the `BacklogItem` class's `setSprint` method, the validation can be done with the following call: `sprintValidator.validate(newSprint, this)`.

4.6.1: Real-Life Example

Let's have a real-life example of creating an API microservice called *order-service*, which handles orders in an e-commerce software system. First, we define our input interface adapter, which is a REST API controller using FastAPI:

Figure 4.68

```
from dependency_injector.wiring import Provide
from fastapi import APIRouter

from ..dtos.InputOrder import InputOrder
from ..dtos.OutputOrder import OutputOrder
from ..services.OrderService import OrderService

# In the request handler functions of the below class
# remember to add authorization, necessary audit logging and
# observability (metric updates) for production.
# Examples are provided in later chapters of this book


class RestOrderController:
    __order_service: OrderService = Provide['order_service']

    def __init__(self):
        self.__router = APIRouter()
        self.__router.add_api_route(
            '/orders/',
            self.create_order,
            methods=['POST'],
            status_code=201,
            response_model=OutputOrder,
        )
        self.__router.add_api_route(
            '/orders/{id_}',
            self.get_order,
            methods=['GET'],
            response_model=OutputOrder,
        )

    @property
    def router(self):
        return self.__router

    def create_order(self, input_order: InputOrder) ->
OutputOrder:
```

```
        return self.__order_service.create_order(input_order)

    def get_order(self, id_: int) -> OutputOrder:
        return self.__order_service.get_order(id_)

    # Rest of API endpoints...
```

The API offered by the microservice depends on the controller, as seen in the earlier diagram. The API is currently a REST API, but we could create and use a GraphQL controller. Then, our API, which depends on the controller, is a GraphQL API. You can create a controller for any client-server technology, like gRPC or WebSocket. You can even create a controller for standard input (stdin) or command line interface (CLI). A CLI controller reads command(s) from the command line arguments supplied to the microservice. Remember that you can have multiple controllers active in the same microservice. Your microservice could be used by frontend clients using a REST controller, or it could be used on the command line using its CLI controller.

As an example, below is a partial implementation of a GraphQL controller using FastAPI and Strawberry GraphQL library:

Figure 4.69

```
import strawberry
from dependency_injector.wiring import Provide
from strawberry.fastapi import GraphQLRouter

from ..graphqltypes.InputOrder import InputOrder
from ..graphqltypes.OutputOrder import OutputOrder
from ..services.OrderService import OrderService

order_service: OrderService = Provide['order_service']

# In the request handler functions of the below class
# remember to add authorization, necessary audit logging and
# observability (metric updates) for production.
# Examples are provided in later chapters of this book


class GraphQlOrderController:
    @strawberry.type
    class Query:
        @strawberry.field
        def order(self, id: int) -> OutputOrder:
            output_order = order_service.get_order(id)
            return OutputOrder.from_pydantic(output_order)

    @strawberry.type
    class Mutation:
```

```
        @strawberry.mutation
        def create_order(self, input_order: InputOrder) ->
OutputOrder:
            output_order = order_service.create_order(
                input_order.to_pydantic()
            )
            return OutputOrder.from_pydantic(output_order)

    __schema = strawberry.Schema(query=Query, mutation=Mutation)
    __router = GraphQLRouter(__schema, path='/graphql')

    @property
    def router(self):
        return self.__router
```

The `RestOrderController` and `GraphQlOrderController` classes depend on the `OrderService` interface, which is part of the use case layer. Notice that the controllers do not rely on a concrete implementation of the use cases but depend on an interface according to the *dependency inversion principle*. Below is the definition for the `OrderService` protocol:

Figure 4.70

```
from typing import Protocol

from ..dtos.InputOrder import InputOrder
from ..dtos.OutputOrder import OutputOrder


class OrderService(Protocol):
    def create_order(self, input_order: InputOrder) ->
OutputOrder:
        pass

    def get_order(self, id_: int) -> OutputOrder:
        pass

    def get_order_by_user_id(self, user_id: int) -> OutputOrder:
        pass

    def update_order(self, id_: int, order_update: InputOrder) -
> None:
        pass

    def delete_order(self, id_: int) -> None:
        pass
```

The below `OrderServiceImpl` class implements the `OrderService` protocol:

Figure 4.71

```
from dependency_injector.wiring import Provide

from ..dtos.InputOrder import InputOrder
from ..dtos.OutputOrder import OutputOrder
```

```
from ..errors.EntityNotFoundError import EntityNotFoundError
from ..repositories.OrderRepository import OrderRepository
from ..services.OrderService import OrderService


class OrderServiceImpl(OrderService):
    __order_repository: OrderRepository =
Provide['order_repository']

    def create_order(self, input_order: InputOrder) ->
OutputOrder:
        order = self.__order_repository.save(input_order)
        return OutputOrder.from_orm(order)

    def get_order(self, id_: int) -> OutputOrder:
        order = self.__order_repository.find(id_)

        if order is None:
            raise EntityNotFoundError('Order', id_)

        return OutputOrder.from_orm(order)

    # Rest of the methods...
```

The `OrderServiceImpl` class has a dependency on an order repository. This dependency is also inverted. The `OrderServiceImpl` class depends only on the `OrderRepository` interface. The order repository is used to orchestrate the persistence of order entities. Note that there is no direct dependency on a database. The term *repository* is abstract. It only means a place where data (entities) are stored. So the repository can be implemented by a relational database, NoSQL database, file system, in-memory cache, message queue, or another microservice, to name a few.

Below is the `OrderRepository` protocol:

Figure 4.72

```
from typing import Protocol

from ..dtos.InputOrder import InputOrder
from ..entities.Order import Order


class OrderRepository(Protocol):
    def initialize(self) -> None:
        pass

    def save(self, order: InputOrder) -> Order:
        pass

    def find(self, order_id: int) -> Order | None:
        pass

    # Rest of methods...
```

The `OrderRepository` interface depends only on the `Order` entity class. You can introduce an *output interface adapter* class that implements the `OrderRepository` interface. An output interface adapter adapts a particular concrete output destination (e.g., a database) to the `OrderRepository` interface. Entity classes do not depend on anything except other entities to create hierarchical entities (aggregates). For example, the `Order` entity consists of `OrderItem` entities. Let's introduce an `OrderRepository` *output interface adapter* class for an SQL database:

Figure 4.73

```
import os

from sqlalchemy import create_engine
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy.orm import sessionmaker

from ..dtos.InputOrder import InputOrder
from ..entities.Base import Base
from ..entities.Order import Order
from ..errors.DatabaseError import DatabaseError
from ..repositories.OrderRepository import OrderRepository
from ..utils import to_entity_dict


class SqlOrderRepository(OrderRepository):
    __engine = create_engine(os.environ.get('DATABASE_URL'))
    __SessionLocal = sessionmaker(
        autocommit=False, autoflush=False, bind=__engine
    )

    def __init__(self):
        try:
            Base.metadata.create_all(bind=self.__engine)
        except SQLAlchemyError as error:
            # Log error
            raise error

    def save(self, input_order: InputOrder) -> Order:
        with self.__SessionLocal() as db_session:
            try:
                order = Order(**to_entity_dict(input_order))
                db_session.add(order)
                db_session.commit()
                db_session.refresh(order)
                return order
            except SQLAlchemyError as error:
                raise DatabaseError(error)

    def find(self, id_: int) -> Order | None:
        with self.__SessionLocal() as db_session:
            try:
                return db_session.get(Order, id_)
```

```
            except SQLAlchemyError as error:
                raise DatabaseError(error)

    # Rest of methods...
```

The above class requires the used database service to be configured in an environment variable named `DATABASE_URL`. For a local MySQL database, you could use:

Figure 4.74

```
# Assuming username:password to be root:password
export
DATABASE_URL=mysql://root:password@localhost:3306/orderservice
```

> If you are interested in the implementation of the `to_entity_dict` method, please check Appendix A.

Changing the database to MongoDB can be done by creating a new output interface adapter class that implements the `OrderRepository` interface. We will implement a MongoDB repository in the coming *database principles* chapter.

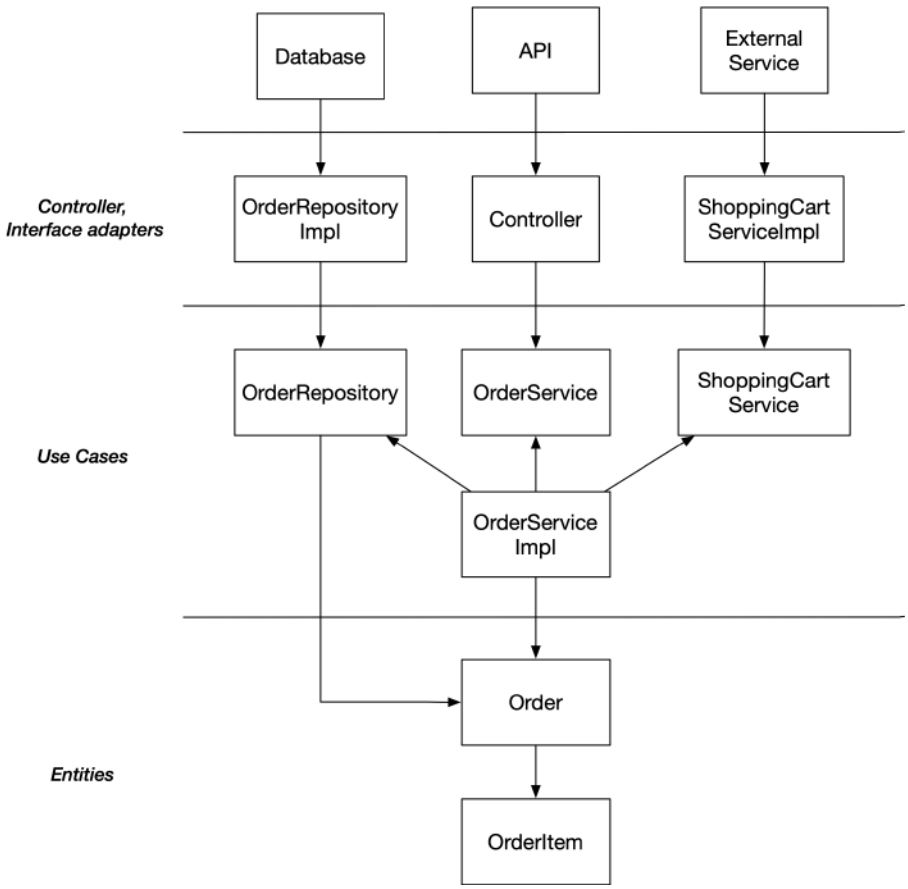

Figure 4.75. Clean Microservice Design for Order Service

When implementing a clean microservice design, everything is wired together using configuration and dependency injection. Dependency injection is configured using the dependency-injector library and by defining a `DiContainer` class:

Figure 4.76

```
from dependency_injector import containers, providers

from .controllers.GraphQlOrderController import
GraphQlOrderController
from .repositories.SqlOrderRepository import SqlOrderRepository
from .services.OrderServiceImpl import OrderServiceImpl


class DiContainer(containers.DeclarativeContainer):
    wiring_config = containers.WiringConfiguration(
        modules=[
            '.services.OrderServiceImpl',
            '.controllers.RestOrderController',
            '.controllers.FlaskRestOrderController',
            '.controllers.GraphQlOrderController',
            '.repositories.SqlOrderRepository',
        ]
    )
```

```
    order_service = providers.Singleton(OrderServiceImpl)
    order_repository = providers.Singleton(SqlOrderRepository)
    # order_controller =
providers.Singleton(RestOrderController)
    order_controller =
providers.Singleton(GraphQlOrderController)
```

If we want to change something in our microservice, we can create a new class and use that new class in the `DiContainer`. We could create a new repository class for a different type of database, create a new service class that implements part of services locally and part of them remotely, or introduce a new controller for gRPC, for example. All these changes would be according to the *open-closed principle* because we are not modifying any existing classes (except for the `DiContainer` of course) but we are extending our application by introducing new classes that implement existing interfaces.

In the *app.py* file, we create an instance of the DI container, create the FastAPI app, define an error handler for mapping business errors to HTTP responses and finally wire the wanted controller to the FastAPI app:

Figure 4.77

```
from fastapi import FastAPI, Request
from fastapi.exceptions import RequestValidationError
from fastapi.responses import JSONResponse

from .DiContainer import DiContainer
from .errors.OrderServiceError import OrderServiceError
from .utils import get_stack_trace

di_container = DiContainer()
app = FastAPI()


@app.exception_handler(OrderServiceError)
def handle_order_service_error(request: Request, error:
OrderServiceError):
    # Log error.cause

    # Increment 'request_failures' counter by one
    # with labels:
    # api_endpoint=f'{request.method} {request.url}'
    # status_code=error.status_code

    return JSONResponse(
        status_code=error.status_code,
        content={
            'errorMessage': error.message,
            'stackTrace': get_stack_trace(error.cause),
        },
    )
```

```
@app.exception_handler(RequestValidationError)
def handle_request_validation_error(
    request: Request, error: RequestValidationError
):
    # Audit log

    # Increment 'request_failures' counter by one
    # with labels:
    # api_endpoint=f'{request.method} {request.url}'
    # status_code=400

    return JSONResponse(
        status_code=400,
        content={'errorMessage': str(error)},
    )


@app.exception_handler(Exception)
def handle_unspecified_error(request: Request, error:
Exception):
    # Increment 'request_failures' counter by one
    # with labels:
    # api_endpoint=f'{request.method} {request.url}'
    # status_code=500
    # error_code='UnspecifiedError'

    return JSONResponse(
        status_code=500,
        content={
            'errorMessage': str(error),
            'stackTrace': get_stack_trace(error),
        },
    )


order_controller = di_container.order_controller()
app.include_router(order_controller.router)
```

> Please check Appendix A if you are interested in the rest of the code (i.e., DTOs, entities, GraphQL schema (= types), and errors. When defining an identifier field for an entity/DTO, you should define it as a string. This will allow passing large integer values (64-bit or larger) to JavaScript clients. You can also use non-integer identifiers that have a string representation, like MongoDB object ids or UUIDs. Let's say you have a repository interface implemented by a file system repository. In that case, you must have a way to generate entity ids. You can use UUIDs or store the latest used id in the file system and use that value as the basis when getting a new id value.

The dependency injection container is the only place in a microservice that contains references to concrete implementations. The *dependency injection principle* is

discussed more in a later section of this chapter. The dependency inversion principle and dependency injection principle usually go hand in hand. Dependency injection is used for wiring interface dependencies so that those become dependencies on concrete implementations, as shown in the figure below.

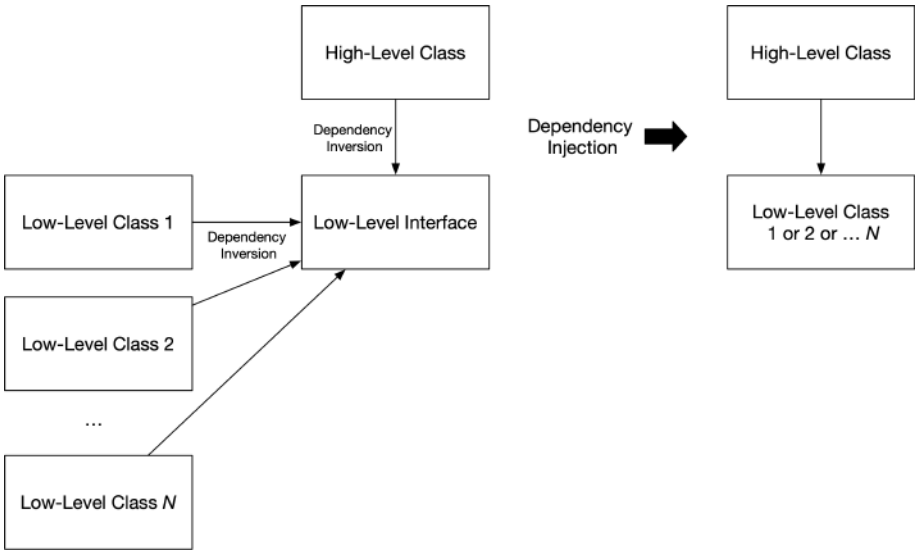

Figure 4.78. Fig 3.4 Dependency Injection

Let's add a feature where the shopping cart is emptied when an order is created:

Figure 4.79

```
from dependency_injector.wiring import Provide

from ..dtos.InputOrder import InputOrder
from ..dtos.OutputOrder import OutputOrder
from ..repositories.OrderRepository import OrderRepository
from ..services.OrderService import OrderService
from ..services.ShoppingCartService import ShoppingCartService


class OrderServiceImpl(OrderService):
    __order_repository: OrderRepository =
Provide['order_repository']
    __shopping_cart_service: ShoppingCartService = Provide[
        'shopping_cart_service'
    ]

    def create(self, order: InputOrder) -> OutputOrder:
        self.__shopping_cart_service.empty_cart(order.user_id)
        return self.__order_repository.save(order)

    # Rest of the methods...
```

As you can see from the above code, the `ShoppingCartOrderService` class does not depend on any concrete implementation of the shopping cart service. We can create an *output interface adapter* class that is a concrete implementation of the `ShoppingCartService` interface. For example, that interface adapter class connects to a particular external shopping cart service via a REST API. Once again, the dependency injector will inject a concrete `ShoppingCartService` implementation to an instance of the `ShoppingCartOrderService` class.

Note that the above `create_order` method is not production quality because it lacks a transaction.

Now we have seen examples of the following benefits of clean microservice design:

- Not tied to any single API technology like REST or GraphQL
- Not tied to a specific client (works with web, desktop, console, and mobile clients)
- Not tied to a specific database
- Not dependent on any specific external service implementation

Let's showcase the final benefit:

- Not tied to any single framework

Let's change the used web framework from FastAPI to Flask. What we need to do is to create a Flask-specific version of the `RestOrderController`:

Figure 4.80

```
from dependency_injector.wiring import Provide
from flask import Response, jsonify, request
from flask_classful import FlaskView, route

from ..dtos.InputOrder import InputOrder
from ..services.OrderService import OrderService

# In the request handler functions of the below class
# remember to add authorization, necessary audit logging and
# observability (metric updates) for production.
# Examples are provided in later chapters of this book
```

```
class FlaskRestOrderController(FlaskView):
    __order_service: OrderService = Provide['order_service']

    @route('/orders', methods=['POST'])
    def create_order(self) -> Response:
        output_order = self.__order_service.create_order(
            InputOrder(**request.json)
        )
        return jsonify(output_order.dict()), 201

    @route('/orders/<id_>')
    def get_order(self, id_: int) -> Response:
        output_order = self.__order_service.get_order(id_)
        return jsonify(output_order.dict())

    # Rest of API endpoints...
```

And finally, we shall create a Flask-specific application file *app_flask.py*:

Figure 4.81

```
import json

from flask import Flask, Response

from .controllers.FlaskRestOrderController import
FlaskRestOrderController
from .DiContainer import DiContainer
from .errors.OrderServiceError import OrderServiceError
from .utils import get_stack_trace

di_container = DiContainer()
app = Flask(__name__)


@app.errorhandler(OrderServiceError)
def handle_order_service_error(error: OrderServiceError):
    return Response(
        json.dumps(
            {
                'errorMessage': error.message,
                'stackTrace': get_stack_trace(error.cause),
            }
        ),
        status=error.status_code,
        mimetype='application/json',
    )


@app.errorhandler(Exception)
def handle_unspecified_error(error: Exception):
    return Response(
        json.dumps(
            {
                'errorMessage': str(error),
                'stackTrace': get_stack_trace(error),
            }
```

```
        ),
        status=500,
        mimetype='application/json',
    )


FlaskRestOrderController.register(app, route_base='/')

if __name__ == '__main__':
    app.run()
```

We were able to change the used web framework by introducing two new small modules. We did not touch any existing modules. We can be confident that we did not break any existing functionality. We were once again successfully applying the *open-closed principle* to our codebase.

It should be noted that the clean microservice design principle applies to other microservices with input or output, not just API microservices. We will have an example of this later in this chapter.

4.7: Vertical Slice Design Principle

Vertical slice design is also known as *vertical slice architecture*. Vertical slice design divides individual features into separate vertical slices. In the source code repository, this means a separate directory for each vertical slice (feature). You can also group these single feature directories under a domain directory. This makes navigating the code-base a breeze and finding a particular feature easy. The main benefit of vertical slice design is that you can follow the open-closed principle when adding new features. There is no need to modify existing code and no risk of breaking existing functionality. You can implement a vertical slice (a feature) as you wish, which is a clear benefit when different features have divergent requirements. So, a vertical slice can contain a single class or multiple classes that form a layered design. Vertical slice design does not enforce any specific way. However, using some design principles for a vertical slice implementation is good practice. I suggest to use the *clean microservice design*. You can use vertical slice design and clean microservice

design principles together. They are not mutually exclusive principles.

If your microservice is a small API, e.g., just containing a handful of simple CRUD operations, you don't necessarily get much benefit from vertical slicing. If you are not using the vertical slice design, adding a new CRUD operation requires changes to existing classes. These changes include adding new methods to mostly immutable or stateless classes, which can be considered extensions instead of modifications and are according to the open-closed principle. Adding a method to an immutable or stateless class rarely breaks existing functionality in existing methods. This is why I haven't used vertical slicing in the examples in this book.

For example, suppose you have an API providing CRUD operations on a resource and want to use vertical slicing. In that case, you implement each CRUD operation separately in different service/controller/repository classes. Here is an example of the source code directory layout for the *order-service* using vertical slice design:

Figure 4.82

```
order-service
└── src
    └── order
        ├── common
        │   ├── dtos
        │   │   ├── InputOrder.py
        │   │   └── OutputOrder.py
        │   └── entities
        │       └── Order.py
        ├── create
        │   ├── CreateOrderRepository.py
        │   ├── CreateOrderService.py
        │   ├── CreateOrderServiceImpl.py
        │   ├── RestCreateOrderController.py
        │   └── SqlCreateOrderRepository.py
        ├── get
        │   ├── GetOrderRepository.py
        │   ├── GetOrderService.py
        │   ├── GetOrderServiceImpl.py
        │   ├── RestGetOrderController.py
        │   └── SqlGetOrderRepository.py
        ├── update
        │   ├── UpdateOrderRepository.py
        │   ├── UpdateOrderService.py
        │   ├── UpdateOrderServiceImpl.py
        │   ├── RestUpdateOrderController.py
        │   └── SqlUpdateOrderRepository.py
        └── delete
            ├── DeleteOrderRepository.py
```

```
├── DeleteOrderService.py
├── DeleteOrderServiceImpl.py
├── RestDeleteOrderController.py
└── SqlDeleteOrderRepository.py
```

4.8: Class Organization Principle

A class should be organized in the following manner:

- Attributes first, methods after them
- Method order: constructor first, then public, then protected, and private last
- Order attributes and public methods by importance or logic, and if there is no logic or importance difference, order alphabetically
- Define public getters/properties/setters after other public methods
- Order private methods in the same order they are used in the public methods

An example of the *logic* is a `Rectangle` class with `width` and `height` attributes. You should give the `width` (x-axis length) before `height` (y-axis length) because coordinates are given `x` first and then `y`.

Let's have an example with a `Circle` class. It has the following attributes:

- `origin` (vital because you cannot draw a circle without knowing its origin)
- `radius` (vital because you cannot draw a circle without knowing its radius)
- `stroke_color` (this is a must attribute even though a default value could be used)
- `fill_color` (this is an optional attribute; a default value of `None` could be used)

Attributes `origin` and `radius` should be given in that order because you need to know the *origin* before you can start

drawing the circle.

The `Circle` class has the following methods:

- draw (most used method, should be the first)
- calculate_area (calculateXXX methods should be in alphabetical order because there is no logic or importance difference between methods)
- calculate_perimeter
- getters/properties (in the same order as respective attributes)
- setters (in the same order as respective attributes)

Our `Circle` class would like the following:

Figure 4.83

```
from Point import Point


class Circle:
    def __init__(
        self,
        origin: Point,
        radius: float,
        stroke_color: str,
        fill_color: str = None
    ):
        self.__origin = origin
        self.__radius = radius
        self.__stroke_color = stroke_color
        self.__fill_color = fill_color

    def draw(self) -> None:
        # ...

    def calculate_area(self) -> float:
        # ...

    def calculate_perimeter(self) -> float:
        # ...

    @property
    def origin(self) -> Point:
        return self.__origin

    @property
    def radius(self) -> float:
        return self.__radius

    @property
    def stroke_color(self) -> str:
        return self.__stroke_color
```

```
    @property
    def fill_color(self) -> str:
        return self.__fill_color

    @origin.setter
    def origin(self, origin: Point) -> None:
        self.__origin = origin

    @radius.setter
    def radius(self, radius: float) -> None:
        self.__radius = radius

    @stroke_color.setter
    def stroke_color(self, stroke_color: str) -> None:
        self.__stroke_color = stroke_color

    @fill_color.setter
    def fill_color(self, fill_color: str) -> None:
        self.__fill_color = fill_color
```

4.9: Uniform Naming Principle

> ***Use a uniform way to name interfaces, classes, and functions.***.

This section presents conventions for uniformly naming interfaces, classes, and functions.

4.9.1: Naming Interfaces and Classes

> ***Classes represent a thing or an actor. They should be named consistently so the class name ends with a noun. An interface represents an abstract thing, actor, or capability. Interfaces representing a thing or an actor should be named like classes but using an abstract noun. Interfaces representing a capability should be named according to the capability.***

When an interface represents an abstract thing, name it according to that abstract thing. For example, if you have a drawing application with various geometrical objects, name the geometrical object interface `Shape`. It is a simple abstract noun. Names should always be the shortest, most descriptive ones. There is no reason to name the geometrical object

interface as `GeometricalObject` or `GeometricalShape`, if we can use simply `Shape`.

When an interface represents an abstract actor, name it according to that abstract actor. The name of an interface should be derived from the functionality it provides. For example, suppose there is a `parse_config` method in the interface. In that case, the interface should be named `ConfigParser`, and if an interface has a `validate_object` method, the interface should be named `ObjectValidator`. Don't use mismatching name combinations like a `ConfigReader` interface with a `parse_config` method or an `ObjectValidator` interface with a `validate_data` method.

When an interface represents a capability, name it according to that capability. Capability is something that a concrete class is capable of doing. For example, a class could be sortable, iterable, comparable, equitable, etc. Name the respective interfaces according to the capability: `Sortable`, `Iterable`, `Comparable`, and `Equitable`. The name of an interface representing a capability usually ends with *able* or *ing*.

Don't name interfaces starting with the *I* prefix (or any other prefix or postfix). Instead, use an *Impl* postfix for class names to distinguish a class from an interface, but only when needed. For example, if you have an interface named `ConfigParser` and you have a concrete class implementing the interface and parsing configuration in JSON format, name the class `JsonConfigParser`, not `JsonConfigParserImpl`, because the `Impl` prefix is not needed to distinguish between the interface and implementing class. Remember that you should be programming against interfaces, and if every interface has its name prefixed with an *I*, it just adds unnecessary noise to the code.

Some examples of class names representing a thing are: `Account`, `Order`, `RectangleShape`, and `CircleShape`. In a class inheritance hierarchy, the names of classes usually refine the interface name or the base class name. For example, if there is an `InputMessage` interface, then there can be different concrete implementations (= classes) of the `InputMessage` interface. They can represent an input message from different

sources, like `KafkaInputMessage` and `HttpInputMessage`. And there could be different subclasses for different data formats: `AvroBinaryKafkaInputMessage` or `JsonHttpInputMessage`.

The interface or base class name should be retained in the class or subclass name. Class names should follow the pattern: `<class-purpose>` + `<interface-name>` or `<sub-class-purpose>` + `<super-class-name>`, e.g., `Kafka` + `InputMessage` = `KafkaInputMessage` and `AvroBinary` + `KafkaInputMessage` = `AvroBinaryKafkaInputMessage`. Name abstract classes with the prefix `Abstract`.

If an interface or class name is over 20-30 characters long, consider abbreviating one or more words in the name. The reason for this is to keep the code readable. Very long names are harder to read and slow a developer down. (Remember that code is more often read than written).

Only use abbreviations that are commonly used and understandable for other developers. If a word does not have a good abbreviation, don't abbreviate. For example, in the class name `AvroBinaryKafkaInputMessage`, we can only abbreviate the `Message` to `Msg`. There are no well-established abbreviations for other words in the class name. Abbreviating `Binary` to `Bin` is questionable because `Bin` could also mean a *bin*. Don't abbreviate a word if you benefit only one or two characters. For example, there is no reason to abbreviate `Account` to `Accnt`.

Instead of abbreviating, you can shorten a name by dropping one or more words from it, provided readers can still easily understand it. For example, if you have classes `InternalMessage`, `InternalMessageSchema` and `InternalMessageField`, you could shorten the last two class names to `InternalSchema` and `InternalField`. This is because these two classes are mainly used in conjunction with the `InternalMessage` class: An `InternalMessage` object has a schema and one or more fields. You can also use nested classes: `InternalMessage.Schema` and `InternalMessage.Field`. The problem with nested classes is that they can make the module too large.

If you have related classes and one or more class names require shortening, you should shorten all related class names to keep the naming uniform. For example, if you have two classes, `ConfigurationParser` and `JsonConfigurationParser`, you should shorten the names of both classes, not only the one longer than 19 characters. The new class names would be `ConfigParser` and `JsonConfigParser`.

If an interface or class name is less than 20 characters long, there is usually no need to shorten it.

Don't add a design pattern name to a class name if it does not bring any real benefit. For example, suppose we have a `DataStore` interface, a `DataStoreImpl` class, and a class wrapping a `DataStore` instance using the *proxy pattern* to add caching functionality to the wrapped data store. We should not name the caching class `CachingProxyDataStore` or `CachingDataStoreProxy`. The word *proxy* does not add significant value, so the class should be named simply `CachingDataStore`. That name clearly tells it is a question about a data store with caching functionality. A seasoned developer notices from the `CachingDataStore` name that the class uses the *proxy pattern*. And if not, looking at the class implementation will finally reveal it.

4.9.2: Naming Functions

> ***Functions should do one thing, and the name of a function should describe what the function does. The function name usually contains a verb that indicates what the function does. The function name often starts with a verb, but exceptions exist. If a function returns a value, try to name the function so that the function name describes what it returns.***

The general rule is to name a function so that the purpose of the function is clear. A good function name should not make you think. If a function name is *20 or more characters long*, consider abbreviating one or more words in the name. The

reason for this is to keep the code readable. Very long names are harder to read and slow a developer down. (Remember that code is more often read than written). Only use abbreviations that are widely used and understandable for other developers. If a word does not have a good abbreviation, don't abbreviate.

Below is an example of a protocol containing two methods named with simple verbs only. It is not necessary to name the methods as `start_thread` and `stop_thread` because the methods are already part of the `Thread` interface, and it is self-evident what the `start` method starts and what the `end` method ends. You don't need to repeat the class name in the method name.

Figure 4.84

```
from typing import Protocol


class Thread(Protocol):
    def start(self) -> None:
        pass

    def stop(self) -> None:
        pass
```

Many languages offer streams that can be written to, like the standard output stream. Streams are usually buffered, and the actual writing to the stream does not happen immediately. For example, the below statement does not necessarily write to the stream immediately. It buffers the text to be written later when the buffer is flushed to the stream. This can happen when the buffer is full, some time has elapsed since the last flush, or when the stream is closed.

Figure 4.85

```
stream.write(...)
```

The above statement is misleading and could be corrected by renaming the function to describe what it actually does:

Figure 4.86

```
stream.write_on_flush(...)
```

The above function name immediately tells a developer that writing happens only on flush, and the developer can consult

the function documentation to determine when the flushing happens.

You can introduce a convenience method to perform a write with an immediate flush:

Figure 4.87

```
# Instead of this:
stream.write_on_flush(...)
stream.flush()

# User can do this:
stream.write_with_flush(...)
```

Many times, the function's action is associated with a target, for example:

Figure 4.88

```
from typing import Protocol


class ConfigParser(Protocol):
    def try_parse_config(self, config_json: str) -> Config:
        pass
```

When a function's action has a target, it is useful to name the function using the following pattern: `<action-verb>` + `<action-target>`, for example, `parse` + `config` = `parse_config`.

We can drop the action target from the function name if the function's first parameter describes the action target. It is not wrong to keep the action target in the function name, though. But if it can be dropped, it usually makes the function call statements read better. In the below example, the word "config" appears repeated: `try_parse_config(config_json)`, which makes the function call statement read a bit clumsy.

Figure 4.89

```
config = config_parser.try_parse_config(config_json)
```

We can drop the action target from the function name:

Figure 4.90

```
class ConfigParser(Protocol):
    def try_parse(self, config_json: str) -> Config:
        pass
```

As shown below, this change makes the code read better, presuming we use a descriptive variable name. And we should, of course, always use a descriptive variable name.

Figure 4.91

```
config = config_parser.try_parse(config_json);
```

Here is another example:

Figure 4.92

```
from typing import Protocol, TypeVar

T = TypeVar('T')


class Vector(Protocol[T]):
    # OK
    def push_back(self, value: T) -> None:
        pass

    # Not ideal,
    # word "value" repeated
    def push_back_value(self, value: T) -> None:
        pass
```

Let's imagine we have the following function:

Figure 4.93

```
class KafkaAdminClient:
    @staticmethod
    def create(self, topic: str) -> None:
        # ...
```

The above function name should be used only when a topic is the only thing a Kafka admin client can create because Python does not support method overloading.

There are two correct ways to call the method:

Figure 4.94

```
KafkaAdminClient.create(topic='xyz')
```

or

Figure 4.95

```
topic = "xyz"
KafkaAdminClient.create(topic)
```

4.9.2.1: Preposition in Function Name

Use a preposition in a function name when needed to clarify the function's purpose.

You don't need to add a preposition to a function name if the preposition can be assumed (i.e., the preposition is implicit). In many cases, only one preposition can be assumed. If you have a function named `wait`, the preposition `for` can be assumed, and if you have a function named `subscribe`, the preposition `to` can be assumed. We don't need to use function names `wait_for` and `subscribe_to`.

Suppose a function is named `laugh(person: Person)`. Now, we have to add a preposition because none can be assumed. We should name the function either `laugh_with(person: Person)` or `laugh_at(person: Person)`.

Let's analyze the Python's list methods and see how well they are named:

`list.append(item)`

> This tells clearly where the value is put in the list (at the end). This reads well.

`list.remove(item)`

> This method removes only the first item found in the list. Therefore, it should be named `remove_first(item)`. This method can also raise an exception. We should communicate that in the method signature. Let's use a *try* prefix: `list.try_remove_first(item)`. The next chapter will discuss exception handling and *try* prefix more.

`list.pop(index)`

> This reads well. We can easily assume an `at` preposition after the `pop` word.

`list.index(item)`

> The word *index* is not a verb here. We could add a correct verb and inform the user that the first index of the found item is returned. This method can also raise an exception. We should communicate that in the method signature

(try-prefix). This method could be renamed to `list.try_find_first_index(item)`.

`list.count(item)`

This reads well. We can easily assume an `of` preposition after the `count` word.

`list.sort()`

This is perfect. It informs that the list is sorted in place. If the method returned a new sorted list, it should be called `list.sorted()`

`list.reverse()`

This is perfect. It informs that the list is reversed in place. If the method returned a new reversed list, it should be called `list.reversed()`

`list.copy()`

The word *copy* is strongly associated with copying from one place to another. I would rename this method to `list.clone()`

4.9.2.2: Naming Method Pairs

Methods in a class can come in pairs. A typical example is a pair of getter and setter methods. Name the methods logically when you define a method pair in a class. The methods in a method pair often do two opposite things, like getting or setting a value. If you are unsure how to name one of the methods, try to find an antonym for a word. For example, if you have a method whose name starts with “create” and are unsure how to name the method for the opposite action, try a Google search: “create antonym”.

Here is a non-comprehensive list of some method names that come in pairs:

- get/put (e.g., when accessing a non-sequence collection like set or map)
- read/write

- add/remove
- store/retrieve
- open/close
- load/save
- initialize/destroy
- create/destroy
- insert/delete
- start/stop
- pause/resume
- start/finish
- increase/decrease
- increment/decrement
- construct/destruct
- encrypt/decrypt
- encode/decode
- obtain/relinquish
- acquire/release
- reserve/release
- startup/shutdown
- login/logout
- begin/end
- launch/terminate
- publish/subscribe
- join/detach
- <something>/un<something>, e.g., assign/unassign, install/uninstall, subscribe/unsubscribe, follow/unfollow
- <something>/de<something>, e.g., serialize/deserialize, allocate/deallocate
- <something>/dis<something>, e.g., connect/disconnect

Let's have a couple of examples from real life. The `apt` tool in Debian/Ubuntu-based Linux has an `install` command to install a package, but the command for uninstalling a package is `remove`. It should be `uninstall`. The Kubernetes package manager Helm has this correct. It has an `install` command to install a Helm release and an `uninstall` command to uninstall it.

4.9.2.3: Naming Boolean Functions (Predicates)

> ***The naming of boolean functions (predicates) should be such that when reading the function call statement, it reads as a boolean statement that can be true or false.***

In this section, we consider naming functions that are predicates and return a boolean value. Here, I don't mean functions that return true or false based on the success of the executed action, but cases where the function call is used to evaluate a statement as true or false. The naming of boolean functions should be such that when reading the function call statement, it makes a statement that can be true or false. Below are some examples:

Figure 4.96

```
class Response:
    def has_error(self) -> bool:
        # ...


class String:
    def is_empty(self) -> bool:
        # ...

    def starts_with(self, another_string: str) -> bool:
        # ...

    def ends_with(self, another_string: str) -> bool:
        # ...

    def contains(self, another_string: str) -> bool:
        # ...


# Here we have a statement: response has error? True or false?
if response.has_error():
    # ...

# Here we have a statement: line is empty? True or false?
```

```
line: String = file_reader.read_line()
if line.is_empty():
    # ...

# Here we have statement: line starts with a space character?
# True or false?
if line.starts_with(' '):
    # ...

# Here we have statement: line ends with a semicolon?
# True or false?
if line.ends_with(";"):
    # ...


class Thread:
    def should_terminate(self) -> bool:
        # ...

    def is_paused(self) -> bool:
        # ...

    def can_resume_execution() -> bool:
        # ...

    def run(self) -> None:
        # ...

        # Here we have statement: self should terminate?
        # True or false?
        if self.should_terminate():
            return


        # Here we have statement: self is paused and
        # self can resume execution? True or false?
        if self.is_paused() and self.can_resume_execution():
            # ...

        # ...
```

A boolean returning function is correctly named when you call the function in code and can read that function call statement in plain English. Below is an example of incorrect and correct naming:

Figure 4.97

```
class Thread:
    # Incorrect naming
    def stopped(self) -> bool:
        # ...

    # Correct naming
    def is_stopped(self) -> bool:
        # ...

if thread.stopped():
    # Here we have: if thread stopped
```

```
    # This is not a statement with a true or false answer.
    # It is a second conditional form,
    # asking what would happen if thread stopped.

    # ...


# Here we have statement: if thread is stopped
# True or false?
if thread.is_stopped():
    # ...
```

From the above examples, we can notice that many names of boolean-returning functions start with either *is* or *has* and follows the below pattern:

- is + <adjective>, e.g. is_open, is_running or is_paused
- has + <noun>

Also, these two forms can be relatively common:

- should + <verb>
- can + <verb>

But as we saw with the `starts_with`, `ends_with`, and `contains` functions, a boolean returning function name can start with any verb in third-person singular form (i.e., ending with an *s*). If you have a collection class, its boolean method names should have a verb in the plural form, for example: `numbers.include(...)` instead of `numbers.includes(...)`. Name your collection variables always in plural form (e.g., `numbers` instead of `number_list)`. We will discuss the uniform naming principles for variables in the next chapter.

Do not include the *does* word in a function name, like *does_start_with*, *does_end_with*, or *does_contain*. Adding the *does* word doesn't add any real value to the name, and such function names are awkward to read when used in code, for example:

Figure 4.98

```
line = text_file_reader.read_line()

# "If line does start with" sounds awkward
if line.does_start_with(' '):
    # ...
```

When you want to use the past tense in a function name, use a *did* prefix in the function name, for example:

Figure 4.99

```
class DatabaseOperation:
    def execute(self) -> None:
        # ...

    # Method name not OK. This is a second conditional form
    # if db_operation.started_transaction(): ...
    def started_transaction(self) -> bool:
        # ...

    # Method name OK, no confusion possible
    def did_start_transaction(self) -> bool:
        # ...
```

4.9.2.4: Naming Builder Methods

A builder class is used to create builder objects that build a new object of a particular type. If you want to construct a URL, a *UrlBuilder* class can be used for that purpose. Builder class methods add properties to the built object. For this reason, it is recommended to name builder class methods starting with the verb *add*. The method that finally builds the wanted object should be named simply *build* or *build* + <*build-target*>, for example, `build_url`. I prefer the longer form to remind the reader what is being built. Below is an example of naming the methods in a builder class:

Figure 4.100

```
class UrlBuilder:
    def add_scheme(self, scheme: str) -> 'UrlBuilder':
        # ...
        return self

    def add_host(self, host: str) -> 'UrlBuilder':
        # ...
        return self

    def add_port(self, port: int) -> 'UrlBuilder':
        # ...
        return self

    def add_path(self, path: str) -> 'UrlBuilder':
        # ...
        return self

    def add_query(self, query: str) -> 'UrlBuilder':
        # ...
        return self
```

```
    def build_url(self) -> Url:
        # ...


url = (

UrlBuilder().add_scheme('https://').add_host('google.com').build
_url()
)
```

4.9.2.5: Naming Methods with Implicit Verbs

Factory method names usually start with the verb *create*. Factory methods can be named so that the *create* verb is implicit, for example:

Figure 4.101

```
Optional.of(value)
Optional.empty() # Not optimal, 'empty' can be confused as a
verb
Either.with_left(value)
Either.with_right(value)
SalesItem.from_dto(input_sales_item)
```

The explicit versions of the above method names would be:

Figure 4.102

```
Optional.create(value)
Optional.create_empty()
Either.create_with_left(value)
Either.create_with_right(value)
SalesItem.create_from_dto(input_sales_item)
```

Similarly, conversion methods can be named so that the *convert* verb is implicit. Conversion methods without a verb usually start with the *to* preposition, for example:

Figure 4.103

```
numeric_value.to_string()
dict_value.to_json()
```

The explicitly named versions of the above methods would be:

Figure 4.104

```
numeric_value.convert_to_string()
dict_value.convert_to_json()
```

I recommend using method names with implicit verbs sparingly and only in circumstances where the implicit verb is

self-evident and does not force a developer to think.

4.9.2.6: Naming Lifecycle Methods

Lifecycle methods are called on certain occasions only. Lifecycle method names should answer the question: When or "on what occasion" will this method be called? Examples of good names for lifecycle methods are: `on_init`, `on_error`, `on_success`, `after_mount`, `before_unmount`. For example, in React, there are lifecycle methods in class components called `componentDidMount`, `componentDidUpdate` and `componentWillUnmount`. There is no reason to repeat the class name in the lifecycle method names. Better names would have been: `afterMount`, `afterUpdate`, and `beforeUnmount`.

4.9.2.7: Naming Function Parameters

Naming rules for function parameters are mostly the same as for variables. *Uniform naming principle* for variables is described in the next chapter in more detail.

There are some exceptions, like naming object parameters. When a function parameter is an object, the name of the object class can be left out from the parameter name when the parameter name and the function name implicitly describe the class of the parameter. This exception is acceptable because the function parameter type can always be easily checked by looking at the function signature. This should be easily done with a glance because a function should be short (a maximum of 5-9 statements). Below is an example of naming object type parameters:

Figure 4.105

```
# Word 'Location' repeated, not optimal, but allowed
def drive(start_location: Location, destination_location:
Location) -> None:
    # ...

# Better way
# When we think about 'drive' and 'start' or 'destination',
# we can assume that 'start' and 'destination' mean locations
def drive(start: Location, destination: Location) -> None:
    # ...
```

Some programming languages like Swift allow adding so-called *external names* to function parameters. Using external names can make a function call statement read better, as shown below:

Figure 4.106

```
func drive(from start: Location, to destination: Location) {
  // ...
}

func send(
  message: String,
  from sender: Person,
  to recipient: Person
) {
  // ...
}

let start = new Location(...);
let destination = new Location(...);
drive(from: start, to: destination);

let message = "Some message";
let sender = new Person(...);
let recipient = new Person(...);
send(message, from: sender, to: recipient);
```

Always make the function call expression such that it has maximum readability: e.g., `copy(source, target)` not `copy(target, source)` or `write(line, file)` not `write(file, line)` or `decode(input_message, internal_message)` and `encode(internal_field, output)`. The examples contain implicit articles and prepositions. You can easily imagine the missing articles and prepositions, e.g., *copy from a source to a target*, *write a line to a file* or *encode an internal field to output*.

4.10: Encapsulation Principle

> ***A class should encapsulate its state so that access to the state happens only via public methods.***

Encapsulation is achieved by declaring class attributes private. Python does not have attribute access modifiers. You should use a naming convention:

Use an attribute name prefixed with __ (double underscore) to denote a private attribute and a _ prefix to denote a protected attribute.

You can create getter and setter methods (or properties) if you need the state to be modifiable outside the class. However, encapsulation is best ensured if you don't need to create getter and setter methods for the class attributes. Do not automatically implement or generate getter and setter methods for every class. Only create those accessor methods if needed, like when the class represents a modifiable data structure. An automatically generated getter can break the encapsulation of a class if the getter returns modifiable internal state, like a list. Only generate setter methods for attributes that need to be modified outside the class. If you have a class with many getters, you might be guilty of *feature envy* code smell, where other objects query your object for its internal state and perform operations based on that state. You should follow the *don't ask, tell principle* (discussed later in this chapter) by removing the getters from your class and implementing the operation in your class.

4.10.1: Immutable Objects

The best way to ensure the encapsulation of an object's state is to make the object immutable. This means that once the object is created, its state cannot be modified afterward. Immutability ensures you cannot accidentally or intentionally modify the object's state. Modifying the object's state outside the object can be a source of bugs.

When creating an immutable object, you give the needed parameters for the object in the constructor, and after that, those properties cannot be modified (You don't create any setters for the class). If you need to modify an immutable object, the only way is to create a new object with different values given in the constructor. The drawback of this approach is that a performance penalty is introduced when creating new objects as compared to modifying existing objects' attributes only. But in many cases, this penalty is negligible compared to

the benefits of immutability. For example, strings are immutable in Python. Once you create a string, you cannot modify it. You can only create new strings.

Immutability also requires that getters and other methods returning a value may not return a modifiable attribute, like a list. If you return a list from a method, that list could be modified by adding or removing elements without the "owning" object being aware of that.

4.10.2: Don't Leak Modifiable Internal State Outside an Object Principle

Beware when you return values from methods. It is possible that a method accidentally returns some internal state of the object that can be modified later by the method caller. Returning modifiable state from a method breaks the encapsulation.

You can safely return an object's internal state when it has a primitive or so-called value type. Those include bool, int, and float. You can also safely return an immutable object, like a string. But you cannot safely return a mutable collection, for example.

There are two ways to protect against leaking internal state outside an object:

1. Return a copy of the modifiable internal state
2. Return an unmodifiable version of the modifiable internal state

Regarding the first approach, when a copy is returned, the caller can use it as they like. Changes made to the copied object don't affect the original object. I am primarily talking about making a shallow copy. In many cases, a shallow copy is enough. For example, a list of primitive values, immutable strings, or immutable objects does not require a deep copy of the list. But you should make a deep copy when needed.

The copying approach can cause a performance penalty, but in many cases, that penalty is insignificant. You can easily create

a copy of a list:

Figure 4.107

```
values = [1, 2, 3, 4, 5]
values2 = values.copy()
```

The second approach requires you to create an unmodifiable version of a modifiable object and return that unmodifiable object. You can create an unmodifiable version of a class by yourself. Below is an example:

Figure 4.108

```
from typing import Protocol, TypeVar

T = TypeVar('T')


class MyList(Protocol[T]):
    def append(self, item: T) -> None:
        # ...

    def get_item(self, index: int) -> T | None:
        # ...


class UnmodifiableMyList(MyList[T]):
    def __init__(self, list_: MyList[T]):
        self.__list = list_

    def append(self, item: T) -> None:
        raise NotImplementedError()

    def get_item(self, index: int) -> T | None:
        return self.__list.get_item(index)
```

In the above example, the unmodifiable list class takes another list (a modifiable list) as a constructor argument. It only implements the `MyList` protocol methods that don't attempt to modify the wrapped list. In this case, it implements only the `get_item` method that delegates to the respective method in the `MyList` class. The methods of the `UnmodifiableMyList` class that attempt to modify the wrapped list should raise an error. The `UnmodifiableMyList` class utilizes the *proxy pattern* (discussed later in this chapter) by wrapping an object of the `MyList` class and partially allowing access to the `MyList` class methods.

Unmodifiable and immutable objects are slightly different. No one can modify an immutable object, but when you return an

unmodifiable object from a method, the owning class can still modify that object, and modifications are visible to everyone who has earlier received an unmodifiable version of the object. If this is something undesirable, you should use a copy instead.

4.10.3: Don't Assign From a Method Parameter to a Modifiable Attribute

If a class receives modifiable objects as constructor or method arguments, it is typically best practice not to directly assign those arguments to the internal state. If assigned directly, the class can purposely or accidentally modify those argument objects, which is probably not what the constructor or method caller expects.

There are two ways to handle this situation:

1. Store a copy of the modifiable argument object to the object's internal state
2. Store an unmodifiable version of the modifiable argument object to the object's internal state. Note that the original modifiable object's owner can modify the object, which is reflected in the unmodifiable object, too. If this is something unwanted, use a copy instead.

Below is an example of the second approach:

Figure 4.109

```
class MyClass:
    def __init__(self, values: MyList[int]):
        self.__values = UnmodifiableMyList(values)
```

4.11: Prefer Composition Over Inheritance Principle

> ***In object-oriented design, like in real life, objects are constructed by constructing larger objects from smaller objects. This is called object composition. Prefer object composition over inheritance.***

This principle is presented in the *Design Patterns* book by the *Gang of Four*. An example of composition is a car object composed of an engine and transmission object (to name a few). Objects are rarely "composed" by deriving from another object, i.e., using inheritance. But first, let's try to specify classes that implement the below `Car` protocol using inheritance and see where it leads us:

Figure 4.110

```
from typing import Protocol


class Car(Protocol):
    def drive(self, start: Location, destination: Location) ->
None:
        pass


class CombustionEngineCar(Car):
    def drive(self, start: Location, destination: Location) ->
None:
        # ...


class ElectricEngineCar(Car):
    def drive(self, start: Location, destination: Location) ->
None:
        # ...


class
ManualTransmissionCombustionEngineCar(CombustionEngineCar):
    def drive(self, start: Location, destination: Location) ->
None:
        # ...


class
AutomaticTransmissionCombustionEngineCar(CombustionEngineCar):
    def drive(self, start: Location, destination: Location) ->
None:
       # ...
```

If we wanted to add other components to a car, like a two or four-wheel drive, the number of classes needed would increase by three. If we wanted to add a design property (sedan, hatchback, wagon, or SUV) to a car, the number of needed classes would explode, and the class names would become ridiculously long, like `HatchbackFourWheelDriveAutomaticTransmissionCombustionEngineCar`. We can notice that inheritance is not the correct way to build more complex classes here.

Class inheritance creates an *is-a* relationship between a superclass and its subclasses. Object composition creates a *has-a* relationship. We can claim that `ManualTransmissionCombustionEngineCar` *is a* kind of `CombustionEngineCar`, so basically, we are not doing anything wrong here, one might think. But when designing classes, you should first determine if object composition could be used: is there a *has-a* relationship? Can you declare a class as an attribute of another class? If the answer is yes, then composition should be used instead of inheritance.

All the above things related to a car are actually properties of a car. A car *has an* engine. A car *has a* transmission. It *has a* two or four-wheel drive and design. We can turn the inheritance-based solution into a composition-based solution:

Figure 4.111

```
from typing import Protocol


class Drivable(Protocol):
    def drive(self, start: Location, destination: Location) ->
None:
        pass


class Engine(Protocol):
    # Methods like start, stop ...


class CombustionEngine(Engine):
    # Methods like start, stop ...


class ElectricEngine(Engine):
    # Methods like start, stop ...


class Transmission(Protocol):
    # Methods like shift_gear ...


class AutomaticTransmission(Transmission):
    # Methods like shift_gear ...


class ManualTransmission(Transmission):
    # Methods like shift_gear ...


# Define DriveType here...
# Define Design here...
```

```
class Car(Drivable):
    def __init__(
        self,
        engine: Engine,
        transmission: Transmission,
        drive_type: DriveType,
        design: Design
    ):
        self.__engine = engine
        self.__transmission = transmission
        self.__drive_type = drive_type
        self.__design = design

  def drive(self, start: Location, destination: Location) ->
None:
        # To implement functionality, delegate to
        # component classes, for example:

        # self.__engine.start()
        # self.__transmission.shift_gear(...)
        #  ...
        # self.__engine.stop()
```

Let's have a more realistic example with different chart types. At first, this sounds like a case where inheritance could be used: We have some abstract base charts that different concrete charts extend, for example:

Figure 4.112

```
from abc import abstractmethod
from typing import Protocol


class Chart(Protocol):
    def render_view(self) -> None:
        pass

    def update_data(self, ...) -> None:
        pass


class AbstractChart(Chart):
    @abstractmethod
    def render_view(self) -> None:
        pass

    @abstractmethod
    def update_data(self, ...) -> None:
        pass

    # Implement some common functionality
    # shared by all chart types


class AbstractXAxisChart(AbstractChart):
    @abstractmethod
    def render_view(self) -> None:
        pass
```

```
    def update_data(self, ...) -> None:
        # This is common for all x-axis charts,
        # like ColumnChart, LineChart and AreaChart


class ColumnChart(AbstractXAxisChart):
    def render_view(self) -> None:
        # Render column chart using library XYZ


# LineChart class definition here...
# AreaChart class definition here...


class AbstractNonAxisChart(AbstractChart):
    @abstractmethod
    def render_view(self) -> None:
        pass

    def update_data(self, ...) -> None:
        # This is common for all non-x-axis charts,
        # like PieChart and DonutChart


class PieChart(AbstractNonAxisChart):
    def render_view(self) -> None:
        # Render pie chart using library XYZ


class DonutChart(PieChart):
    def render_view(self) -> None:
        # Render donut chart using library XYZ
```

The above class hierarchy looks manageable: there should not be too many subclasses that need to be defined. We can, of course, think of new chart types, like a geographical map or data table for which we could add subclasses. One problem with a deep class hierarchy arises when you need to change or correct something related to a particular chart type. Let's say you want to change or correct some behavior related to a pie chart. You will first check the `PieChart` class to see if the behavior is defined there. If you can't find what you are looking for, navigate to the base class of the `PieChart` class (`AbstractNonAxisChart`) and look there. You might need to continue this navigation until you reach the base class where the behavior you want to change or correct is located. When you are incredibly familiar with the codebase, you might be able to locate the correct subclass on the first try. But in general, this is not a straightforward task and is a major drawback of deep class hierarchies.

Using class inheritance can introduce class hierarchies where some classes have significantly more methods than other classes. For example, in the chart inheritance chain, the `AbstractChart` class probably has significantly more methods than classes at the end of the inheritance chain. This class size difference creates an imbalance between classes, making it hard to reason about what functionality each class provides.

Even if the above class hierarchy looks okay at first sight, there is one problem. We have hardcoded what kind of chart view we are rendering. We use the *XYZ* chart library and render `XYZChart` views. Let's say we would like to introduce another chart library called *ABC*. We want to use both chart libraries in parallel so that the open-source version of our data visualization application uses the *XYZ* chart library, which is open source. The paid version of our application uses the commercial *ABC* chart library. When using class inheritance, we must create new classes for each concrete chart type for the *ABC* chart library. So, we would have two classes for each concrete chart type, like here for the pie chart:

Figure 4.113

```
class XyzPieChart(AbstractXyzNonAxisChart):
    def render_view(self) -> None:
        # Render pie chart using XYZ library


class AbcPieChart(AbstractAbcNonAxisChart):
    def render_view(self) -> None:
        # Render pie chart using ABC library
```

Implementing the above functionalities using composition instead of inheritance has several benefits:

- It is more apparent what behavior each class contains
- There is no significant size imbalance between classes, where some classes are huge and others relatively small
- You can split chart behaviors into classes as you find fit, and is in accordance with the *single responsibility principle*

In the below example, we have split some chart behavior into two types of classes: chart view renderers and chart data

factories:

Figure 4.114

```
from enum import Enum
from typing import Protocol


class Chart(Protocol):
    def render_view(self) -> None:
        pass

    def update_data(self, ...) -> None:
        pass


# Define ChartData class...
# Define ChartOptions class...


class ChartViewRenderer(Protocol):
    def render_view(self, data: ChartData, options:
ChartOptions) -> None:
        pass


class ChartDataFactory(Protocol):
    def create_data(self, ...) -> ChartData:
        pass


class ChartImpl(Chart):
    def __init__ (
        self,
        view_renderer: ChartViewRenderer,
        data_factory: ChartDataFactory,
        options: ChartOptions
    ):
        self.__view_renderer = view_renderer
        self.__data_factory = data_factory
        self.__options = options
        self.__data = None,

    def render_view(self) -> None:
        self.__view_renderer.render_view(self.__data,
self.__options)

    def update_data(self, ...) -> None:
        self.__data = self.__data_factory.create_data(...)


class XyzPieChartViewRenderer(ChartViewRenderer):
    def render_view(self, data: ChartData, options:
ChartOptions) -> None:
        # Render pie chart with XYZ library


class AbcPieChartViewRenderer(ChartViewRenderer):
    def render_view(self, data: ChartData, options:
ChartOptions) -> None:
        # Render pie chart with ABC library
```

```
# Define AbcColumnChartViewRenderer class...
# Define XyzColumnChartViewRenderer class...
# Define XAxisChartDataFactory class ...
# Define NonAxisChartDataFactory class ...


class ChartType(Enum):
    COLUMN = 1
    PIE = 2


class ChartFactory(Protocol):
    def create_chart(self, chart_type: ChartType) -> Chart:
        pass


class AbcChartFactory(ChartFactory):
    def create_chart(self, chart_type: ChartType) -> Chart:
        match chart_type:
            case ChartType.COLUMN:
                return ChartImpl(AbcColumnChartViewRenderer(),
                                 XAxisChartDataFactory())
            case ChartType.PIE:
                return ChartImpl(AbcPieChartViewRenderer(),
                                 NonAxisChartDataFactory())
            case _:
                raise ValueError('Invalid chart type')


class XyzChartFactory(ChartFactory):
    def create_chart(self, chart_type: ChartType) -> Chart:
        match chart_type:
            case ChartType.COLUMN:
                return ChartImpl(XyzColumnChartViewRenderer(),
                                 XAxisChartDataFactory())
            case ChartType.PIE:
                return ChartImpl(XyzPieChartViewRenderer(),
                                 NonAxisChartDataFactory())
            case _:
                raise ValueError('Invalid chart type')
```

The `XyzPieChartViewRenderer` and `AbcPieChartViewRenderer` classes use the *adapter pattern* as they convert the supplied data and options to an implementation-specific (ABC or XYZ chart library) interface.

We can easily add more functionality by composing the `ChartImpl` class of more classes. There could be, for example, a title formatter, tooltip formatter class, y/x-axis label formatter, and event handler classes.

Figure 4.115

```
class ChartImpl(Chart):
    def __init__ (
        self,
```

```
        view_renderer: ChartViewRenderer,
        data_factory: ChartDataFactory,
        title_formatter: ChartTitleFormatter,
        tooltip_formatter: ChartTooltipFormatter,
        x_axis_label_formatter: ChartXAxisLabelFormatter,
        event_handler: ChartEventHandler,
        options: ChartOptions
    ):
        # ...

    # Chart methods ...


class AbcChartFactory(ChartFactory):
    def create_chart(self, chart_type: ChartType) -> Chart:
        match chart_type:
            case ChartType.COLUMN:
                return ChartImpl(AbcColumnChartViewRenderer(),
                                 XAxisChartDataFactory(),
                                 ChartTitleFormatterImpl(),
                                 XAxisChartTooltipFormatter(),
                                 ChartXAxisLabelFormatterImpl(),
                                 ColumnChartEventHandler())

            case ChartType.PIE:
                return ChartImpl(AbcColumnChartViewRenderer(),
                                 NonAxisChartDataFactory(),
                                 ChartTitleFormatterImpl(),
                                 NonAxisChartTooltipFormatter(),
                                 NullXAxisLabelFormatter(),
                                 NonAxisChartEventHandler())

            case _:
                raise ValueError('Invalid chart type')
```

4.12: Tactical Domain-Driven Design Principle

> ***Domain-driven design (DDD) is a software design approach where software is modeled to match the language of the problem domain that the software tries to solve. DDD is hierarchical. The top-level domain can be divided into subdomains, which can be further divided into subdomains.***

We continue here where we left with strategical DDD in the last chapter. Strategic DDD was about dividing a software system to subdomains and bounded contexts (microservices). Tactical DDD is about implementing a single bounded context. Tactical DDD means that the structure of a bounded context

and the names appearing in the code (interface, class, function, and variable names) should match the domain's vocabulary and the ubiquitous language. For example, names like *Account*, *withdraw*, *deposit*, *make_payment* should be used in a *payment-service* software component.

4.12.1: Tactical DDD Concepts

Tactical domain-driven design recognizes multiple concepts:

- Entities
- Value Objects
- Aggregates
- Aggregate Roots
- Factories
- Repositories
- Services
- Events

4.12.1.1: Entities

An entity is a domain object that has an identity. Usually, this is indicated by the entity class having some *id* attribute. Examples of entities are an *employee* and a *bank account*. An employee object has an employee id, and a bank account has a number that identifies the bank account. Entities can contain methods that operate on the attributes of the entity. For example, a bank account entity can have methods *withdraw* and *deposit* that operate on the *balance* attribute of the entity.

4.12.1.2: Value Objects

Value objects are domain objects that don't have an identity. Examples of value objects are an address or a price object. The price object can have two attributes: *amount* and *currency*, but it does not have an identity. Similarly an address object can

have the following attributes: *street address*, *postal code*, *city* and *country*.

4.12.1.3: Aggregates

Aggregates are entities composed of other entities and value objects. For example, an *order* entity can have one or more *order item* entities. Regarding object-oriented design, this is the same as object composition. Each aggregate has a root (entity). The figure below shows a `SalesItem` aggregate. A `SalesItem` entity is an aggregate and aggregate root. It can contain one or more images of the sales item, and it consists of a `Price` value object, which has two attributes: price and currency.

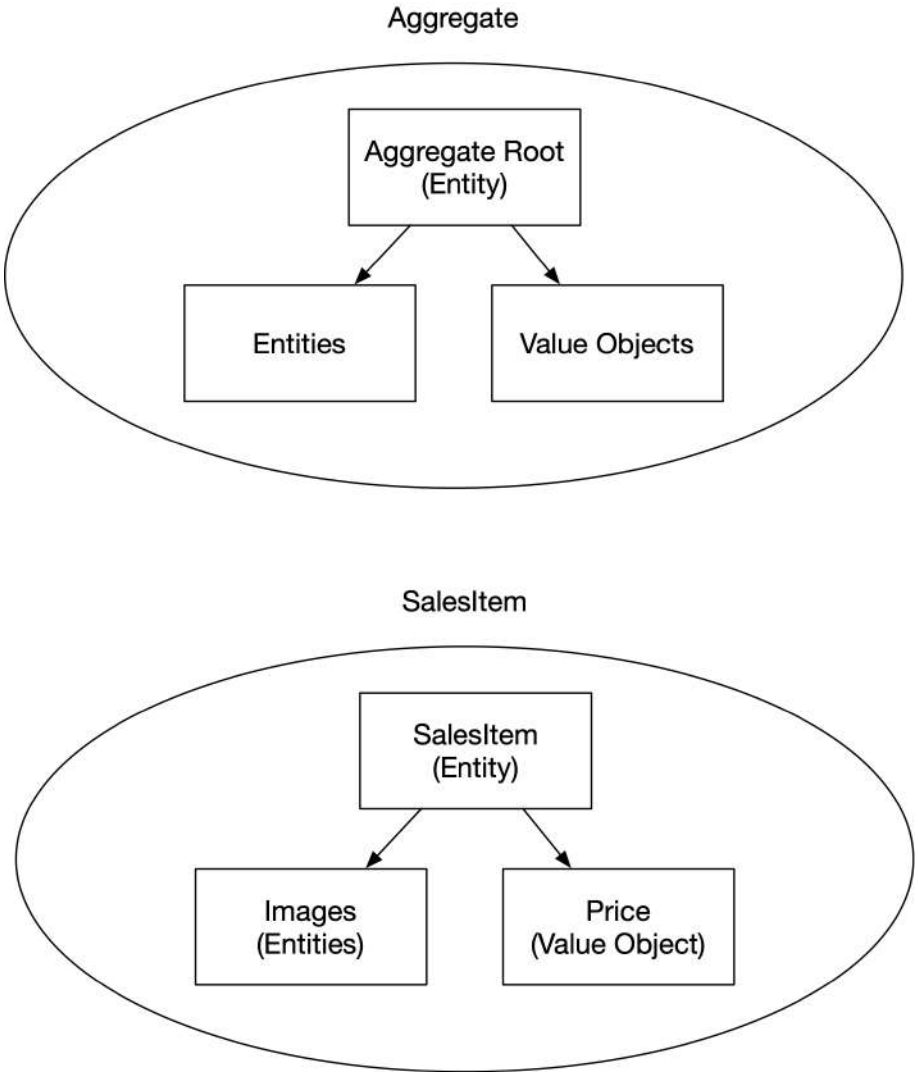

Figure 4.116. Aggregate Example

4.12.1.4: Aggregate Roots

Aggregate roots are domain objects that don't have any parent objects. An *order* entity is an aggregate root when it does not have a parent entity. But an *order item* entity is not an aggregate root when it belongs to an *order*. Aggregate roots serve as facade objects. Operations should be performed on the aggregate root objects, not directly accessing the objects behind the facade (e.g., not directly accessing the individual order items, but performing operations on order objects). For example, if you have an aggregate car object containing wheels, you don't operate the wheels outside of the car object.

The car object provides a facade like a *turn* method, and the car object internally operates the wheels, making the car object an aggregate root. More about the *facade pattern* in a later section of this chapter.

Let's have an example of aggregate roots in a microservice architecture. Suppose we have a bank account, an aggregate root containing transaction entities. The bank account and transaction entities can be handled in different microservices (*account-service* and *account-transaction-service*), but only the *account-service* can directly access and modify the transaction entities using the *account-transaction-service*. Our bounded context is the *account-service*. The role and benefit of an aggregate root are the following:

- The aggregate root protects against invariant violation. For example, no other service should directly remove or add transactions using the *account-transaction-service*. That would break the invariant that the sum of transactions should be the same as the balance of the account maintained by the *account-service*.
- The aggregate root simplifies (database/distributed) transactions. Your microservice can access the *account-service* and let it manage the distributed transactions between the *account-service* and *account-transaction-service*. It's not something that your microservice needs to do.

You can easily split an aggregate root into more entities. For example, the bank account aggregate root could contain balance and transaction entities. The balance entity could be handled by a separate *account-balance-service*. Still, all bank account operations must be made to the *bank-account-service*, which will orchestrate, e.g., *withdraw* and *deposit* operations using the *account-balance-service* and *account-transaction-service*. We can even split the *account-service* to two separate microservices: *account-service* for account CRUD operations (excluding updates related to balance) and *account-money-transfer-service* that will handle *withdraw* and *deposit* operations using the two lower-level microservices: *account-*

balance-service and *account-transaction-service*. In the previous chapter, we had an example of the latter case when we discussed distributed transactions.

4.12.2: Actors

Actors perform commands. End-users are actors, but also services can be actors. For example, in a data exporter microservice, there can be an input message consumer service that has a command to consume a message from a data source.

4.12.2.1: Factories

In domain-driven design, the creation of domain objects can be separated from the object classes to factories. Factories are objects that are dedicated to creating objects of a particular type. More about the *factory pattern* in a later section of this chapter.

4.12.2.2: Repositories

A repository is an object with methods for persisting domain objects and retrieving them from a data store (e.g., a database). Typically, there is one repository for each aggregate root, e.g., an *order repository* for order entities.

4.12.2.3: Services

Services are used to implement business use cases and contain functionality that is not directly part of any specific object. Services orchestrate operations on aggregate roots. For example, `OrderService* orchestrates operations on order entities. A service typically uses a related repository to perform persistence-related operations. A service can also be seen as an actor with specific command(s). For example, in a data exporter microservice, there can be an input message consumer service that has a command to consume a message from a data source.

4.12.2.4: Events

Events are operations on entities and form the business use cases. Services usually handle events. For example, there could be the following events related to order entities: create, update, and cancel an order. These events can be implemented by having an `OrderService` with the following methods: `create_order`, `update_order`, and `cancel_order`.

4.12.2.5: Event Storming

Eevent storming is a lightweight method a team can use to discover DDD-related concepts in a software component. The event storming process typically follows the below steps:

1. Figure out *domain events* (events are usually written in past tense)
2. Figure out *commands* that caused the *domain events*
3. Add *actors*/*services* that execute the *commands*
4. Figure out related *entities*

In event storming, the different DDD concepts like events, commands, actors, and entities are represented with sticky notes in different colors. These sticky notes are put on a wall, and related sticky notes are grouped together, like actor(s), command(s), and entity/entities for a specific domain event. If you are interested in details of the event storming process, there is a book *Introducing EventStorming* by *Alberto Brandolini*.

4.12.3: Domain-Driven Design Example: Data Exporter Microservice

Let's have a DDD example with a microservice for exporting data. Data exporting will be our top-level domain. The development team should participate in the DDD and object-oriented design (OOD) process. An expert-level software developer, e.g., the team tech lead, could do the DDD and

OOD alone, but it is not how it should be done. Other team members, especially the junior ones, should be involved to learn and develop their skills further.

The DDD process is started by first defining the big picture (top-level domain) based on requirements from the product management and the architecture team:

> Data exporter handles data that consists of messages that contain multiple fields. Data exporting should happen from an input system to an output system. During the export, various transformations to the data can be made, and the data formats in the input and output systems can differ.

Let's start the event-storming process by figuring out the domain events:

1. A message is consumed from the input system
2. The input message is decoded into an internal representation (i.e., an internal message)
3. The internal message is transformed
4. The transformed message is encoded to the wanted output format
5. The transformed message is produced in the output system
6. Configuration is read and parsed

From the above events, we can figure out four subdomains:

- Input (Events 1, 2 and 6)
- Internal Message (Events 2 and 3)
- Transform (Events 3 and 6)
- Output (Events 4, 5 and 6)

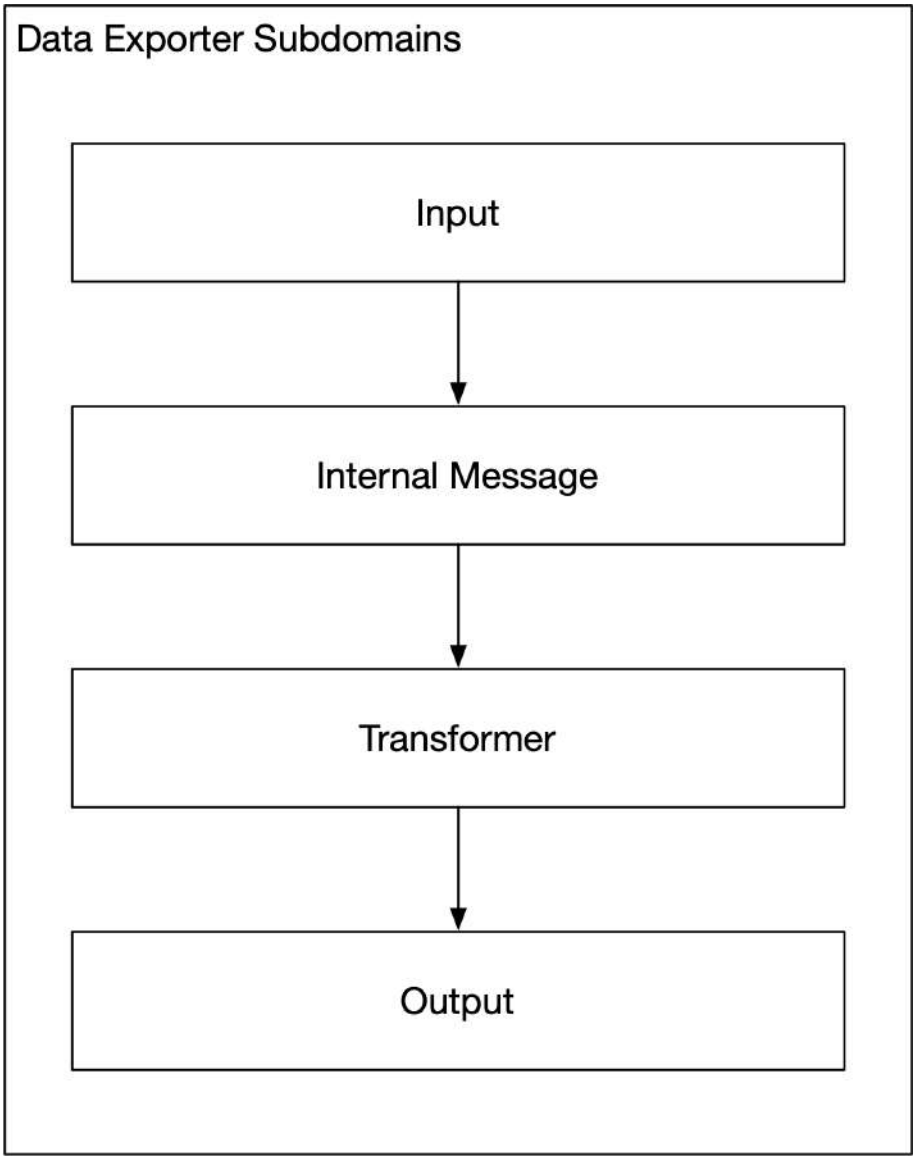

Figure 4.117. Data Exporter Subdomains

Let's take the first domain event, "Messages are consumed from the input system," and figure out what caused the event and who was the actor. Because no end-user is involved, we can conclude that the event was caused by an "input message consumer" *service* executing a "consume message" *command*. This operation creates an "input message" *entity*. The picture below shows how this would look with sticky notes on the wall.

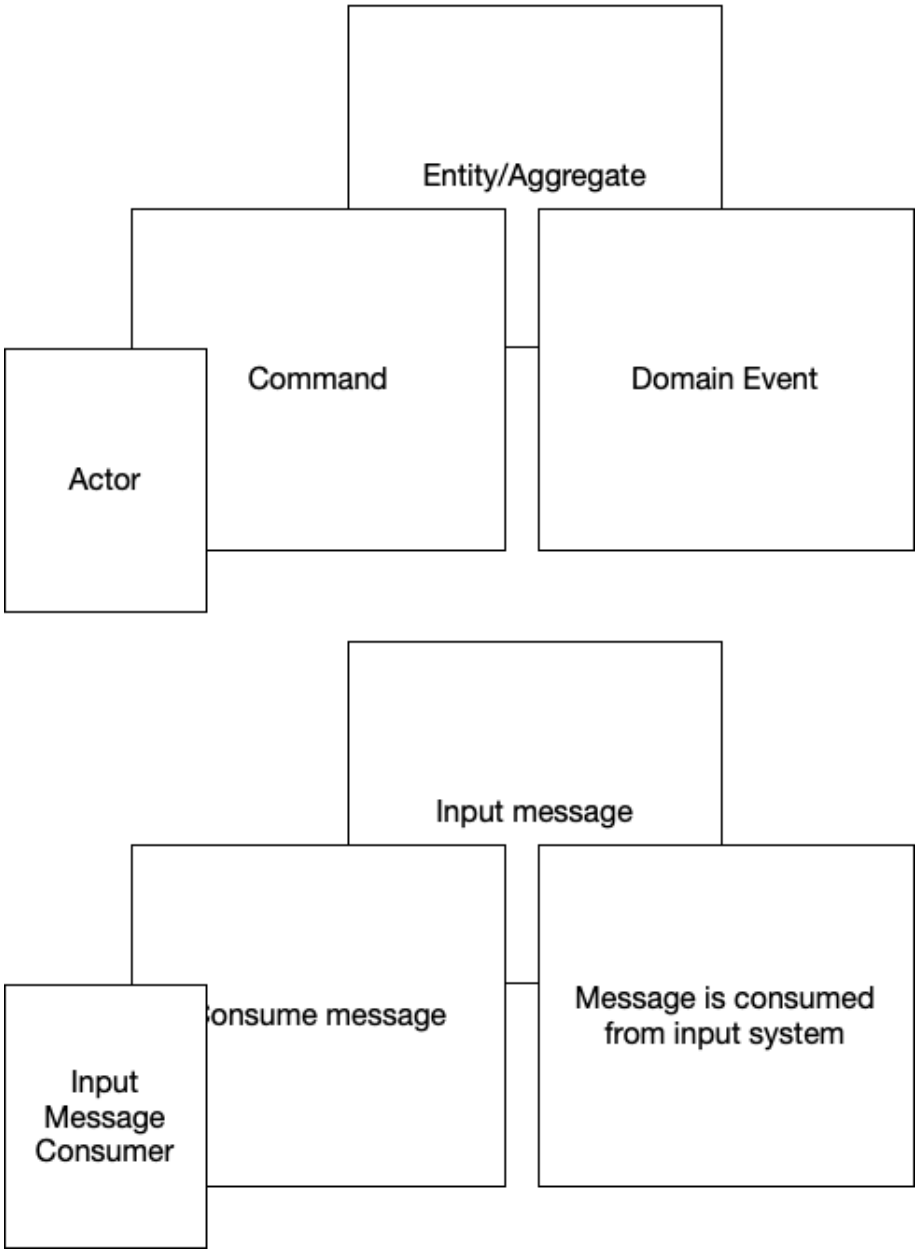

Figure 4.118. Event Storming

When continuing the event storming process further for the *Input* domain, we can figure out that it consists of the following additional DDD concepts:

- Commands
    - Read input configuration
    - Parse input configuration
    - Consume input message

    - Decode input message
- Actors/Services
    - Input configuration reader
    - Input configuration parser
    - Input message consumer
    - Input message decoder
- Entities
    - Input message
- Value Objects
    - Input configuration

The event-storming process that resulted in the above list of DDD concepts is actually object-oriented analysis (OOA). We have an initial set of objects that our use case needs when implemented. The actors/services are singleton objects. Entities and value objects are objects. Commands are the main methods in the actor/service classes. The OOA phase should result in an initial class diagram showing the main classes and their relationships with other classes.

Below is the list of sub-domains, interfaces, and classes in the *Input* domain:

- Input message
    - Contains the message consumed from the input data source
    - `InputMessage` is a protocol that can have several concrete implementations, like `KafkaInputMessage` representing an input message consumed from a Kafka data source
- Input message consumer
    - Consumes messages from the input data source and creates `InputMessage` instances
    - `InputMessageConsumer` is a protocol that can have several concrete implementations, like

`KafkaInputMessageConsumer` for consuming messages from a Kafka data source

- Input Message decoder
  - Decodes input messages into internal messages
  - `InputMessageDecoder` is a protocol that can have several concrete implementations, like `AvroBinaryInputMessageDecoder`, which decodes input messages encoded in Avro binary format. (If you provide multiple implementations for the `InputMessageDecoder` protocol, you must also create a factory class that can create different kinds of input message decoders).
- Input configuration
  - Input configuration reader
    - Reads the domain's configuration
    - `InputConfigReader` is a protocol that can have several concrete implementations, like `LocalFileSystemInputConfigReader` or `HttpRemoteInputConfigReader`
  - Input configuration parser
    - Parses the read configuration to produce an `InputConfig` instance
    - `InputConfigParser` is a protocol that can have several concrete implementations, like `JsonInputConfigParser` or `YamlInputConfigParser`
  - `InputConfig` instance contains parsed configuration for the domain, like the input data source type, host, port, and input data format.

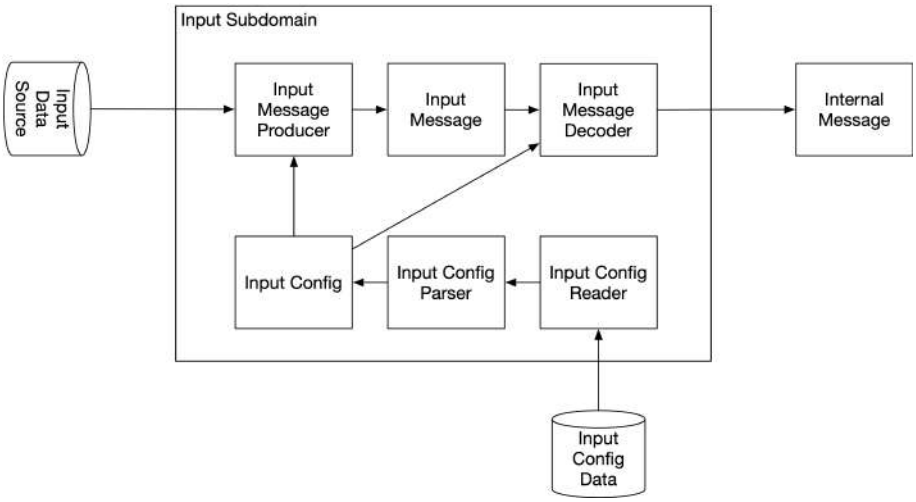

Figure 4.119. Input Subdomain

Next, we should perform object-oriented design (OOD) and design objects in a more detailed way, using various design principles and patterns. As shown in the below class diagram, we have applied the *dependency inversion / program against interfaces principle* to the result of the earlier OOA phase:

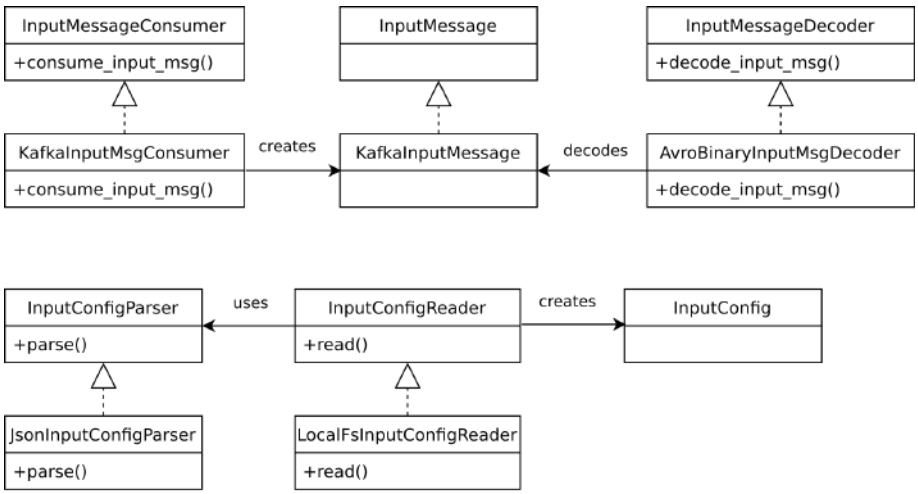

Figure 4.120. Input Subdomain Class Diagram

When applying the event storming process to the *Internal Message* domain, we can figure out that it consists of the following DDD concepts:

- Entities
    - Internal message

    - Internal field
- Aggregate
    - Internal message (consists of fields)
- Aggregate root
    - Internal message

Below is the list of sub-domains, interfaces, and classes in the *Internal Message* domain:

- Internal Message
    - Internal message consists of one or more internal fields
    - `InternalMessage` is an interface for a class that provides an internal representation of an input message
- Internal Field
    - `InternalField` is an interface for classes representing a single field of an internal message

When applying the event storming process to the *Transform* domain, we can figure out that it consists of the following DDD concepts:

- Commands
    - Read transformer configuration
    - Parse transformer configuration
    - Transform message
    - Transform field
- Actors/Services
    - Transformer configuration reader
    - Transformer configuration parser
    - Message transformer
    - Field transformer

- Value objects
    - Transformer configuration

Below is the list of sub-domains, interfaces, and classes in the *Transform* domain:

- Field transformer
    - `FieldTransformers` is a collection of `FieldTransformer` objects
    - A Field transformer transforms the value of an internal field into the value of an output message field
    - `FieldTransformer` is a protocol that can have several concrete implementations, like `FilterFieldTransformer`, `CopyFieldTransformer`, `TypeConversionFieldTransformer` and `ExpressionTransformer`
- Message Transformer
    - `MessageTransformer` takes an internal message and transforms it using field transformers
- Transformer configuration
    - Transformer configuration reader
        - Reads the domain's configuration
        - `TransformerConfigReader` is a protocol that can have several concrete implementations, like `LocalFileSystemTransformerConfigReader`
    - Transformer configuration parser
        - Parses read configuration to produce a `TransformerConfig` instance
        - `TransformerConfigParser` is a protocol that can have several concrete implementations, like `JsonTransformerConfigParser`

- `TransformerConfig` instance contains parsed configuration for the *Transformer* domain

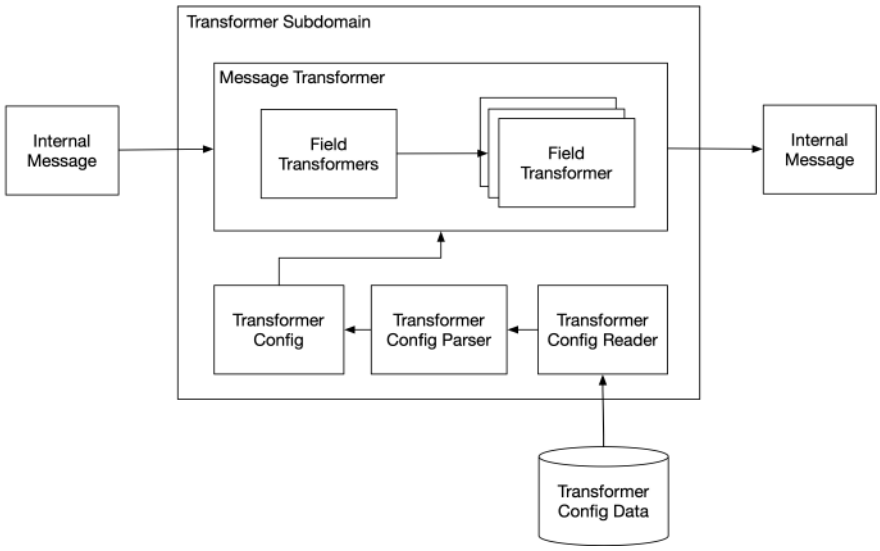

Figure 4.121. Transform Subdomain

Below is the class diagram for the Transform subdomain. I have left the configuration part out of the diagram because it is pretty much the same as the configuration part in the Input domain.

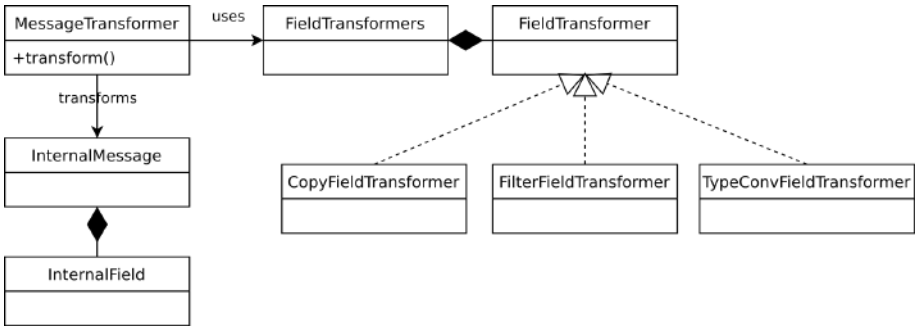

Figure 4.122. Transform Subdomain Class Diagram

When applying the event storming process to the *Output* domain, we can figure out that it consists of the following DDD concepts:

- Commands
  - Read output configuration
  - Parse output configuration

    - Encode output message
    - Produce output message
- Actors/Services
    - Output configuration reader
    - Output configuration parser
    - Output message encoder
    - Output message producer
- Entities
    - Output message
- Value objects
    - Output configuration

Below is the list of sub-domains, interfaces, and classes in the *Output* domain:

- Output Message encoder
    - Encodes transformed message to an output message with a specific data format
    - `OutputMessageEncoder` is a protocol that can have several concrete implementations, like `CsvOutputMessageEncoder`, `JsonOutputMessageEncoder`, `AvroBinaryOutputMessageEncoder`
- Output message
    - `OutputMessage` is a container for an output byte sequence
- Output message producer
    - Produces output messages to the output destination
    - `OutputMessageProducer` is a protocol that can have several concrete implementations, like `KafkaOutputMessageProducer`
- Output configuration

- Output configuration reader
  - Reads the domain's configuration
  - `OutputConfigReader` is a protocol that can have several concrete implementations, like `LocalFileSystemOutputConfigReader`
- Output configuration parser
  - Parse the read configuration to an `OutputConfig` instance
  - `OutputConfigParser` is a protocol that can have several concrete implementations, like `JsonOutputConfigParser`
- `OutputConfig` instance contains parsed configuration for the domain, like output destination type, host, port, and the output data format

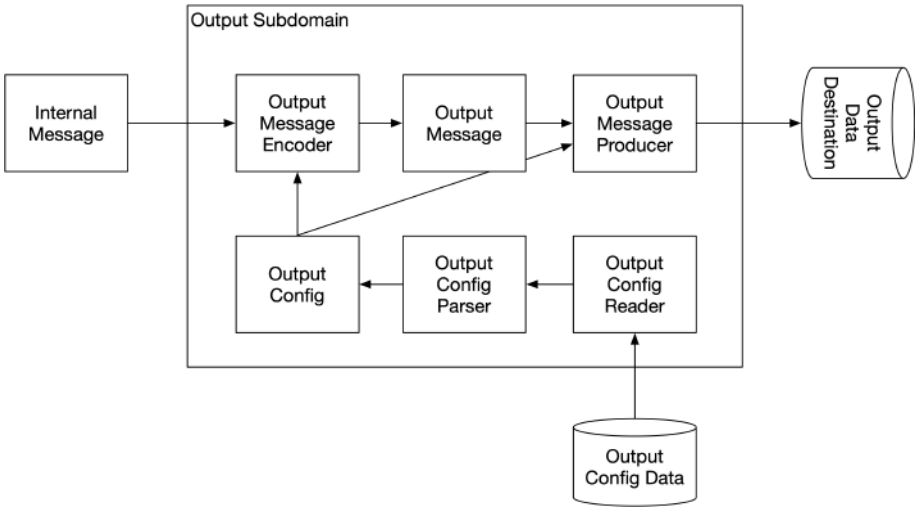

Figure 4.123. Output Subdomain

Below is the class diagram for the Output subdomain. I have left the configuration part out of the diagram because it is pretty much the same as the configuration part in the Input domain.

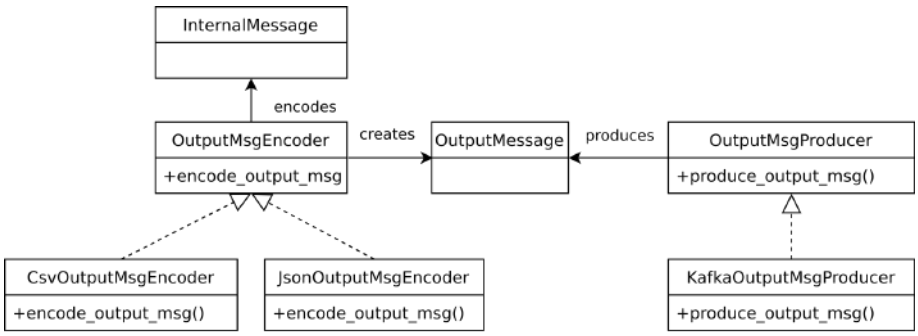

Figure 4.124. Output Subdomain Class Diagram

The above design also follows the *clean microservice design* principle. Note that this principle applies to all kinds of microservices with input or output, not just APIs. From the above design, we can discover the following interface adapters that are not part of the business logic of the microservice:

- `InputMessageConsumer` interface implementations
- `InputMessageDecoder` interface implementations
- `OutputMessageEncoder` interface implementations
- `OutputMessageProducer` interface implementations
- `InputConfigReader` interface implementations
- `InputConfigParser` interface implementations
- `TransformerConfigReader` interface implementations
- `TransformerConfigParser` interface implementations
- `OutputConfigReader` interface implementations
- `OutputConfigParser` interface implementations

We should be able to modify the implementations mentioned above or add a new implementation without modifying other parts of the code (the core or business logic). This means that we can easily adapt our microservice to consume data from different data sources in different data formats and output the transformed data to different data sources in various data formats. Additionally, the configuration of our microservice can be read from various sources in different formats. For example, if we now read some configuration from a local file

in JSON format, in the future, we could introduce new classes for reading the configuration from an API using some other data format.

After defining the interfaces between the above-defined subdomains, the four subdomains can be developed very much in parallel. This can speed up the microservice development significantly. The code of each subdomain should be put into separate source code folders. We will discuss source code organization more in the next chapter.

Based on the above design, the following data processing pipeline can be implemented:

Figure 4.125

```
class DataExporterApp:
    def run(self) -> None:
        while self.__is_running:
            input_message = (

self.__input_msg_consumer.consume_input_message()
            )

            internal_message = self.__input_msg_decoder.decode(
                input_message
            )

            transformed_message =
self.__msg_transformer.transform(
                internal_message
            )

            output_message = self.__output_msg_encoder.encode(
                transformed_message
            )

            self.__output_msg_producer.produce(output_message)
```

The `transform` method of the `MessageTransformer` class can be implemented in the following way:

Figure 4.126

```
class MessageTransformer:
    def transform(
        self, internal_message: InternalMessage
    ) -> InternalMessage:
        transformed_message = InternalMessage()

        for field_transformer in self.__field_transformers:
            field_transformer.transform_field(
                internal_message, transformed_message
            )

        return transformed_message
```

4.12.4: Domain-Driven Design Example 2: Anomaly Detection Microservice

Let's have another DDD example with an anomaly detection microservice. The purpose of the microservice is to detect anomalies in measurement data. This concise description of the microservice's purpose reveals the two subdomains of the microservice:

- Anomaly
- Measurement

Let's first analyze the *Measurement* subdomain in more detail and define domain events for it:

- Measurement data source definitions are loaded
- Measurement data source definitions are parsed
- Measurement definitions are loaded
- Measurement definitions are parsed
- Measurement data is fetched from data sources
- Measurement data is scaled (for further AI processing)

Let's continue using the event storming and define additional DDD concepts:

- Commands
    - Load measurement data source definitions
    - Parse measurement data source definitions
    - Load measurement definitions
    - Parse measurement definitions
    - Fetch measurement data from data sources
    - Scale measurement data -Actors/Services
    - Measurement data source definitions loader

        - Measurement data source definitions parser
        - Measurement definitions loader
        - Measurement definitions parser
        - Measurement data fetcher
        - Measurement data scaler
    - Entities
        - Measurement data source
        - Measurement data
        - Measurement
    - Aggregates
        - Measurement
            - Measurement data source
            - Measurement query
            - Measurement data
    - Aggregate root
        - Measurement
    - Value Objects
        - Measurement query

Let's define domain events for the *Anomaly* subdomain:

- Anomaly detection configuration is parsed
- Anomaly detection configuration is created
- Anomaly detection rule is parsed
- Anomaly detection rule is created
- Anomalies are detected in a measurement according to the anomaly detection rule using a trained anomaly model
- Anomaly detection is triggered at regular intervals
- Anomaly model is trained for a measurement
- Anomaly model is created

- Anomaly model training is triggered at regular intervals
- A detected anomaly (i.e., an anomaly indicator) is created
- A detected anomaly (i.e., an anomaly indicator) is serialized to a wanted format, e.g., JSON
- The detected anomaly (i.e., an anomaly indicator) is published to a specific destination using a specific protocol

Let's continue with the event storming and define additional DDD concepts:

- Commands
  - Parse anomaly detection configuration
  - Create anomaly detection configuration
  - Parse anomaly detection rule definition
  - Create anomaly detection rule
  - Detect anomalies in a measurement according to the anomaly detection rule using a trained anomaly model
  - Trigger anomaly detection at regular intervals
  - Train anomaly model for a measurement using a specific AI technique, like self-organizing maps (SOM)
  - Create an anomaly model
  - Trigger anomaly model training at regular intervals
  - Create anomaly indicator
  - Serialize anomaly indicator
  - Publish anomaly indicator
- Actors/Services
  - Anomaly detection configuration parser
  - Anomaly detection rule parser
  - Anomaly detector

    - Anomaly detection engine
    - Anomaly model trainer (e.g. SOM)
    - Anomaly training engine
    - Anomaly indicator serializer (e.g. JSON)
    - Anomaly indicator publisher (e.g., REST or Kafka)
- Factories
    - Anomaly detection configuration factory
    - Anomaly detection rule factory
    - Anomaly model factory
    - Anomaly indicator factory
- Entities
    - Anomaly detection rule
    - Anomaly model
    - Anomaly indicator

The two domains, anomaly and measurement, can be developed in parallel. The anomaly domain interfaces with the measurement domain to fetch data for a particular measurement from a particular data source. The development effort of both the anomaly and measurement domains can be further split to achieve even more development parallelization. For example, one developer could work with anomaly detection, another with anomaly model training, and the third with anomaly indicators.

If you want to know more about DDD, a short read is available: *Domain-Driven Design Distilled* by *Vaughn Vernon*.

4.13: Design Patterns

The following sections present 25 design patterns, most of which are made famous by the *Gang of Four* and their book Design Patterns. Design patterns are divided into creational, structural, and behavioral patterns.

4.13.1: Design Patterns for Creating Objects

This section describes design patterns for creating objects. The following design patterns will be presented:

- Factory pattern
- Abstract factory pattern
- Static factory method pattern
- Builder pattern
- Singleton pattern
- Prototype pattern
- Object pool pattern

4.13.1.1: Factory Pattern

> ***Factory pattern allows deferring what kind of object will be created to the point of calling the create-method of the factory***.

A factory allows a dynamic way of creating objects instead of a static way by directly calling a concrete class constructor. A factory typically consists of precisely one or multiple methods for creating objects of a particular base type. This base type is usually an interface type. The factory decides what concrete type of object will be created. A factory separates the logic of creating objects from the objects themselves, which is in accordance with the *single responsibility principle*.

Below is an example `ConfigParserFactory` that has a single `create` method for creating different kinds of `ConfigParser` objects. The return type of the factory's create method is an interface. This allows different kinds of objects in the class hierarchy to be created. In the case of a factory with a single create method, the method usually contains a match-case statement or an if/elif structure. Factories are the only place where extensive match-case statements or if/elif structures are allowed in object-oriented programming. If you have a lengthy

match-case statement or long if/elif structure somewhere else in the code, that is typically a sign of a non-object-oriented design.

Figure 4.127

```
from enum import Enum
from typing import Protocol


class ConfigParser(Protocol):
    # ...


class JsonConfigParser(ConfigParser):
    # ...


class YamlConfigParser(ConfigParser):
    # ...


class ConfigFormat(Enum):
    JSON = 1
    YAML = 2


class ConfigParserFactory:
    @staticmethod
    def create_config_parser(config_format: ConfigFormat) ->
ConfigParser:
        match config_format:
            case ConfigFormat.JSON:
                return JsonConfigParser()
            case ConfigFormat.YAML:
                return YamlConfigParser()
            case _:
                raise ValueError('Unsupported config format')
```

Below is an example of a factory with multiple *create* methods:

Figure 4.128

```
class ShapeFactory:
    @staticmethod
    def create_circle_shape(radius: int) -> Shape:
        return CircleShape(radius)

    @staticmethod
    def create_rectangle_shape(width: int, height: int) ->
Shape:
        return RectangleShape(width, height)

    @staticmethod
    def create_square_shape(side_length: int) -> Shape:
        return SquareShape(side_length)
```

4.13.1.2: Abstract Factory Pattern

In the abstract factory pattern, there is an abstract factory (interface) and one or more concrete factories (classes that implement the factory interface).

The abstract factory pattern extends the earlier described *factory pattern*. Usually, the abstract factory pattern should be used instead of the plain factory pattern. Below is an example of an abstract `ConfigParserFactory` with one concrete implementation:

Figure 4.129

```
class ConfigParserFactory(Protocol):
    def create_config_parser(self, config_format: ConfigFormat)
-> ConfigParser:
        pass


class ConfigParserFactoryImpl(ConfigParserFactory):
    def create_config_parser(self, config_format: ConfigFormat)
-> ConfigParser:
        match config_format:
            case ConfigFormat.JSON:
                return JsonConfigParser()
            case ConfigFormat.YAML:
                return YamlConfigParser()
            case _:
                raise ValueError('Unsupported config format')
```

You should follow the *program against interfaces principle* and use the abstract `ConfigParserFactory` in your code instead of a concrete factory. Then, using the *dependency injection principle*, you can inject the wanted factory implementation, like `ConfigParserFactoryImpl`.

When unit testing code, you should create mock objects instead of real ones with a factory. The abstract factory pattern comes to your help because you can supply a mock instance of the `ConfigParserFactory` in the tested code. Then, you can expect the mocked `create_config_parser` method to be called and return a mock instance conforming to the `ConfigParser` protocol. Then, you can expect the `parse` method to be called on the `ConfigParser` mock and return a mocked configuration. Below is an example unit test. We test the `initialize` method in the `Application` class containing a

`ConfigParserFactory` type attribute. The `Application` class uses the `ConfigParserFactory` instance to create a `ConfigParser` object to parse the application configuration.

Figure 4.130

```
from typing import Protocol


class Config(Protocol):
    # ...


class ConfigParser(Protocol):
    def parse(self) -> Config:
        pass

    # ...


class ConfigParserFactory(Protocol):
    def create_config_parser(self) -> ConfigParser:
        pass

    # ...


class Application:
    def __init__(self, config_parser_factory:
ConfigParserFactory):
        self.__config_parser_factory = config_parser_factory
        self.__config: Config | None = None

    def initialize(self) -> None:
        # ...
        config_parser =
self.__config_parser_factory.create_config_parser(...)
        self.__config = config_parser.parse(...)
        # ...

    @property
    def config(self):
        return self.__config
```

Here is the `ApplicationTests` class:

Figure 4.131

```
from unittest import main, TestCase
from unittest.mock import Mock

from Application import Application
from Config import Config
from ConfigParser import ConfigParser
from ConfigParserFactory import ConfigParserFactory


class ConfigParserFactoryMock(ConfigParserFactory):
     pass
```

```
class ConfigParserMock(ConfigParser):
    pass


class ConfigMock(Config):
    pass


class ApplicationTests(TestCase):
    def test_initialize(self):
        # GIVEN
        config_parser_factory_mock = ConfigParserFactoryMock()
        config_parser_mock = ConfigParserMock()

        config_parser_factory_mock.create_config_parser = Mock(
            return_value=config_parser_mock
        )

        config_mock = ConfigMock()
        config_parser_mock.parse =
Mock(return_value=config_mock)
        application = Application(config_parser_factory_mock)

        # WHEN
        application.initialize()

        # THEN
        self.assertEqual(application.config, config_mock)


if __name__ == '__main__':
    main()
```

In the above example, we created mocks manually. The `Mock` constructor creates a mock object that is also callable (i.e., it is also a mock function). You can supply the return value the mock should return when it is called in the `Mock` constructor. It is also possible to create mocks automatically using the `@patch` decorator, which can make code slightly less verbose, as shown below:

Figure 4.132

```
from unittest import TestCase, main
from unittest.mock import Mock, patch

from Application import Application
from Config import Config
from ConfigParser import ConfigParser
from ConfigParserFactory import ConfigParserFactory


class ApplicationTests(TestCase):
    @patch.object(ConfigParserFactory, '__new__')
    @patch.object(ConfigParser, '__new__')
    @patch.object(Config, '__new__')
    def test_initialize(
```

```
        self,
        config_mock: Mock,
        config_parser_mock: Mock,
        config_parser_factory_mock: Mock,
    ):
        # GIVEN

config_parser_factory_mock.create_config_parser.return_value = (
            config_parser_mock
        )

        config_parser_mock.parse.return_value = config_mock
        application = Application(config_parser_factory_mock)

        # WHEN
        application.initialize()

        # THEN
        self.assertEqual(application.config, config_mock)


if __name__ == '__main__':
    main()
```

Unit testing and mocking are better described later in the *testing principles* chapter.

4.13.1.3: Static Factory Method Pattern

> ***In the static factory method pattern, objects are created using one or more factory methods in a class, and the class constructor is made private. The factory methods are usually class methods.***

Static factory methods are useful if you need more than one constructor with different arguments because in Python a class can have only one constructor, the `__init__` methods.

If you want to validate the parameters supplied to a constructor, the constructor may raise an error. You cannot return an error value from a constructor. Creating constructors that cannot raise an error is recommended because it is relatively easy to forget to catch errors raised in a constructor if nothing in the constructor signature tells it can raise an error. See the next chapter for a discussion about the *error/exception handling principle*.

Figure 4.133

```
class Url:
    def __init__(
        self,
        scheme: str,
        port: int,
        host: str,
        path: str,
        query: str
    ):
        # Validate the arguments and raise an error if invalid
```

You can use the static factory method pattern to overcome the problem of raising an error in a constructor. You can make a factory method to return an optional value (if you don't need to return an error cause) or make the factory method raise an error. You can add a *try* prefix to the factory method name to signify that it can raise an error. Then, the function signature (function name) communicates to readers that the function may raise an error.

Below is an example class with two factory methods. The constructor of the class is made private using a `PrivateConstructor` metaclass. Users can only create instances of the class by using a factory method.

Figure 4.134

```
from typing import Any, TypeVar

T = TypeVar("T")


class PrivateConstructor(type):
    def __call__(
        cls: type[T],
        *args: tuple[Any, ...],
        **kwargs: dict[str, Any]
     ):
        raise TypeError('Constructor is private')

    def _create(
        cls: type[T],
        *args: tuple[Any, ...],
        **kwargs: dict[str, Any]
    ) -> T:
        return super().__call__(*args, **kwargs)


class Url(metaclass=PrivateConstructor):
    def __init__(
        self,
        scheme: str,
        port: int,
        host: str,
        path: str,
```

```
        query: str
    ):
        # ...

    @classmethod
    def create_url(
        cls,
        scheme: str,
        port: int,
        host: str,
        path: str,
        query: str
    ) -> 'Url | None':
        # Validate the arguments and return 'None' if invalid
        # If valid return a 'Url' instance:
        # return cls._create(str, port, host, path, query)

    @classmethod
    def try_create_url(
        cls,
        scheme: str,
        port: int,
        host: str,
        path: str,
        query: str
    ) -> 'Url':
        # Validate the arguments and raise an error if invalid
        # If valid return a 'Url' instance:
        # return cls._create(str, port, host, path, query)
```

Returning an optional value from a factory method allows the utilization of functional programming techniques. Python does not have an optional class, but let's first define an `Optional` class:

Figure 4.135

```
from collections.abc import Callable
from typing import Generic, TypeVar

T = TypeVar('T')
U = TypeVar('U')


class Optional(Generic[T], metaclass=PrivateConstructor):
    def __init__(self, value: T | None):
        self.__value = value

    @classmethod
    def of(cls, value: T) -> 'Optional[T]':
        return cls._create(value)

    @classmethod
    def of_nullable(cls, value: T | None) -> 'Optional[T]':
        return cls._create(value)

    @classmethod
    def empty(cls) -> 'Optional[T]':
        return cls._create(None)
```

```
    def is_empty(self) -> bool:
        return True if self.__value is None else False

    def is_present(self) -> bool:
        return False if self.__value is None else True

    def try_get(self) -> T:
        if self.__value is None:
            raise ValueError('No value to get')
        return self.__value

    def try_get_or_else_raise(self, error: Exception):
        if self.__value is None:
            raise error
        return self.__value

    def if_present(self, consume: Callable[[T], None]) ->
'Optional[T]':
        if self.__value is not None:
            consume(self.__value)
        return self

    def or_else(self, other_value: T) -> T:
        return other_value if self.__value is None else
self.__value

    def or_else_get(self, supply_value: Callable[[], T]) -> T:
        return supply_value() if self.__value is None else
self.__value

    def map(self, map_: Callable[[T], U | None]) ->
'Optional[U]':
        return (
            self
            if self.__value is None
            else self.of_nullable(map_(self.__value))
        )

    def flat_map(self, map_: Callable[[T], 'Optional[U]']) ->
'Optional[U]':
        return self if self.__value is None else
map_(self.__value)
```

> **NOTE!** When I use the `Optional` class in this book, it is always the above-defined class, not the `Optional` from the Python's `typing` module.

Notice how the above `Optional` class code utilized the static factory method pattern. It has a private constructor and three factory methods to create different kinds of `Optional` objects. The additional benefit of using the static factory method pattern is that you can name the factory methods descriptively,

which you can't do with a single constructor. The name of the factory method tells what kind of object will be created.

Figure 4.136

```
class Url(metaclass=PrivateConstructor):
     def __init__(
          self,
          scheme: str,
          port: int,
          host: str,
          path: str,
          query: str
      ):
          # ...

    @classmethod
    def create_url(
        cls,
        scheme: str,
        host: str,
        port: int,
        path: str,
        query: str
    ) -> Optional['Url']:
        # ...


maybe_url = Url.create_url(...)

# Do something with the URL in lambda
maybe_url.if_present(lambda url: print(url))

def print_url(url: Url):
    print(url)

# Do something with the URL using a function
maybe_url.if_present(print_url)
```

4.13.1.4: Builder Pattern

> ***Builder pattern allows you to construct objects piece by piece.***

In the builder pattern, you add properties to the built object with the *add_xxx* methods of the builder class. After adding all the needed properties, you can build the final object using the *build* or *build_xxx* method of the builder class.

For example, you can construct a URL from parts of the URL. Below is an example of using a `UrlBuilder` class:

Figure 4.137

```
url =
UrlBuilder().add_scheme('https').add_host('www.google.com').buil
d_url()
```

The builder pattern has the benefit that properties given for the builder can be validated in the build method. You can make the builder's build method return an optional indicating whether the building was successful. Or, you can make the build method raise an error if you need to provide an error. If the build method can raise an error, name it using a *try* prefix, for example, `try_build_url`. The builder pattern also has the benefit of not needing to add default properties to the builder. For example, *https* could be the default scheme, and if you are building an HTTPS URL, the `add_scheme` does not need to be called. The only problem is that you must consult the builder documentation to determine the default values. They are not visible in the API.

One drawback with the builder pattern is that you can give the parameters logically in the wrong order like this:

Figure 4.138

```
url =
UrlBuilder().add_host('www.google.com').add_scheme('https').buil
d_url()
```

It works, of course, but it does not look so nice. So, if you are using a builder, always try to give the parameters for the builder in a logically correct order if such an order exists. The builder pattern works well when there isn't any inherent order among the parameters. Below is an example of such a case: A house built with a `HouseBuilder` class.

Figure 4.139

```
house = HouseBuilder()\
  .add_kitchen()\
  .add_living_room()\
  .add_bedrooms(3)\
  .add_bath_rooms(2)\
  .add_garage()\
  .build_house()
```

You can achieve functionality similar to a builder with a factory method with parameters that have default values:

Figure 4.140

```
class Url(metaclass=PrivateConstructor):
     def __init__(
          self,
          host: str,
          path: str,
          query: str,
          scheme: str = 'https',
          port: int = 443,
      ):
          # ...

    @classmethod
    def create_url(
        cls,
         host: str,
         path: str,
         query: str,
         scheme: str = 'https',
         port: int = 443,
    ) -> 'Url | None':
        # ...
```

In the factory method above, there is clear visibility of what the default values are. But you cannot now give the parameters in a logical order. There is also a greater possibility that you accidentally provide some parameters in the wrong order because many of them are of the same type (string). This won't be a potential issue with a builder where you use a method with a specific name to give a specific parameter. In modern development environments, giving parameters in the wrong order is less probable because IDEs offer so-called inlay hints for parameters. With inlay hints enabled, it is easier to see if you provide a particular parameter in the wrong position. As shown below, giving parameters in the wrong order can also be avoided using semantically validated function parameter types. That topic will be discussed later in this chapter when we discuss primitive-type obsession.

Figure 4.141

```
class Url(metaclass=PrivateConstructor):
    # ...

    @classmethod
    def create_url(
        cls,
        host: Host,
        path: Path,
        query: Query,
        scheme: Scheme = Scheme.create('https'),
        port: Port = Port.create(443),
    ) -> 'Url | None':
        # ...
```

You can always use a parameter object as an additional alternative to a builder. Below is an example:

Figure 4.142

```
from dataclasses import dataclass

from Optional import Optional
from PrivateConstructor import PrivateConstructor

@dataclass
class UrlParams:
    host: str
    scheme: str = 'https'
    port: int = 443
    path: str = ""
    query: str = ""


class Url(metaclass=PrivateConstructor):
    def __init__(self, url_params: UrlParams):
        # ...

    @classmethod
    def create_url(cls, url_params: UrlParams) ->
'Optional[Url]':
        # ...


url_params = UrlParams('www.google.com',
query='query=design+patterns')
maybe_url = Url.create_url(url_params)
```

The above solution is quite similar to using a builder.

4.13.1.5: Singleton Pattern

> ***Singleton pattern defines that a class can have only one instance.***

Singletons are very common in pure object-oriented languages like Java. In many cases, a singleton class can be identified as not having any state. This is why only one instance of the class is needed. There is no point in creating multiple instances that are the same. In some non-pure object-oriented languages, singletons are not necessarily as common as in pure object-oriented languages and can often be replaced by just defining functions.

In Python, a singleton instance can be created in a module and exported from it. When you import the singleton instance from the module in other modules, the other modules will always get the same exported instance, not a new instance every time. Below is an example of such a singleton. First, we define a singleton instance of the `__MyClass` class in a module named *my_class_singleton.py*. The `__MyClass` class is prefixed with two underscores to denote it is private. The only public thing in the module is the singleton instance.

Figure 4.143. my_class_singleton.py

```
class __MyClass:
    # ...

my_class_singleton = __MyClass()
```

And in the *other_module_1.py* module, we import the singleton instance and use it:

Figure 4.144. other_module_1.py

```
from my_class_singleton import my_class_singleton

print(my_class_singleton)
```

We use the singleton instance also in *other_module_2.py* module:

Figure 4.145. other_module_2.py

```
from my_class_singleton import my_class_singleton
import other_module_1

print(my_class_singleton)
```

When you run the *other_module_2.py*, you should have the following kind of output where the object addresses are the same meaning that `my_class_singleton` is really a singleton.

Figure 4.146

```
<my_class_singleton.__MyClass object at 0x101042f90>
<my_class_singleton.__MyClass object at 0x101042f90>
```

The singleton pattern can also be implemented using a class with static methods only. The problem with such a class is that the singleton class is then hardcoded, and static classes can be hard or impossible to mock in unit testing (at least in some languages). We should remember to *program against*

interfaces. Also, if you try to implement the singleton pattern yourself, you must remember to ensure thread safety in case of a multi-threaded program when creating the singleton instance. You might be better off using a dependency injection library that can provide thread-safe singletons out of the box.

The best way to implement the singleton pattern is by using the *dependency inversion principle* and the *dependency injection principle*. Below is an example using the dependency-injector library for handling dependency injection. The constructor of the `FileConfigReader` class expects a `ConfigParser` instance. We annotate the constructor with the `@inject` annotation and provide a `ConfigParser` instance with the name `config_parser` from the DI container (defined later):

Figure 4.147

```
from typing import Final, Protocol

from dependency_injector.wiring import Provide, inject
# Import ConfigParser protocol with try_parse method here...
# Import Config ...


class ConfigReader(Protocol):
    def try_read(self, config_location: str) -> Config:
        pass


class FileConfigReader(ConfigReader):
    @inject
    def __init__(
        self,
         config_parser: ConfigParser = Provide['config_parser']
     ):
        self.__config_parser: Final = config_parser


    def try_read(self, config_file_path_name: str) -> Config:
        config_file_contents = # Read configuration file
        config =
self.__config_parser.try_parse(config_file_contents)
        return config
```

In the below `DiContainer` class, we first configure wiring, and then the name `config_parser` is bound to a singleton instance of the `ConfigParserImpl` class. (The `ConfigParserImpl` class code is not shown here). The `wiring_config` expects that the `FileConfigReader` class is defined in a module named `FileConfigReader.py`.

Figure 4.148

```
from dependency_injector import containers, providers


class DiContainer(containers.DeclarativeContainer):
    wiring_config = containers.WiringConfiguration(
        modules=['FileConfigReader']
    )

    config_parser =
providers.ThreadSafeSingleton(ConfigParserImpl)
```

The *dependency-injector* library will ensure that creating the `config_parser` singleton happens without race conditions in a multiple threads context. You can use `providers.Singleton` if you only have a single thread in your software component.

Another way to ensure that only one singleton instance is created is to ensure the DI container is created at the beginning of the application initialization (before starting threads) and singletons are created eagerly, not lazily. Eagerly means the singleton is created immediately, and lazily means it is created only when somebody needs it. A lazy instantiation can bring problems in a multi-threaded environment if synchronization is not used when the singleton instance is actually created.

4.13.1.6: Prototype Pattern

> ***The prototype pattern lets you create a new object using an existing object as a prototype.***

Let's have an example with a `DrawnShape` class:

Figure 4.149

```
from typing import Final, Protocol


class Shape(Protocol):
    # ...


# Implement concrete shapes...


class Position(Protocol):
    def get_x(self) -> int:
        pass

    def get_y(self) -> int:
```

```
        pass


class DrawnShape:
    def __init__ (self, position: Position, shape: Shape):
        self.__position: Final = position
        self.__shape: Final = shape

    def clone(self, position: Position) -> 'DrawnShape':
        return DrawnShape(position, self.__shape)
```

Applying the prototype pattern is to call the `clone` method on a prototype object and give the `position` parameter to specify where the new shape should be positioned.

The prototype pattern is also used in JavaScript to implement prototypal inheritance. Since EcmaScript version 6, class-based inheritance has been available, and prototypal inheritance does not need to be used.

The idea behind prototypal inheritance is that the common parts for the same class objects are stored in a prototype instance. These common parts typically mean the shared methods. There is no sense in storing the methods multiple times in each object. That would be a waste of resources because Javascript functions are objects themselves.

When you create a new object with the `Object.create` method, you give the prototype as a parameter. After that, you can set properties for the newly created object. When you call a method on the created object, and if that method is not found in the object's properties, the prototype object will be looked up for the method. Prototypes can be chained so that a prototype object contains another prototype object. This chaining is used to implement an inheritance chain. Below is a simple example of prototypal inheritance:

Figure 4.150

```
const pet = {
  name: '',
  getName: function() { return this.name; }
};

// Creates a new object with 'pet' object as a prototype
const petNamedBella = Object.create(pet);

petNamedBella.name = 'Bella';
console.log(petNamedBella.getName()); // Prints 'Bella'

// Prototype of a dog which contains 'pet' as nested prototype
```

```
const dog = {
  bark: function() { console.log('bark'); },
  __proto__: pet
}

// Creates a new object with 'dog' object as prototype
const dogNamedLuna = Object.create(dog);

dogNamedLuna.name = 'Luna';
console.log(dogNamedLuna.getName()); // Prints 'Luna'
dogNamedLuna.bark(); // Prints 'bark'
```

4.13.1.7: Object Pool Pattern

In the object pool pattern, created objects are stored in a pool where objects can be acquired from and returned for reuse. The object pool pattern is an optimization pattern because it allows the reuse of once-created objects.

If you need to create many short-lived objects, you should utilize an object pool and reduce the need for memory allocation and de-allocation, which takes time. Frequent object creation and destruction in garbage-collected languages cause extra work for the garbage collector, which consumes CPU time.

Below is an example object pool protocol.

Figure 4.151

```
from typing import Final, Protocol, TypeVar

T = TypeVar('T')


class ObjectPool(Protocol[T]):
    def acquire_object(self, cls: type[T]) -> T:
        pass

    def return_object(self, object_: T) -> None:
        pass
```

Below is an example object pool implementation.

Figure 4.152

```
class LimitedSizeObjPool(ObjectPool[T]):
    def __init__(self, max_pool_size: int):
        self.__max_pool_size: Final = max_pool_size
        self.__pooled_objects: Final[list[T]] = []
```

```
    def acquire_object(self, cls: type[T]) -> T:
        if self.__pooled_objects:
            return self.__pooled_objects.pop(0)
        else:
            return cls()

    def return_object(self, object_: T) -> None:
        pool_is_not_full = len(self.__pooled_objects) <
self.__max_pool_size

        if pool_is_not_full:
            self.__pooled_objects.append(object_)


class MyObject:
    # ...


my_object_pool: LimitedSizeObjPool[MyObject] =
LimitedSizeObjPool(2)
my_object_1 = my_object_pool.acquire_object(MyObject)
my_object_2 = my_object_pool.acquire_object(MyObject)
my_object_3 = my_object_pool.acquire_object(MyObject)
my_object_pool.return_object(my_object_1)
my_object_pool.return_object(my_object_2)
my_object_pool.return_object(my_object_3) # Does not fit to pool
print(len(my_object_pool._LimitedSizeObjPool__pooled_objects))
my_object_4 = my_object_pool.acquire_object(MyObject)
print(my_object_1 == my_object_4) # Prints True
```

Below is a slightly different implementation of an object pool. The below implementation accepts clearable objects, meaning objects returned to the pool are cleared before reusing. You can also supply parameters used when constructing an object.

Figure 4.153

```
class Clearable(Protocol):
    def clear(self) -> None:
        pass


T = TypeVar('T', bound=Clearable)


class LimitedSizeObjPool(ObjectPool[T]):
    def __init__(self, max_pool_size: int, *args, **kwargs):
        self.__max_pool_size: Final = max_pool_size
        self.__args: Final = args
        self.__kwargs: Final = kwargs
        self.__pooled_objects: Final[list[T]] = []

    def acquire_object(self, cls: type[T]) -> T:
        if self.__pooled_objects:
            return self.__pooled_objects.pop(0)
        else:
            return cls(*self.__args, **self.__kwargs)

    def return_object(self, object_: T) -> None:
```

```
        pool_is_not_full = len(self.__pooled_objects) <
self.__max_pool_size

        if pool_is_not_full:
            object_.clear()
            self.__pooled_objects.append(object_)


class MyObject(Clearable):
    def __init__(self, param1: int, param2: str, **kwargs):
        print(param1, param2, kwargs)

    def clear(self) -> None:
        print('Cleared')


my_object_pool: LimitedSizeObjPool[MyObject] =
LimitedSizeObjPool(
    2, 1, 'test', name='John'
)

# Prints: 1 test {'name': 'John'}
my_object_1 = my_object_pool.acquire_object(MyObject)

# Prints: Cleared
my_object_pool.return_object(my_object_1)
```

4.13.2: Structural Design Patterns

This section describes structural design patterns. Most patterns use object composition as the primary method to achieve a particular design. The following design patterns are presented:

- Composite pattern
- Facade pattern
- Bridge pattern
- Strategy pattern
- Adapter pattern
- Proxy pattern
- Decorator pattern
- Flyweight pattern

4.13.2.1: Composite Pattern

In the composite pattern, a class can be composed of itself, i.e., the composition is recursive.

Recursive object composition can be depicted by how a user interface can be composed of different widgets. The example below shows a `Pane` class that is a `Widget`. A `Pane` object can contain several other `Widget` objects, meaning a `Pane` object can contain other `Pane` objects.

Figure 4.154

```
from typing import Final, Protocol


class Widget(Protocol):
    def render(self) -> None:
        pass


class Pane(Widget):
    def __init__(self, widgets: list[Widget]):
        self.__widgets: Final = widgets.copy()

    def render(self) -> None:
        # Render each widget inside pane


class StaticText(Widget):
    def render(self) -> None:
        # Render static text widget


class TextInput(Widget):
    def render(self) -> None:
        # Render text input widget


class Button(Widget):
   def render(self) -> None:
        # Render button widget


class Window:
    def __init__(self, widgets: list[Widget]):
        self.__widgets: Final = widgets.copy()

    def render(self) -> None:
        for widget in self.__widgets:
            widget.render()
```

Objects that form a tree structure are composed of themselves recursively. Below is an Avro record field schema with a nested record field:

Figure 4.155

```
{
  "type": "record",
  "name": "sampleMessage",
  "fields": [
    {
      "name": "field1",
      "type": "string"
    },
    {
      "name": "nestedRecordField",
      "namespace": "nestedRecordField",
      "type": "record",
      "fields": [
        {
          "name": "nestedField1",
          "type": "int",
          "signed": "false"
        }
      ]
    }
  ]
}
```

For parsing an Avro schema, we could define classes for different sub-schemas by the field type. When analyzing the below example, we can notice that the `RecordAvroFieldSchema` class can contain any `AvroFieldSchema` object, also other `RecordAvroFieldSchema` objects, making a `RecordAvroFieldSchema` object a composite object.

Figure 4.156

```
from typing import Protocol


class AvroFieldSchema(Protocol):
    # ...


class RecordAvroFieldSchema(AvroFieldSchema):
    def __init__(self, sub_field_schemas:
list[AvroFieldSchema]):
        self.__sub_field_schemas: Final =
sub_field_schemas.copy()


class StringAvroFieldSchema(AvroFieldSchema):
    # ...


class IntAvroFieldSchema(AvroFieldSchema):
    # ...


# Schema classes for rest of Avro data types ...
```

4.13.2.2: Facade Pattern

> ***In the facade pattern, an object on a higher level of abstraction is composed of objects on a lower level of abstraction. The higher-level object acts as a facade in front of the lower-level objects. Lower-level objects behind the facade are either only or mainly only accessible by the facade.***

Let's use the data exporter microservice as an example. For that microservice, we could create a `Config` interface that can be used to obtain configuration for the different parts (input, transform, and output) of the data exporter microservice. The `Config` interface acts as a facade. Users of the facade need not see behind the facade. They don't know what happens behind the facade. And they shouldn't care because they are just using the interface provided by the facade.

There can be various classes doing the actual work behind the facade. In the below example, there is a `ConfigReader` class that reads configuration from possibly different sources (from a local file or a remote service, for example) and there are configuration parsers that can parse a specific part of the configuration, possibly in different data formats like JSON or YAML. None of these implementations and details are visible to the user of the facade. Any of these implementations behind the facade can change at any time without affecting the users of the facade because facade users are not coupled to the lower-level implementations.

Below is the implementation of the `Config` facade:

Figure 4.157

```
from typing import Protocol

from dependency_injector.wiring import Provide, inject
# Rest of imports ...


class Config(Protocol):
    def try_get_input_config(self) -> InputConfig:
        pass

    def try_get_transformer_config(self) -> TransformerConfig:
        pass
```

```
    def try_get_output_config(self) -> OutputConfig:
        pass


class ConfigImpl(Config):
    @inject
    def __init__(
        self,
        config_reader: ConfigReader = Provide['config_reader'],
        input_config_parser: InputConfigParser = Provide[
            'input_config_parser'
        ],
        transformer_config_parser: TransformerConfigParser =
Provide[
            'transformer_config_parser'
        ],
        output_config_parser: OutputConfigParser = Provide[
            'output_config_parser'
        ]
    ):
        self.__config_reader: Final = config_reader
        self.__input_config_parser: Final = input_config_parser
        self.__transformer_config_parser: Final =
transformer_config_parser
        self.__output_config_parser: Final =
output_config_parser
        self.__config_string = ""
        self.__input_config = None
        self.__output_config = None
        self.__transformer_config = None

    def try_get_input_config(self) -> InputConfig:
        if self.__input_config is None:
            self.__try_read_config_if_needed()

            self.__input_config =
self.__input_config_parser.try_parse(
                self.__config_string
            )

        return self.__input_config

    def try_get_transformer_config(self) -> TransformerConfig:
        # ...

    def try_get_output_config(self) -> OutputConfig:
        # ...

    def __try_read_config_if_needed(self) -> None:
        if not self.__config_string:
            self.__config_string =
self.__config_reader.try_read(...)
```

4.13.2.3: Bridge Pattern

In the bridge pattern, the implementation of a class is delegated to another class. The original class is

> ***"abstract" in the sense that it does not have any behavior except the delegation to another class, or it can have some higher level control logic on how it delegates to another class.***

Don't confuse the word "abstract" here with an abstract class. In an abstract class, some behavior is not implemented at all, but the implementation is deferred to subclasses of the abstract class. Here, instead of "abstract class", we could use the term *delegating class* instead.

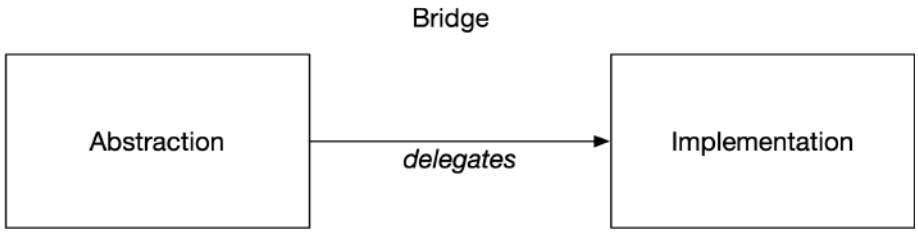

Figure 4.158. Bridge Pattern

Let's have an example with shapes and drawings capable of drawing different shapes:

Figure 4.159

```
from typing import Final, Protocol

from Point import Point
from ShapeRenderer import ShapeRenderer


class Shape(Protocol):
    def render(self, renderer: ShapeRenderer) -> None:
        pass


class RectangleShape(Shape):
    def __init__(
        self,
        upper_left_corner: Point,
        width: int,
        height: int
    ):
        self.__upper_left_corner: Final = upper_left_corner
        self.__width: Final = width
        self.__height: Final = height

    def render(self, renderer: ShapeRenderer) -> None:
        renderer.render_rectangle(
            self.__upper_left_corner,
            self.__width,
            self.__height
```

```
        )


class CircleShape(Shape):
    def __init__(self, center: Point, radius: int):
        self.__center: Final = center
        self.__radius: Final = radius

    def render(self, renderer: ShapeRenderer):
        renderer.render_circle(self.__center, self.__radius)
```

The above `RectangleShape` and `CircleShape` classes are abstractions (or delegating classes) because they delegate their functionality (rendering) to an external class (implementation class) of the `ShapeRenderer` type. We can provide different rendering implementations for the shape classes. Let's define two shape renderers, one for rendering raster shapes and another for rendering vector shapes:

Figure 4.160

```
from typing import Final, Protocol

from Canvas import Canvas
from Point import Point
from SvgElement import SvgElement


class ShapeRenderer(Protocol):
    def render_circle(self, center: Point, radius: int) -> None:
        pass

    def render_rectangle(
        self,
        upper_left_corner: Point,
        width: int,
        height: int
    ) -> None:
        pass

    # Methods for rendering other shapes ...


class RasterShapeRenderer(ShapeRenderer):
    def __init__(self, canvas: Canvas):
        self.__canvas: Final = canvas

    def render_circle(self, center: Point, radius: int):
        # Render circle to canvas

    def render_rectangle(
        self,
        upper_left_corner: Point,
        width: int,
        height: int
    ):
        # Render rectangle to canvas
```

```
    # Methods for rendering other shapes to canvas ...


class VectorShapeRenderer(ShapeRenderer):
    def __init__(self, svg_root: SvgElement):
        self.__svg_root: Final = svg_root

    def render_circle(self, center: Point, radius: int):
        # Render circle as SVG element and attach as child to
SVG root

    def render_rectangle(
        self,
        upper_left_corner: Point,
        width: int,
        height: int
    ):
        # Render rectangle as SVG element
        # and attach as child to SVG root

    # Methods for rendering other shapes ...
```

Let's implement two different drawings, a raster and a vector drawing:

Figure 4.161

```
from abc import abstractmethod
from typing import Final, Protocol

from Canvas import Canvas
from RasterShapeRenderer import RasterShapeRenderer
from Shape import Shape
from ShapeRenderer import ShapeRenderer
from SvgElement import SvgElement
from VectorShapeRenderer import VectorShapeRenderer


class Drawing(Protocol):
    def draw(self, shapes: list[Shape]) -> None:
        pass

    def save(self) -> None:
        pass


class AbstractDrawing(Drawing):
    def __init__(self, name: str):
        self.__name: Final = name

    @final
    def draw(self, shapes: list[Shape]) -> None:
        for shape in shapes:
            shape.render(self._get_shape_renderer())

    @final
    def save(self) -> None:
        file_name = self.__name + self._get_file_extension()
        data = self._get_data()
```

```
        # Save the 'data' to 'file_name' ...

    @abstractmethod
    def _get_shape_renderer(self) -> ShapeRenderer:
        pass

    @abstractmethod
    def _get_file_extension(self) -> str:
        pass

    @abstractmethod
    def _get_data(self) -> bytearray:
        pass


class RasterDrawing(AbstractDrawing):
    def __init__(self, name: str):
        super().__init__(name)
        self.__canvas: Final = Canvas()
        self.__shape_renderer =
RasterShapeRenderer(self.__canvas)

    def _get_shape_renderer(self) -> ShapeRenderer:
        return self.__shape_renderer

    def _get_file_extension(self) -> str:
        return '.png'

    def _get_data(self) -> bytearray:
        # Get data from the 'canvas' object


class VectorDrawing(AbstractDrawing):
    def __init__(self, name: str):
        super().__init__(name)
        self.__svg_root: Final = SvgElement()
        self.__shape_renderer =
VectorShapeRenderer(self.__svg_root)

    def _get_shape_renderer(self) -> ShapeRenderer:
        return self.__shape_renderer

    def _get_file_extension(self) -> str:
        return '.svg'

    def _get_data(self) -> bytearray:
        # Get data from the 'svg_root' object
```

In the above example, we have delegated the rendering behavior of the shape classes to concrete classes implementing the `ShapeRenderer` protocol. The `Shape` classes only represent a shape but don't render the shape. They have a single responsibility of representing a shape. Regarding rendering, the shape classes are "abstractions" because they delegate the rendering to another class responsible for rendering different shapes.

Now, we can have a list of shapes and render them differently. We can do this as shown below because we did not couple the shape classes with any specific rendering behavior.

Figure 4.162

```
shapes = [RectangleShape(Point(...), 2, 3),
CircleShape(Point(...), 4)]

raster_drawing = RasterDrawing('raster-drawing')
raster_drawing.draw(shapes)
raster_drawing.save()

vector_drawing = VectorDrawing('vector-drawing')
vector_drawing.draw(shapes)
vector_drawing.save()
```

4.13.2.4: Strategy Pattern

> ***In the strategy pattern, the functionality of an object can be changed by changing an instance of a composed type to a different instance of that type.***

Below is an example where the behavior of a `ConfigReader` class can be changed by changing the value of the `config_parser` attribute to an instance of a different class. The default behavior is to parse the configuration in JSON format, which can be achieved by calling the constructor without a parameter.

Figure 4.163

```
from typing import Final, Protocol


# Define Config class ...


class ConfigParser(Protocol):
    def try_parse(self, config_str: str) -> Config:
        pass


# Define JsonConfigParser class ...


class ConfigReader:
    def __init__(self, config_parser: ConfigParser =
JsonConfigParser()):
        self.__config_parser: Final = config_parser

    def try_read(self, config_file_path_name: str) -> Config:
        # Try read 'config_file_path_name' to 'config_str'
```

```
        config = self.__config_parser.try_parse(config_str)
        return config
```

Using the strategy pattern, we can change the functionality of a `ConfigReader` instance by changing the `config_parser` attribute value. For example, there could be the following classes available that implement the `ConfigParser` protocol:

- `JsonConfigParser`
- `YamlConfigParser`
- `TomlConfigParser`
- `XmlConfigParser`

We can dynamically change the behavior of a `ConfigReader` instance to use the YAML parsing strategy by giving an instance of the `YamlConfigParser` class as a parameter for the `ConfigReader` constructor.

4.13.2.5: Adapter Pattern

> ***The adapter pattern changes one interface to another interface. It allows you to adapt different interfaces to a single interface.***

In the below example, we have defined a `Message` protocol for messages that can be consumed from a data source using a `MessageConsumer`.

Figure 4.164. Message.py

```
from typing import Protocol


class Message(Protocol):
    def get_data(self) -> bytearray:
        pass

    def get_length_in_bytes(self) -> int:
        pass
```

Figure 4.165. MessageConsumer.py

```
from typing import Protocol

import Message from Message
```

```
class MessageConsumer(Protocol):
    def consume_message(self) -> Message:
        pass
```

Next, we can define the message and message consumer adapter classes for Apache Kafka and Apache Pulsar:

Figure 4.166. KafkaMsgConsumer.py

```
from MessageConsumer import MessageConsumer


class KafkaMsgConsumer(MessageConsumer):
    def consume_message(self) -> Message:
        # Consume a message from Kafka using a 3rd party
        # Kafka library
        # Wrap the consumed message inside an instance
        # of the below-defined KafkaMessage class
        # Return that KafkaMessage instance
```

Figure 4.167. KafkaMessage.py

```
from typing import Final

from Message import Message


class KafkaMessage(Message):
    def __init__(self, kafka_lib_msg):
        # This is the "raw" message from the Kafka library
        self.__kafka_lib_msg: Final = kafka_lib_msg

    def get_data(self) -> bytearray:
        # Suppose there is 'value' property to get the
        # message data as byte array
        return self.__kafka_lib_msg.value

    def get_length_in_bytes(self) -> int:
        return len(self.__kafka_lib_msg.value)
```

Figure 4.168. PulsarMsgConsumer.py

```
from MessageConsumer import MessageConsumer


class PulsarMsgConsumer(MessageConsumer):
    def consume_message(self) -> Message:
        # Consume a message from Pulsar using a Pulsar client
        # library.
        # Wrap the consumed Pulsar message inside an instance
        # of the below-defined PulsarMessage
        # Return that PulsarMessage instance
```

Figure 4.169. PulsarMessage.py

```
from typing import Final

from Message import Message
```

```
class PulsarMessage(Message):
     def __init__(self, pulsar_lib_msg):
         # This is the "raw" message from the Pulsar library
         self.__pulsar_lib_msg: Final = pulsar_lib_msg

    def get_data(self) -> bytearray:
         # Suppose there is 'data' method to get the
         # message data as byte array
         return self.__pulsar_lib_msg.data()

    def get_length_in_bytes(self) -> int:
         # Suppose there is 'length' method
         # to get the message length
         return len(self.__pulsar_lib_msg.length())
```

Now, we can use Kafka or Pulsar data sources with identical consumer and message interfaces. In the future, it will be easy to integrate a new data source into the system. We only need to implement appropriate adapter classes (message and consumer classes) for the new data source. No other code changes are required. Thus, we would be following the *open-closed principle* correctly.

Let's imagine that the API of the used Kafka library changed. We don't need to make changes in many places in the code. We need to create new adapter classes (message and consumer classes) for the new API and use those new adapter classes in place of the old adapter classes. All of this work is again following the *open-closed principle*.

Consider using the adapter pattern even if there is nothing to adapt to, especially when working with 3rd party libraries. Because then you will be prepared for the future when changes can come. It might be possible that a 3rd party library interface changes or there is a need to take a different library into use. If you have not used the adapter pattern, taking a new library or library version into use could mean that you must make many small changes in several places in the codebase, which is error-prone and against the *open-closed principle*.

Let's have an example of using a 3rd party logging library. Initially, our adapter `AbcLogger` for a fictive *abc-logging-library* is just a wrapper around the `abc_logger` instance from the library. There is not any actual adapting done.

Figure 4.170. Logger.py

```
from typing import Protocol

from LogLevel import LogLevel


class Logger(Protocol):
    def log(self, log_level: LogLevel, message: str) -> None:
        pass
```

Figure 4.171. AbcLogger.py

```
from abc_logging_library import abc_logger
from Logger import Logger
from LogLevel import LogLevel

class AbcLogger(Logger):
    def log(self, log_level: LogLevel, message: str) -> None:
        abc_logger.log(log_level, message)
```

When you use the logger in your application, you can utilize the *singleton pattern* and create a singleton instance of the `AbcLogger` in the DI container and let the DI framework inject the logger to all parts of the software component where a logger is needed. Here is how the DI container could look:

Figure 4.172. DiContainer.py

```
from AbcLogger import AbcLogger
from dependency_injector import containers, providers


class DiContainer(containers.DeclarativeContainer):
    wiring_config = containers.WiringConfiguration(
        # List all the modules here
        modules=[...]
    )

    logger = providers.ThreadSafeSingleton(AbcLogger)
    # Other bindings ...
```

Now, when you need the logger in any other class of the application, you can get it:

Figure 4.173

```
from dependency_injector.wiring import Provide, inject
from Logger import Logger


class SomeClass:
    @inject
    def __init__(self, logger: Logger = Provide['logger']):
        # ...
```

or even easier:

Figure 4.174

```
from dependency_injector.wiring import Provide
from Logger import Logger


class SomeClass:
    __logger: Logger = Provide['logger']

    def __init__(self):
        # ...
```

Suppose that in the future, a better logging library is available called *xyz-logging-library*, and we would like to take that into use, but it has a slightly different interface. Its logging instance is called `xyz_log_writer`, the logging method is named differently, and the parameters are given in different order compared to the *abc-logging-library*. We can create a `XyzLogger` adapter class for the new logging library and update the `DiContainer`. No other code changes are required elsewhere in the codebase to take the new logging library into use.

Figure 4.175. XyzLogger.py

```
from xyz_logging_library import xyz_log_writer
from Logger import Logger
from LogLevel import LogLevel

class XyzLogger(Logger):
    def log(self, log_level: LogLevel, message: str) -> None:
        xyz_log_writer.write_log_entry(message, log_level)
```

Figure 4.176. DiContainer.py

```
# ...

class DiContainer(containers.DeclarativeContainer):
    # ...
    logger = providers.ThreadSafeSingleton(XyzLogger)
    # ...
```

We didn’t have to modify all the places where logging is used in the codebase (and we can be sure that logging is used in many places!). We have saved ourselves from a lot of error-prone and unnecessary work, and once again, we have followed the *open-closed principle* successfully.

In some languages (not Python) where mocking of, e.g., concrete classes is not possible, wrapping a 3rd party library in an adapter class enables you to unit test against the adapter

class interface instead of the concrete classes of the 3rd party library.

4.13.2.6: Proxy Pattern

> ***The proxy pattern enables conditionally modifying or augmenting the behavior of an object.***

When using the proxy pattern, you define a proxy class that wraps another class (the proxied class). The proxy class conditionally delegates to the wrapped class. The proxy class implements the interface of the wrapped class and is used in place of the wrapped class in the code.

Below is an example of a proxy class, `CachingEntityStore`, that caches the results of entity store operations:

Figure 4.177. Cache.py

```
from typing import Protocol, TypeVar

TKey = TypeVar('TKey')
TValue = TypeVar('TValue')


class Cache(Protocol[TKey, TValue]):
    def retrieve_value(self, key: TKey) -> TValue | None:
        pass

    def store(
        self,
        key: TKey,
        value: TValue,
        time_to_live_in_secs: int = 0
    ) -> None:
        pass
```

Figure 4.178. MemoryCache.py

```
from typing import TypeVar

from Cache import Cache

TKey = TypeVar('TKey')
TValue = TypeVar('TValue')

class MemoryCache(Cache[TKey, TValue]):
    # ...

    def retrieve_value(self, key: TKey) -> TValue | None:
        # ...

    def store(
```

```
        self,
        key: TKey,
        value: TValue,
        time_to_live_in_secs: int = 0
    ) -> None:
        # ...
```

Figure 4.179. EntityStore.py

```
from collections.abc import Awaitable
from typing import Protocol, TypeVar

T = TypeVar('T')


class EntityStore(Protocol[T]):
    async def try_get_entity(self, id_: int) -> Awaitable[T]:
        pass
```

Figure 4.180. DbEntityStore.py

```
from collections.abc import Awaitable
from typing import TypeVar

from EntityStore import EntityStore

T = TypeVar('T')


class DbEntityStore(EntityStore[T]):
    async def try_get_entity(self, id_: int) -> Awaitable[T]:
        # Try get entity from database ...
```

Figure 4.181. CachingEntityStore.py

```
from collections.abc import Awaitable
from typing import Final, TypeVar

from EntityStore import EntityStore
from MemoryCache import MemoryCache

T = TypeVar('T')


class CachingEntityStore(EntityStore[T]):
    def __init__(self, entity_store: EntityStore[T]):
        self.__entity_store: Final = entity_store
        self.__entity_cache: Final[MemoryCache[int, T]] =
MemoryCache()

    async def try_get_entity(self, id_: int) -> Awaitable[T]:
        entity = self.__entity_cache.retrieve_value(id_)

        if entity is None:
            entity = await
self.__entity_store.try_get_entity(id_)
            time_to_live_in_secs = 60
            self.__entity_cache.store(id_, entity,
time_to_live_in_secs)

        return entity
```

In the above example, the `CachingEntityStore` class is the proxy class wrapping an instance implementing the `EntityStore` protocol, like an instance of the `DbEntityStore` class that stores entities in a database. The proxy class is modifying the wrapped class behavior by conditionally delegating to the wrapped class. It only delegates to the wrapped class if an entity is not found in the cache. Instead of a class, you can use the proxy pattern with a function by utilizing Python's `@cache` decorator.

Below is another example of a proxy class that authorizes a user before performing a service operation:

Figure 4.182

```
from collections.abc import Awaitable
from typing import Final, Protocol

from User import User
from UserAuthorizer import UserAuthorizer


class UserService(Protocol):
    class Error(Exception):
        pass

    async def try_get_user(self, id_: int) -> Awaitable[User]:
        pass


class UserServiceImpl(UserService):
    async def try_get_user(self, id_: int) -> Awaitable[User]:
        # Try get user by id ...


class AuthorizingUserService(UserService):
    def __init__(
        self,
        user_service: UserService,
        user_authorizer: UserAuthorizer
    ):
        self.__user_service: Final = user_service
        self.__user_authorizer: Final = user_authorizer

    async def try_get_user(self, id_: int) -> Awaitable[User]:
        try:
            await self.__user_authorizer.try_authorize_user(id_)
        except UserAuthorizer.AuthorizeUserError as error:
            raise self.Error(error)

        return await self.__user_service.try_get_user(id_)
```

In the above example, the `AuthorizingUserService` class is a proxy class that wraps an instance implementing the

`UserService` protocol, like an instance of the `UserServiceImpl` class (the actual implementation of the user service). The proxy class is modifying the wrapped class behavior by conditionally delegating to the wrapped class. It will delegate to the wrapped class only if authorization is successful.

As the last example, we could define a `RateLimitedXyzService` proxy class that wraps a `XyzService` class. The rate-limited service class delegates to the wrapped class only if the service calling rate limit is not exceeded. The rate-limited service class should raise an error if the rate is exceeded.

4.13.2.7: Decorator Pattern

> ***The decorator pattern enables augmenting the functionality of a class method(s) without the need to modify the class method(s).***

A decorator class wraps another class whose functionality will be augmented. The decorator class implements the interface of the wrapped class and is used in place of the wrapped class in the code. The decorator pattern is useful when you cannot modify an existing class, e.g., the existing class is in a 3rd party library. The decorator pattern also helps to follow the *open-closed principle* because you don't have to modify an existing method to augment its functionality. Instead, you can create a decorator class that contains the new functionality.

Below is an example of the decorator pattern. There is a standard SQL statement executor implementation and two decorated SQL statement executor implementations: one that adds logging functionality and one that adds SQL statement execution timing functionality. Finally, a double-decorated SQL statement executor is created that logs an SQL statement and times its execution.

Figure 4.183

```
import time
from collections.abc import Awaitable
from typing import Any, Protocol
```

```
from logger import logger
from LogLevel import LogLevel


class SqlStatementExecutor(Protocol):
    async def try_execute(
        self, sql_statement: str, parameter_values: list[Any] |
None = None
    ) -> Awaitable[Any]:
        pass


class SqlStatementExecutorImpl(SqlStatementExecutor):
    async def try_execute(
        self, sql_statement: str, parameter_values: list[Any] |
None = None
    ) -> Awaitable[Any]:
        return await self.__get_connection().execute(
            sql_statement, parameter_values
        )

    # Implement __get_connection() ...


class LoggingSqlStatementExecutor(SqlStatementExecutor):
    def __init__(self, sql_statement_executor:
SqlStatementExecutor):
        self.__sql_statement_executor = sql_statement_executor

    async def try_execute(
        self, sql_statement: str, parameter_values: list[Any] |
None = None
    ) -> Awaitable[Any]:

        logger.log(
            LogLevel.DEBUG, f'Executing SQL statement:
{sql_statement}'
        )

        return await self.__sql_statement_executor.try_execute(
            sql_statement, parameter_values
        )


class TimingSqlStatementExecutor(SqlStatementExecutor):
    def __init__(self, sql_statement_executor:
SqlStatementExecutor):
        self.__sql_statement_executor = sql_statement_executor

    async def try_execute(
        self, sql_statement: str, parameter_values: list[Any] |
None = None
    ) -> Awaitable[Any]:
        start_time_in_ns = time.time_ns()

        result = await
self.__sql_statement_executor.try_execute(
            sql_statement, parameter_values
        )

        end_time_in_ns = time.time_ns()
```

```
        duration_in_ns = end_time_in_ns - start_time_in_ns
        duration_in_ms = duration_in_ns / 1_000_000

        logger.log(
            LogLevel.DEBUG,
            f'SQL statement execution duration: {duration_in_ms}
ms',
        )

        return result


timing_and_logging_sql_statement_executor =
LoggingSqlStatementExecutor(
    TimingSqlStatementExecutor(SqlStatementExecutorImpl())
)
```

You can also use the decorator pattern with functions and methods in Python. Python decorators allow us to wrap a function to extend its behavior. Python decorators are functions that take a function as a parameter and return another function that is used in place of the decorated function. Let's have an elementary example of a function decorator in Python:

Figure 4.184

```
# Decorator
def print_hello(func):
    def wrapped_func(*args, **kwargs):
        print('Hello')
        return func(*args, **kwargs)

    return wrapped_func


@print_hello
def add(a: int, b: int) -> int:
    return a + b


result = add(1, 2)
print(result) # Prints: Hello 3
```

Let's have another example with a decorator that times the execution of a function and prints it to the console:

Figure 4.185

```
import time
from functools import wraps


# Decorator
def timed(func):
    @wraps(func)
```

```
    def wrapped_func(*args, **kwargs):
        start_time_in_ns = time.perf_counter_ns()
        result = func(*args, **kwargs)
        end_time_in_ns = time.perf_counter_ns()
        duration_in_ns = end_time_in_ns - start_time_in_ns

        print(
            f'Exec of func "{func.__name__}" took
{duration_in_ns} ns'
        )

        return result

    return wrapped_func


@timed
def add(a: int, b: int) -> int:
    return a + b


result = add(1, 2)
print(result)
# Prints, for example:
# Exec of func "add" took 625 ns
# 3
```

You can combine multiple decorators, for example:

Figure 4.186

```
# Decorator
def logged(func):
    @wraps(func)
    def wrapped_func(*args, **kwargs):
        result = func(*args, **kwargs)
        # In real-life, you use a logger here instead of print
        print(f'Func "{func.__name__}" executed')
        return result

    return wrapped_func

@logged
@timed
def add(a: int, b: int) -> int:
    return a + b

result = add(1, 2)
print(result)
# Prints, for example:
# Exec of func "add" took 583 ns
# Func "add" executed
# 3
```

If you change the order of decorators, you get the output in a different order, too. And the execution time of the function is now longer because the time spent in logging is also added to the total execution time:

Figure 4.187

```
@timed
@logged
def add(a: int, b: int) -> int:
    return a + b


result = add(1, 2)
print(result)
# Prints, for example:
# Func "add" executed
# Exec of func "add" took 9708 ns
# 3
```

You can also use decorator functions without the @-syntax to create new functions:

Figure 4.188

```
def add(a: int, b: int) -> int:
    return a + b

logged_add = logged(add)
timed_add = timed(add)
logged_timed_add = logged(timed(add))
timed_logged_add = timed(logged(add))

print(logged_add(1,2))
# Prints
# Func "add" executed
# 3

print(timed_add(1,2))
# Prints, for example
# Exec of func "add" took 209 ns
# 3

print(logged_timed_add(1, 2))
# Prints, for example
# Exec of func "add" took 208 ns
# Func "add" executed
# 3

print(timed_logged_add(1, 2))
# Prints, for example
# Func "add" executed
# Exec of func "add" took 1250 ns
# 3
```

4.13.2.8: Flyweight Pattern

The flyweight pattern is a memory-saving optimization pattern where flyweight objects reuse objects.

Let’s have a simple example with a game where different shapes are drawn at different positions. Let’s assume that the game draws a lot of similar shapes but in different positions so that we can notice the difference in memory consumption after applying this pattern.

Shapes that the game draws have the following properties: size, form, fill color, stroke color, stroke width, and stroke style.

Figure 4.189

```
from typing import Final, Protocol


class Shape(Protocol):
    # ...


# Define Color...
# Define StrokeStyle...


class AbstractShape(Shape):
    def __init__(
        self,
        fill_color: Color,
        stroke_color: Color,
        stroke_width: int,
        stroke_style: StrokeStyle
    ):
        self.__fill_color: Final = fill_color
        self.__stroke_color: Final = stroke_color
        self.__stroke_width: Final = stroke_width
        self.__stroke_style: Final = stroke_style


class CircleShape(AbstractShape):
    def __init__(
        self,
        fill_color: Color,
        stroke_color: Color,
        stroke_width: int,
        stroke_style: StrokeStyle,
        radius: int
    ):
        super().__init__(
            fill_color,
            stroke_color,
            stroke_width,
            stroke_style
        )

        self.__radius: Final = radius


# Define LineSegment ...
```

```
class PolygonShape(AbstractShape):
  def __init__(
      self,
      fill_color: Color,
      stroke_color: Color,
      stroke_width: int,
      stroke_style: StrokeStyle,
      line_segments: list[LineSegment]
  ):
      super().__init__(
          fill_color,
          stroke_color,
          stroke_width,
          stroke_style
      )

      self.__line_segments: Final = line_segments.copy()
```

When analyzing the `PolygonShape` class, we notice that it contains many properties that consume memory. A polygon with many line segments can consume a noticeable amount of memory. If the game draws many identical polygons in different screen positions and always creates a new `PolygonShape` object, there would be a lot of identical `PolygonShape` objects in the memory. To remediate this, we can introduce a flyweight class, `DrawnShapeImpl`, which contains the position of a shape and a reference to the actual shape. In this way, we can draw a lot of `DrawnShapeImpl` objects that all contain a reference to the same `PolygonShape` object:

Figure 4.190

```
from typing import Final, Protocol


class DrawnShape(Protocol):
    # ...


class DrawnShapeImpl(DrawnShape):
    def __init__(self, shape: Shape, screen_position: Position):
        self.__shape: Final = shape
        self.__screen_position: Final = screen_position

    # ...


polygon = PolygonShape(...)
positions = generate_lots_of_positions()
drawn_polygons = [DrawnShapeImpl(polygon, position) for position
in positions]
```

4.13.3: Behavioral Design Patterns

Behavioral design patterns describe ways to implement new behavior using object-oriented design. The following behavioral design patterns will be presented in the following sections:

- Chain of responsibility pattern
- Observer pattern
- Command/Action pattern
- Iterator pattern
- State pattern
- Mediator pattern
- Template method pattern
- Memento pattern
- Visitor pattern
- Null object pattern

4.13.3.1: Chain of Responsibility Pattern

> ***The chain of responsibility pattern lets you pass requests along a chain of handlers.***

The chain of responsibility pattern allows you to add pluggable behavior to handling requests. That pluggable behavior is something that can be executed always or conditionally. This pattern allows you to follow the open-closed principle because you don't modify the request-handling process directly but only extend it with plug-ins. This pattern allows you to follow the single responsibility principle by putting specific behavior into a plug-in.

When receiving a request, each handler can decide what to do:

- Process the request and then pass it to the next handler in the chain

- Process the request without passing it to the subsequent handlers (terminating the chain)
- Leave the request unprocessed and pass it to the next handler

The FastAPI web framework utilizes the chain of responsibility pattern for handling requests. In the FastAPI framework, you can write pluggable behavior using middlewares, a concept similar to Servlet filters in Java. Below is an example of a middleware that adds HTTP request processing time to the response in a custom HTTP header:

Figure 4.191

```
import time

from fastapi import FastAPI, Request

app = FastAPI()


@app.middleware('http')
async def add_request_processing_time_header(request: Request,
call_next):
    start_time_in_ns = time.time_ns()
    response = await call_next(request)
    end_time_in_ns = time.time_ns()
    processing_time_in_ns = end_time_in_ns - start_time_in_ns
    processing_time_in_ms = processing_time_in_ns / 1_000_000

    response.headers['X-Processing-Time-Millis'] = str(
        processing_time_in_ms
    )

    return response
```

Below is an authorization and logging middleware example:

Figure 4.192

```
from fastapi import FastAPI, Response, Request

app = FastAPI()

# Authorization middleware
@app.middleware('http')
async def authorize(request: Request, call_next):
    # From request's 'Authorization' header,
    # extract the bearer JWT, if present
    # set 'token_is_present' variable value to True
    # Verify the validity of JWT and assign result
    # to 'token_is_valid' variable

    if token_is_valid:
        response = await call_next(request)
```

```
    elif token_is_present:
        # NOTE! call_next is not invoked,
        # this will terminate the request
        response = Response('Unauthorized', 403)
    else:
        # NOTE! call_next is not invoked,
        # this will terminate the request
        response = Response('Unauthenticated', 401)

    return response


# Logging middleware
@app.middleware('http')
async def log(request: Request, call_next):
    print(f'GET {str(request.url)}')
    return await call_next(request)


@app.get('/hello')
def hello():
    return 'Hello!'
```

4.13.3.2: Observer Pattern

The observer pattern lets you define an observe-notify (or publish-subscribe) mechanism to notify one or more objects about events that happen to the observed object.

One typical example of using the observer pattern is a UI view observing a model. The UI view will be notified whenever the model changes and can redraw itself. Let's have an example:

Figure 4.193

```
from typing import Protocol, Final


class Observer(Protocol):
    def notify_about_change(self) -> None:
        pass


class Observable(Protocol):
    def observe_by(self, observer: Observer) -> None:
        pass


class ObservableImpl(Observable):
    def __init__(self):
        self.__observers: Final[list[Observer]] = []

    def observe_by(self, observer: Observer):
        self.__observers.append(observer)
```

```
    def _notify_observers(self) -> None:
        for observer in self.__observers:
            observer.notify_about_change()


# Define Todo ...


class TodosModel(ObservableImpl):
    __todos: list[Todo]

    def __init__(self):
        super().__init__()
        self.__todos = []

    def add_todo(self, todo: Todo) -> None:
        self.__todos.append(todo)
        self._notify_observers()

    def remove_todo(self, todo: Todo) -> None:
        self.__todos.remove(todo)
        self._notify_observers()


class TodosView(Observer):
    def __init__(self, todos_model: TodosModel):
        self.__todos_model = todos_model
        todos_model.observe_by(self)

    def notify_about_change(self) -> None:
        # Will be called when todos model change
        self.render()

    def render(self) -> None:
        # Renders todos ...
```

Let's have another example that utilizes the publish-subscribe pattern. Below, we define a `MessageBroker` class that contains the following methods: `publish`, `subscribe`, and `unsubscribe`.

Figure 4.194

```
from collections.abc import Callable
from typing import Final, Protocol, TypeVar

T = TypeVar('T')


class MessagePublisher(Protocol[T]):
    def publish(self, topic: str, message: T) -> None:
        pass


class MessageSubscriber(Protocol[T]):
    def subscribe(
        self, topic: str, handle_message: Callable[[T], None]
    ) -> None:
        pass
```

```
MessageHandlers = list[Callable[[T], None]]


class MessageBroker(MessagePublisher[T], MessageSubscriber[T]):
    def __init__(self):
        self.__topic_to_handle_msgs: Final[dict[str,
MessageHandlers]] = {}

    def publish(self, topic: str, message: T) -> None:
        handle_messages = self.__topic_to_handle_msgs.get(topic)

        if handle_messages is not None:
            for handle_message in handle_messages:
                handle_message(message)

    def subscribe(
        self, topic: str, handle_message: Callable[[T], None]
    ) -> None:
        handle_messages = self.__topic_to_handle_msgs.get(topic)

        if handle_messages is None:
            self.__topic_to_handle_msgs[topic] =
[handle_message]
        else:
            handle_messages.append(handle_message)

    def unsubscribe(
        self, topic: str, handle_message: Callable[[T], None]
    ) -> None:
        handle_messages = self.__topic_to_handle_msgs.get(topic)

        if handle_messages is not None:
            handle_messages.remove(handle_message)


message_broker: MessageBroker[str] = MessageBroker()


def print_message(message: str):
    print(message)


topic = 'test'
message_broker.subscribe(topic, print_message)
message_broker.publish(topic, 'Hi!') # Prints 'Hi!'
message_broker.unsubscribe(topic, print_message)

# Does not print, because unsubscribed
message_broker.publish('test', 'Hi!')
```

4.13.3.3: Command/Action Pattern

Command or action pattern defines commands or actions as objects that can be given as parameters to other functions for later execution.

The command/action pattern is one way to follow the open-closed principle, i.e., extending code by creating new command/action classes for additional functionality instead of modifying existing code.

> Python full-stack developers familiar with React and Redux know that Redux requires implementing reducers, which are typically switch-case statements for executing a particular action. For improved object-oriented design, you could create a reducer that takes action objects as input and performs the action when it receives an action object. This approach is discussed in more detail in my other book *Clean Code Principles And Patterns: A Software Practitioner's Handbook*.

Let's create a simple action and command protocol:

Figure 4.195

```
from typing import Protocol


class Action(Protocol):
    def perform(self) -> None:
        pass


class Command(Protocol):
    def execute(self) -> None:
        pass
```

Let's create a simple concrete action/command that prints a message:

Figure 4.196

```
from typing import Final


class PrintAction(Action):
    def __init__(self, message: str):
        self.__message: Final = message

    def perform(self) -> None:
        print(self.__message)


class PrintCommand(Command):
    def __init__(self, message: str):
        self.__message: Final = message

    def execute(self) -> None:
        print(self.__message)
```

As can be seen, the above `PrintAction` and `PrintCommand` instances encapsulate the state that is used when the action/command is performed (usually at a later stage compared to action/command instance creation).

Now we can use our print action/command:

Figure 4.197

```
actions = [PrintAction('Hello'), PrintAction('World')]
for action in actions:
    action.perform()

commands = [PrintCommand('Hello'), PrintCommand('World')]
for command in commands:
    command.execute()
```

Using actions or commands makes it possible to follow the open-closed principle because every new action or command does not modify existing code but adds a new class.

Actions and commands can be made undoable, provided that the action/command is undoable. The above print action/command is not undoable because you cannot undo print to the console. Let's introduce an undoable action: add an item to a list. It is an action that can be undone by removing the item from the list.

Figure 4.198

```
from typing import Final, Generic, TypeVar

T = TypeVar('T')


class AddToListAction(Action, Generic[T]):
    def __init__(self, item: T, items: list[T]):
        self.__item: Final = item
        self.__items: Final = items

    def perform(self) -> None:
        self.__items.append(self.__item)

    def undo(self) -> None:
        try:
            self.__items.remove(self.__item)
        except ValueError:
            pass


values = [1, 2]
add3ToValuesAction = AddToListAction(3, values)
add3ToValuesAction.perform()
print(values)    # Prints [1, 2, 3]
```

```
add3ToValuesAction.undo()
print(values)    # Prints [1, 2]
```

4.13.3.4: Iterator Pattern

The iterator pattern can be used to add iteration capabilities to a class.

Let's create a reverse iterator for the Python's `list` class. We implement the `Iterator` abstract base class by supplying an implementation for the `__next__` method:

Figure 4.199

```
from collections.abc import Iterator
from typing import Final, TypeVar

T = TypeVar('T')


class ReverseListIterator(Iterator[T]):
    def __init__(self, values: list[T]):
        self.__values: Final = values.copy()
        self.__position = len(values) - 1

    def __next__(self) -> T:
        if self.__position < 0:
            raise StopIteration()
        next_value = self.__values[self.__position]
        self.__position -= 1
        return next_value

    def __iter__(self) -> Iterator[T]:
        return self
```

We can put the `ReverseListIterator` class into use in a `ReverseList` class defined below:

Figure 4.200

```
class ReverseList(list[T]):
    def  __iter__(self) -> Iterator[T]:
      return ReverseListIterator(self)
```

Now, we can use the new iterator to iterate over a list in reverse order:

Figure 4.201

```
reversed_numbers = ReverseList([1, 2, 3, 4, 5])

for number in reversed_numbers:
    print(number)
```

```
// Prints:
// 5
// 4
// 3
// 2
// 1
```

4.13.3.5: State Pattern

The state pattern lets an object change its behavior depending on its current state.

Developers don't often treat an object's state as an object but as an enumerated value (enum), for example. Below is an example where we have defined a `UserStory` class representing a user story that can be rendered on screen. An enum value represents the state of a `UserStory` object.

Figure 4.202

```
from enum import Enum
from typing import Protocol


class UserStoryState(Enum):
    TODO = 1
    IN_DEVELOPMENT = 2
    IN_VERIFICATION = 3
    READY_FOR_REVIEW = 4
    DONE = 5


class Icon(Protocol):
    # ...


class TodoIcon(Icon):
    # ...


# Define rest of icons ...


class UserStory:
    def __init__(self, name: str):
        self.__name = name
        self.__state = UserStoryState.TODO

    def set_state(self, state: UserStoryState) -> None:
        self.__state = state

    def render(self) -> None:
        match self.__state:
            case UserStoryState.TODO:
```

```
                icon = TodoIcon()
            case UserStoryState.IN_DEVELOPMENT:
                icon = InDevelopmentIcon()
            case UserStoryState.IN_VERIFICATION:
                icon = InVerificationIcon()
            case UserStoryState.READY_FOR_REVIEW:
                icon = ReadyForReviewIcon()
            case UserStoryState.DONE:
                icon = DoneIcon()
            case _:
                raise ValueError('Invalid user story state')

        # Draw a UI element on screen representing the user
story
        # using the above assigned 'icon'
```

The above solution is not an object-oriented one. We should replace the conditionals (match-case statement) with a polymorphic design. This can be done by introducing state objects. In the state pattern, the state of an object is represented with an object instead of an enum value. Below is the above code modified to use the state pattern:

Figure 4.203

```
from typing import Protocol

# Import Icon classes ...


class UserStoryState(Protocol):
    @property
    def icon(self) -> Icon:
        pass


class TodoUserStoryState(UserStoryState):
    @property
    def icon(self) -> Icon:
        return TodoIcon()


class InDevelopmentUserStoryState(UserStoryState):
    @property
    def icon(self) -> Icon:
        return InDevelopmentIcon()


class InVerificationUserStoryState(UserStoryState):
    @property
    def icon(self) -> Icon:
        return InVerificationIcon()


class ReadyForReviewUserStoryState(UserStoryState):
    @property
    def icon(self) -> Icon:
        return ReadyForReviewIcon()
```

```
class DoneUserStoryState(UserStoryState):
    @property
    def icon(self) -> Icon:
        return DoneIcon()


class UserStory:
    def __init__(self, name: str):
        self.__name = name
        self.__state = TodoUserStoryState()

    def set_state(self, state: UserStoryState) -> None:
        self.__state = state

    def render(self) -> None:
        icon = self.__state.icon
        # Draw a UI element on screen representing
        # the user story using the given 'icon'
```

Let's have another example with an `Order` class. An order can have a state, like paid, packaged, delivered, etc. Below, we implement the order states as classes:

Figure 4.204

```
from typing import Protocol

from Customer import Customer
from EmailService import EmailService


class OrderState(Protocol):
    def create_message(self, order_id: str) -> str:
        pass


class PaidOrderState(OrderState):
    def create_message(self, order_id: str) -> str:
        return f'Order {order_id} is successfully paid'


class DeliveredOrderState(OrderState):
    def create_message(self, order_id: str) -> str:
        return f'Order {order_id} is delivered'


# Implement the rest of possible order states ...


class Order:
    def __init__(self, id_: str, state: OrderState, customer:
Customer):
        self.__id = id_
        self.__state = state
        self.__customer = customer

    @property
    def customer_email_address(self) -> str:
```

```
        return self.__customer.email_address

    @property
    def state_message(self) -> str:
        return self.__state.create_message(self.__id)


email_service = EmailService(...)
order = Order(...)

email_service.send_email(
    order.customer_email_address,
    order.state_message
)
```

4.13.3.6: Mediator Pattern

> ***The mediator pattern lets you reduce dependencies between objects. It restricts direct communication between two different layers of objects and forces them to collaborate only via a mediator object or objects.***

The mediator pattern eliminates the coupling of two different layers of objects. So, changes to one layer of objects can be made without the need to change the objects in the other layer.

A typical example of the mediator pattern is the Model-View-Controller (MVC) pattern. In the MVC pattern, model and view objects do not communicate directly but only via mediator objects (controllers). Next, several ways to use the MVC pattern in frontend clients are presented. Traditionally, the MVC pattern was used in the backend when the backend also generated the view to be shown in the client device (web browser). With the advent of single-page web clients, a modern backend is a simple API containing only a model and controller (MC).

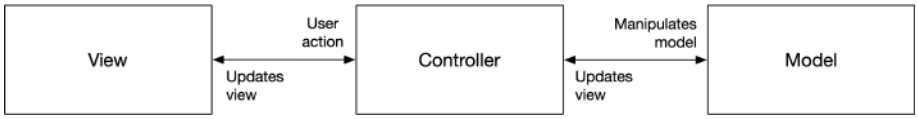

Figure 4.205. Model-View-Controller

In the below picture, you can see how dependency inversion is used, and none of the implementation classes depend on

concrete implementations. You can easily change any implementation class to a different one without the need to modify any other implementation class. Notice how the `ControllerImpl` class uses the *bridge pattern* and implements two bridges, one towards the model and the other towards the view.

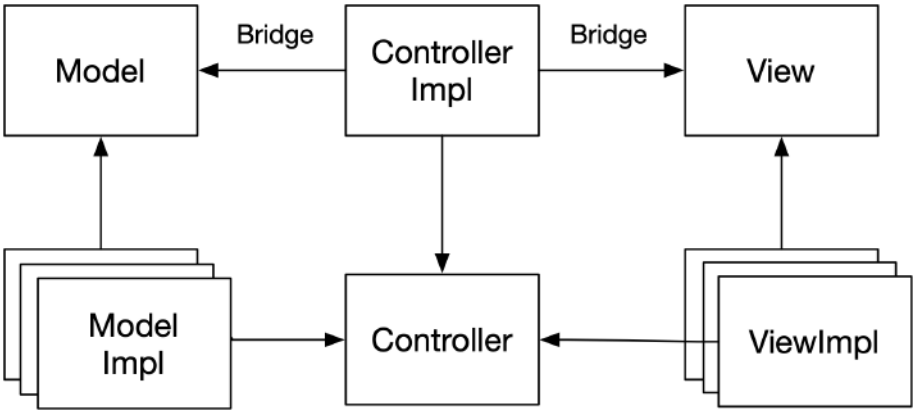

Figure 4.206. Dependencies in MVC pattern

The picture below shows that the controller can also be used as a bridge adapter. The controller can be modified to adapt to changes in the view layer (`View2` instead of `View`) without changing the model layer. The modified modules are shown with a gray background in the picture. Similarly, the controller can be modified to adapt to changes in the model layer without changing the view layer (not shown in the picture).

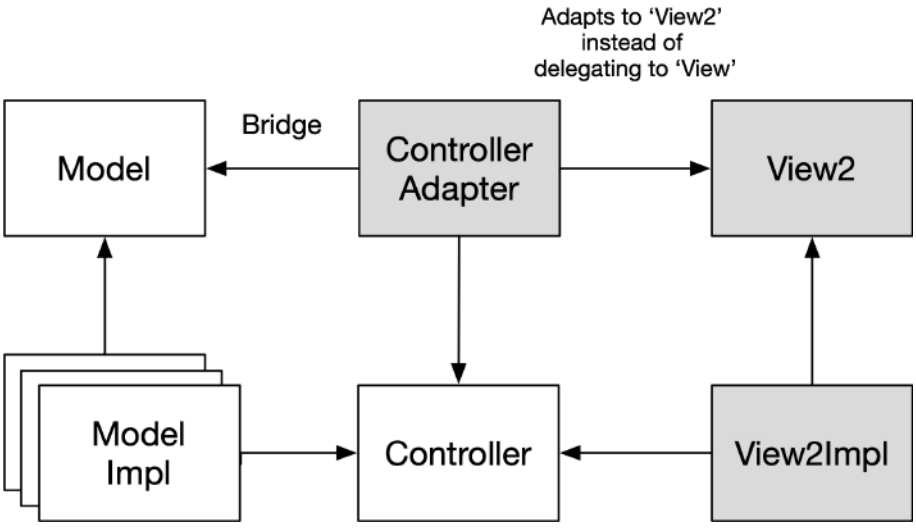

Figure 4.207. Adapting to Changes in MVC pattern

The following examples use a specialization of the MVC pattern called Model-View-Presenter (MVP). In the MVP pattern, the controller is called the presenter. I use the more generic term *controller* in all examples, though. A presenter acts as a middle-man between a view and a model. A presenter-type controller object has a reference to a view object and a model object. A view object commands the presenter to perform actions on the model. The model object asks the presenter to update the view object.

Let's have a simple todo application as an example. First, we implement the `Todo` class, which is part of the model.

Figure 4.208. Todo.py

```
class Todo:
    def __init__(self, id_: int, name: str, is_done: bool):
        self.__id = id_
        self.__name = name
        self.__is_done = is_done

    @property
    def id(self) -> int:
        return self.__id

    @id.setter
    def id(self, id_: int) -> None:
        self.__id = id

    @property
    def name(self) -> str:
        return self.__name

    @name.setter
    def name(self, name: str) -> None:
        self.__name = name

    @property
    def is_done(self) -> bool:
        return self.__is_done

    @is_done.setter
    def is_done(self, is_done: bool) -> None:
        self.__is_done = is_done
```

Next, we implement the protocol for the view layer:

Figure 4.209. TodoView.py

```
from typing import Protocol

from Todo import Todo


class TodoView(Protocol):
    def show_todos(self, todos: list[Todo]) -> None:
```

```
        pass

    def show_error_message(self, error_message: str) -> None:
        pass
```

Below is the implementation of the view layer:

Figure 4.210. TodoViewImpl.py

```
from typing import Final

from Todo import Todo
from TodoController import TodoController
from TodoView import TodoView


class TodoViewImpl(TodoView):
    def __init__(self, controller: TodoController):
        self.__controller: Final = controller
        controller.view = self
        controller.start_fetch_todos()

    def show_todos(self, todos: list[Todo]) -> None:
        # Update the view to show the given todos
        # Add listener for each todo checkbox
        # Listener calls:
self.__controller.toggle_todo_done(todo.id)

    def show_error_message(self, error_message: str) -> None:
        # Update the view to show error message
```

Then, we implement a generic `Controller` class that acts as a base class for concrete controllers:

Figure 4.211. Controller.py

```
from typing import Protocol, TypeVar

TModel = TypeVar('TModel')
TView = TypeVar('TView')


class Controller(Protocol[TModel, TView]):
    def __init__(self):
        self.__model: TModel | None = None
        self.__view: TView | None = None

    @property
    def model(self) -> TModel | None:
        return self.__model

    @model.setter
    def model(self, model: TModel) -> None:
        self.__model = model

    @property
    def view(self) -> TView | None:
        return self.__view

    @view.setter
```

```
    def view(self, view: TView) -> None:
        self.__view = view
```

The below `TodoControllerImpl` class implements two model-related actions, `start_fetch_todos` and `toggle_todo_done`, which delegate to the model layer. It also implements two view-related actions, `update_view_with_todos` and `update_view_with_error_message`, that delegate to the view layer.

Figure 4.212. TodoController.py

```
from typing import Protocol

from Todo import Todo


class TodoController(Protocol):
    async def start_fetch_todos(self) -> None:
        pass

    async def toggle_todo_done(self, id_: int) -> None:
        pass

    def update_view_with_todos(self, todos: list[Todo]) -> None:
        pass

    def update_view_with_error_message(self, error_message: str)
-> None:
        pass
```

Figure 4.213. TodoControllerImpl.py

```
from Controller import Controller
from Todo import Todo
from TodoController import TodoController
from TodoModel import TodoModel
from TodoView import TodoView


class TodoControllerImpl(Controller[TodoModel, TodoView],
TodoController):
    async def start_fetch_todos(self) -> None:
        if self.model is not None:
            await self.model.fetch_todos()

    async def toggle_todo_done(self, id_: int) -> None:
        if self.model is not None:
            await self.model.toggle_todo_done(id_)

    def update_view_with_todos(self, todos: list[Todo]) -> None:
        if self.view is not None:
            self.view.show_todos(todos)

    def update_view_with_error_message(self, error_message: str)
-> None:
        if self.view is not None:
            self.view.show_error_message(error_message)
```

The below `TodoModelImpl` class implements the fetching of todos (`fetch_todos`) using the supplied `todo_service`. The `todo_service` accesses the backend to read todos from a database, for example. The `TodoService` protocol is a counterpart for the same protocol in the backend API software component. When todos are successfully fetched, the controller is told to update the view. If fetching of the todos fails, the view is updated to show an error. Toggling a todo done is implemented using the `todo_service` and its `try_update_todo` method.

Figure 4.214. TodoService.py

```
from collections.abc import Awaitable
from typing import Protocol

from Todo import Todo


class TodoService(Protocol):
    class Error(Exception):
        # ...

    async def try_get_todos(self) -> Awaitable[list[Todo]]:
        pass

    async def try_update_todo(self, todo: Todo) -> None:
        pass
```

Figure 4.215. TodoModel.py

```
from typing import Protocol

class TodoModel(Protocol):
    async def fetch_todos(self) -> None:
        pass

    async def toggle_todo_done(self, id_: int) -> None:
        pass
```

Figure 4.216. TodoModelImpl.py

```
from typing import Final

from Todo import Todo
from TodoController import TodoController
from TodoModel import TodoModel
from TodoService import TodoService


class TodoModelImpl(TodoModel):
    __todos: list[Todo]

    def __init__(
        self, controller: TodoController, todo_service:
TodoService
    ):
```

```
        self.__controller: Final = controller
        controller.model = self
        self.__todo_service: Final = todo_service
        self.__todos = []

    async def fetch_todos(self) -> None:
        try:
            self.__todos = await
self.__todo_service.try_get_todos()
        except TodoService.Error as error:

self.__controller.update_view_with_error_message(error.message)
        else:

self.__controller.update_view_with_todos(self.__todos)

    async def toggle_todo_done(self, id_: int) -> None:
        found_todos = [todo for todo in self.__todos if todo.id
== id_]
        todo = found_todos[0] if found_todos else None

        if todo:
            todo.is_done = not todo.is_done

            try:
                await self.__todo_service.try_update_todo(todo)
            except TodoService.Error as error:

self.__controller.update_view_with_error_message(
                    error.message
                )
```

Let's make an exception to having all examples in Python and implement the above example using Web Components. If you are not a full-stack Python developer, you can skip the rest of this section because it will be frontend-related TypeScript code. The web component view should extend the `HTMLElement` class. The `connectedCallback` method of the view will be called on the component mount. It starts fetching todos. The `showTodos` method renders the given todos as HTML elements. It also adds event listeners for the *Mark done* buttons. The `showError` method updates the inner HTML of the view to show an error message.

Figure 4.217

```
export type Todo = {
  id: number;
  name: string;
  isDone: boolean;
};
```

Figure 4.218

```
interface TodoView {
  showTodos(todos: Todo[]): void;
  showError(errorMessage: string): void;
}
```

Figure 4.219

```
import controller from './todoController';
import { Todo } from './Todo';

export default class TodoViewImpl
         extends HTMLElement implements TodoView {
  constructor() {
    super();
    controller.setView(this);
  }

  connectedCallback() {
    controller.startFetchTodos();
    this.innerHTML = '<div>Loading todos...</div>';
  }

  showTodos(todos: Todo[]) {
    const todoElements = todos.map(({ id, name, isDone }) => `
      <li id="todo-${id}">
        ${id} ${name} 
        ${isDone ? '' : '<button>Mark done</button>'}
      </li>
    `);

    this.innerHTML = `<ul>${todoElements}</ul>`;

    todos.map(({ id }) => this
      .querySelector(`#todo-${id} button`)?
      .addEventListener('click',
                        () => controller.toggleTodoDone(id)));
  }

  showError(message: string) {
    this.innerHTML = `
      <div>
        Failure: ${message}
      </div>
    `;
  }
}
```

We can use the same controller and model APIs for this Web Component example as in the Python example. We need to convert the Python code to the respective TypeScript code:

Figure 4.220

```
export default class Controller<TModel, TView> {
  private model: TModel | undefined;
  private view: TView | undefined;

  getModel(): TModel | undefined {
    return this.model;
```

```
  }

  setModel(model: TModel): void {
    this.model = model;
  }

  getView(): TView | undefined {
    return this.view;
  }

  setView(view: TView): void {
    this.view = view;
  }
}
```

Figure 4.221

```
import { Todo } from "./Todo";

export interface TodoController {
  startFetchTodos(): void;
  toggleTodoDone(id: number): void;
  updateViewWithTodos(todos: Todo[]): void;
  updateViewWithError(message: string): void;
}
```

Figure 4.222

```
import TodoView from './TodoView';
import Controller from "./Controller";
import { TodoController } from './TodoController';
import { Todo } from "./Todo";
import TodoModel from './TodoModel';

class TodoControllerImpl
         extends Controller<TodoModel, TodoView>
         implements TodoController {

  startFetchTodos(): void {
    this.getModel()?.fetchTodos();
  }

  toggleTodoDone(id: number): void {
    this.getModel()?.toggleTodoDone(id);
  }

  updateViewWithTodos(todos: Todo[]): void {
    this.getView()?.showTodos(todos);
  }

  updateViewWithError(message: string): void {
    this.getView()?.showError(message);
  }
}

const controller = new TodoControllerImpl();
export default controller;
```

Figure 4.223

```
export interface TodoService {
  getTodos(): Promise<Todo[]>;
  updateTodo(todo: Todo): Promise<void>;
}
```

Figure 4.224

```
export interface TodoModel {
  fetchTodos(): void;
  toggleTodoDone(id: number): void;
}
```

Figure 4.225

```
import controller, { TodoController } from './todoController';
import { TodoModel } from './TodoModel';
import { Todo } from "./Todo";

export default class TodoModelImpl implements TodoModel {
  private todos: Todo[] = [];

  constructor(
    private readonly controller: TodoController,
    private readonly todoService: TodoService
  ) {
    controller.setModel(this);
  }

  fetchTodos(): void {
    this.todoService.getTodos()
      .then((todos) => {
        this.todos = todos;
        controller.updateViewWithTodos(todos);
      })
      .catch((error) =>
        controller.updateViewWithError(error.message));
  }

  toggleTodoDone(id: number): void {
    const foundTodo = this.todos.find(todo => todo.id === id);

    if (foundTodo) {
      foundTodo.isDone = !foundTodo.isDone;
      this.todoService
          .updateTodo(foundTodo)
          .catch((error: any) =>
            controller.updateViewWithError(error.message));
    }
  }
}
```

We could use the above-defined controller and model as such with a React view component:

Figure 4.226

```
// ...
import controller from './todoController';

// ...
```

```
export default class ReactTodoView
        extends Component<Props, State>
        implements TodoView {

  constructor(props: Props) {
    super(props);
    controller.setView(this);

    this.state = {
      todos: []
    }
  }

  componentDidMount() {
    controller.startFetchTodos();
  }

  showTodos(todos: Todo[]) {
    this.setState({ ...this.state, todos });
  }

  showError(errorMessage: string) {
     this.setState({ ...this.state, errorMessage });
  }

  render() {
    // Render todos from 'this.state.todos' here
    // Or show 'this.state.errorMessage' here
  }
}
```

If you have multiple views using the same controller, you can derive your controller from the below-defined `MultiViewController` class:

Figure 4.227

```
export default class MultiViewController<TModel, TView> {
  private model: TModel | undefined;
  private views: TView[] = [];

  getModel(): TModel | undefined {
    return this.model;
  }

  setModel(model: TModel): void {
    this.model = model;
  }

  getViews(): TView[]  {
    return this.views;
  }

  addView(view: TView): void {
    this.views.push(view);
  }
}
```

Let's say we want to have two views for todos, one for the actual todos and one viewing the todo count. We need to modify the controller slightly to support multiple views:

Figure 4.228

```
import TodoView from './TodoView';
import MultiViewController from './MultiViewController';
import { Todo } from "./Todo";
import { TodoController } from './TodoController';
import TodoModel from './TodoModel';

class TodoControllerImpl
         extends MultiViewController<TodoModel, TodoView>
         implement TodoController {
  startFetchTodos(): void {
    this.getModel()?.fetchTodos();
  }

  toggleTodoDone(id: number): void {
    this.getModel()?.toggleTodoDone(id);
  }

  updateViewsWithTodos(todos: Todo[]): void {
    this.getViews().forEach(view => view.showTodos(todos));
  }

  updateViewWithError(message: string): void {
    this.getViews().forEach(view => view.showError(message));
  }
}

const controller = new TodoController();
export default controller;
```

Many modern UI frameworks and state management libraries implement a specialization of the MVC pattern called, Model-View-ViewModel (MVVM). In the MVVM pattern, the controller is called the view model. I use the more generic term *controller* in the below example, though. The main difference between the view model and the presenter in the MVP pattern is that in the MVP pattern, the presenter has a reference to the view, but the view model does not. The view model provides bindings between the view's events and actions in the model. This can happen so that the view model adds action dispatcher functions as properties of the view. In the other direction, the view model maps the model's state to the properties of the view. For example, when using React and Redux, you can connect the view to the model using the `mapDispatchToProps` function and connect the model to the view using the `mapStateToProps` function. These two mapping

functions form the view model (or the controller) that binds the view and model together.

Let's first implement the todo example with React and Redux and later show how the React view can be replaced with an Angular view without modifying the controller or the model layer.

Let's implement a list view for todos:

Figure 4.229

```
import { connect } from 'react-redux';
import { useEffect } from "react";
import { controller, ActionDispatchers, State }
  from './todosController';

type Props = ActionDispatchers & State;

function TodosListView({
  toggleTodoDone,
  startFetchTodos,
  todos
}: Props) {

  useEffect(() => {
    startFetchTodos();
  }, [startFetchTodos]);

  const todoElements = todos.map(({ id, name, isDone }) => (
    <li key={id}>
      {id} 
      {name} 
      {isDone
        ? undefined
        : <button onClick={() => toggleTodoDone(id)}>
            Mark done
          </button>
      }
    </li>
  ));

  return <ul>{todoElements}</ul>;
}

// Here we connect the view to the model using the controller
export default connect(
  controller.getState,
  () => controller.actionDispatchers
)(TodosListView);
```

Below is the base class `AbstractAction` for Redux actions and the base class `Controller` for controllers:

Figure 4.230

```
export default abstract class AbstractAction<S> {
  abstract perform(state: S): S;
```

```
}
```

Figure 4.231

```
import AbstractAction from "./AbstractAction";

export type ReduxDispatch =
 (reduxActionObject: { type: AbstractAction<any> }) => void;

export default class Controller {
  protected readonly dispatch:
    (action: AbstractAction<any>) => void;

  constructor(reduxDispatch: ReduxDispatch) {
    this.dispatch = (action: AbstractAction<any>) =>
      reduxDispatch({ type: action });
  }
}
```

Below is the controller for todos:

Figure 4.232

```
import store from './store';
import { AppState } from "./AppState";
import ToggleDoneTodoAction from "./ToggleDoneTodoAction";
import StartFetchTodosAction from "./StartFetchTodosAction";
import Controller from "./Controller";

class TodosController extends Controller {
  readonly actionDispatchers = {
    toggleTodoDone: (id: number) =>
      this.dispatch(new ToggleDoneTodoAction(id)),

    startFetchTodos: () =>
      this.dispatch(new StartFetchTodosAction())
  }

  getState(appState: AppState) {
    return {
      todos: appState.todosState.todos,
    }
  }
}

export const controller = new TodosController(store.dispatch);
export type State = ReturnType<typeof controller.getState>;
export type ActionDispatchers = typeof
controller.actionDispatchers;
```

In the development phase, we can use the following temporary implementation of the `StartFetchTodosAction` class:

Figure 4.233

```
import { Todo } from './Todo';

export type TodoState = {
  todos: Todo[];
}
```

```
const initialTodosState = {
  todos: []
} as TodoState

export default initialTodosState;
```

Figure 4.234

```
import AbstractAction from './AbstractAction';

export default abstract class AbstractTodoAction extends
  AbstractAction<TodoState> {}
```

Figure 4.235

```
import { TodoState } from "./TodoState";
import AbstractTodoAction from "./AbstractTodoAction";

export default class StartFetchTodosAction extends
                        AbstractTodoAction {
  perform(state: TodoState): TodoState {
    return {
      todos: [
        {
          id: 1,
          name: "Todo 1",
          isDone: false,
        },
        {
          id: 2,
          name: "Todo 2",
          isDone: false,
        },
      ],
    };
  }
}
```

Figure 4.236

```
import AbstractTodoAction from './AbstractTodoAction';

export default class MarkDoneTodoAction extends
AbstractTodoAction {
  // ...
}
```

Now we can introduce a new view for todos, a `TodosTableView` which can utilize the same controller as the `TodosListView`.

Figure 4.237

```
import { connect } from 'react-redux';
import { useEffect } from "react";
import { controller, ActionDispatchers, State }
  from './todosController';
```

```
type Props = ActionDispatchers & State;

function TodosTableView({
  toggleTodoDone,
  startFetchTodos,
  todos
}: Props) {
  useEffect(() => {
    startFetchTodos();
  }, [startFetchTodos]);

  const todoElements = todos.map(({ id, isDone, name }) => (
    <tr key={id}>
      <td>{id}</td>
      <td>{name}</td>
        <td>
          <input
            type="checkbox"
            checked={isDone}
            onChange={() => toggleTodoDone(id)}
          />
        </td>
    </tr>
  ));

  return <table><tbody>{todoElements}</tbody></table>;
}

export default connect(
  controller.getState,
  () => controller.actionDispatchers
)(TodosTableView);
```

We can notice some duplication in the `TodosListView` and `TodosTableView` components. For example, both are using the same effect. We can create a `TodosView` for which we can give as parameter the type of a single todo view, either a list item or a table row view:

Figure 4.238

```
import { useEffect } from "react";
import { connect } from "react-redux";
import ListItemTodoView from './ListItemTodoView';
import TableRowTodoView from './TableRowTodoView';
import { controller, ActionDispatchers, State }
  from './todosController';

type Props = ActionDispatchers & State & {
    TodoView: typeof ListItemTodoView | typeof TableRowTodoView;
};

function TodosView({
  toggleTodoDone,
  startFetchTodos,
  todos,
  TodoView
}: Props) {
  useEffect(() => {
```

```
    startFetchTodos()
  }, [startFetchTodos]);

  const todoViews = todos.map((todo) =>
    <TodoView
      key={todo.id}
      todo={todo}
      toggleTodoDone={toggleTodoDone}
    />
  );

  return TodoView === ListItemTodoView
      ? <ul>{todoViews}</ul>
      : <table><tbody>{todoViews}</tbody></table>;
}

export default connect(
  controller.getState,
  () => controller.actionDispatchers
)(TodosView);
```

Below is the view for showing a single todo as a list item:

Figure 4.239

```
import { Todo } from "./Todo";

export type TodoViewProps = {
  toggleTodoDone: (id: number) => void,
  todo: Todo
}
```

Figure 4.240

```
import { TodoViewProps } from './TodoViewProps';

export default function ListItemTodoView({
  toggleTodoDone,
  todo: { id, name, isDone }
}: TodoViewProps) {
  return (
    <li>
      {id} 
      {name} 
      { isDone ?
        undefined :
        <button onClick={() => toggleTodoDone(id)}>
          Mark done
        </button> }
    </li>
  );
}
```

Below is the view for showing a single todo as a table row:

Figure 4.241

```
import { TodoViewProps } from './TodoViewProps';

export default function TableRowTodoView({
```

```
  toggleTodoDone,
  todo: { id, name, isDone }
}: TodoViewProps) {
  return (
    <tr>
      <td>{id}</td>
      <td>{name}</td>
      <td>
        <input
          type="checkbox"
          checked={isDone}
          onChange={() => toggleTodoDone(id)}
        />
      </td>
    </tr>);
}
```

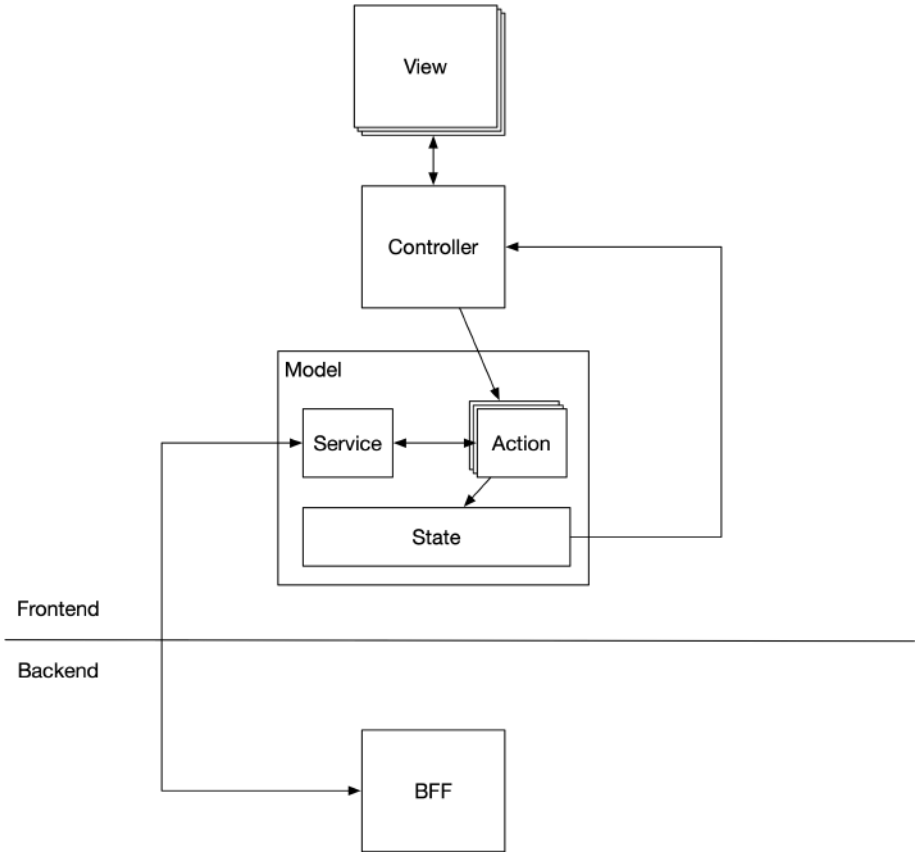

Figure 4.242. Frontend MVC Architecture with Redux

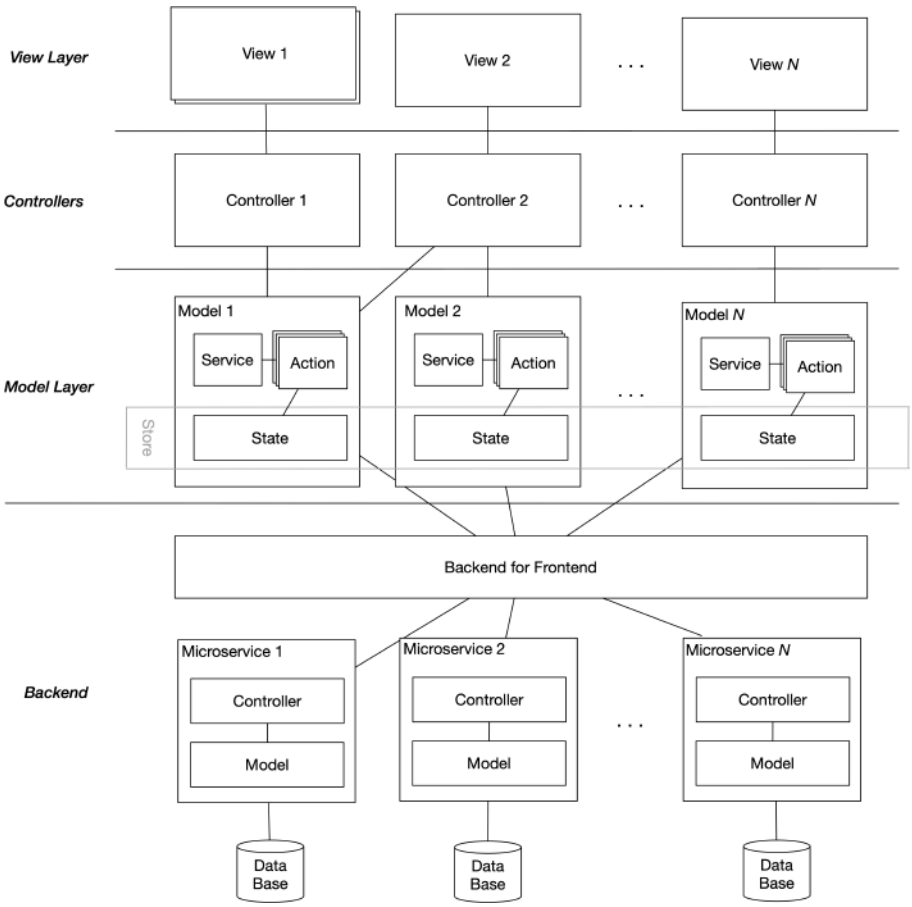

Figure 4.243. Frontend MVC Architecture with Redux + Backend

In many cases, you should not store the state in a view, even if the state is for that particular view only. Instead, when you store it in the model, it brings the following benefits:

- Possibility to easily persist state either in the browser or in the backend
- Possibility to easily implement undo-actions
- State can be easily shared with another view(s) later if needed
- Migrating views to use a different view technology is more straightforward
- Easier debugging of state-related problems, e.g., using the Redux DevTools browser extension

We can also change the view implementation from React to Angular without modifying the controller or model layer. This

can be done, for example, using the @angular-redux2/store library. Below is a todos table view implemented as an Angular component:

Figure 4.244

```
import { Component, OnInit } from "@angular/core";
import { NgRedux, Select } from '@angular-redux2/store';
import { Observable } from "rxjs";
import { controller } from './todosController';
import { TodoState } from "./TodoState";
import { AppState } from "./AppState";

const { startFetchTodos,
        toggleTodoDone } = controller.actionDispatchers;

@Component({
  selector: 'todos-table-view',
  template: `
    <table>
      <tr *ngFor="let todo of (todoState | async)?.todos">
        <td>{{ todo.id }}</td>
        <td>{{ todo.name }}</td>
        <td>
          <input
            type="checkbox"
            [checked]="todo.isDone"
            (change)="toggleTodoDone(todo.id)"
          />
        </td>
      </tr>
    </table>
  `
})
export class TodosTableView implements OnInit {
  @Select(controller.getState) todoState: Observable<TodoState>;

  constructor(private ngRedux: NgRedux<AppState>) {}

  ngOnInit(): void {
    startFetchTodos();
  }

  toggleTodoDone(id: number) {
    toggleTodoDone(id);
  }
}
```

Figure 4.245

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
  <div>
    <todos-table-view></todos-table-view>
  </div>`,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
```

```
  title = 'angular-test';
}
```

Figure 4.246

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { NgReduxModule, NgRedux } from '@angular-redux2/store';

import { AppComponent } from './app.component';
import store from './store';
import { AppState } from "./AppState";
import { TodosTableView } from "./todos-table-view.component";

@NgModule({
  declarations: [
    AppComponent, TodosTableView
  ],
  imports: [
    BrowserModule,
    NgReduxModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
  constructor(ngRedux: NgRedux<AppState>) {
    ngRedux.provideStore(store);
  }
}
```

4.13.3.7: Template Method Pattern

> ***Template method pattern allows you to define a template method in a base class, and subclasses define the final implementation of that method. The template method contains one or more calls to abstract methods implemented in the subclasses.***

In the below example, the `AbstractDrawing` class contains a template method, `draw`. This method includes a call to the `_get_shape_renderer` method, an abstract method implemented in the subclasses of the `AbstractDrawing` class. The `draw` method is a template method, and a subclass defines how to draw a single shape. The template method should always be marked as final using the `@final` decorator. The subclasses should never override it. They should implement the abstract methods.

Figure 4.247

```
from abc import abstractmethod
from typing import Final, Protocol

from Shape import Shape
from ShapeRenderer import ShapeRenderer


class Drawing(Protocol):
    def draw(self) -> None:
        pass


class AbstractDrawing(Drawing):
    def __init__(self, shapes: list[Shape]):
        self.__shapes: Final = shapes

    @final
    def draw(self) -> None:
        for shape in self.__shapes:
            shape.render(self._get_shape_renderer())

    @abstractmethod
        def _get_shape_renderer(self) -> ShapeRenderer:
            pass
```

We can now implement two subclasses of the `AbstractDrawing` class, which define the final behavior of the templated `draw` method.

Figure 4.248

```
from typing import Final

from AbstractDrawing import AbstractDrawing
from Canvas import Canvas
from RasterShapeRenderer import RasterShapeRenderer
from Shape import Shape
from ShapeRenderer import ShapeRenderer
from SvgElement import SvgElement
from VectorShapeRenderer import VectorShapeRenderer


class RasterDrawing(AbstractDrawing):
    def __init__(self, shapes: list[Shape]):
        super().__init__(shapes)
        canvas = Canvas()
        self.__shape_renderer: Final =
RasterShapeRenderer(canvas)

    def _get_shape_renderer(self) -> ShapeRenderer:
        return self.__shape_renderer


class VectorDrawing(AbstractDrawing):
    def __init__(self, shapes: list[Shape]):
        super().__init__(shapes)
        svg_root = SvgElement()
        self.__shape_renderer: Final =
VectorShapeRenderer(svg_root)
```

```
    def _get_shape_renderer(self) -> ShapeRenderer:
        return self.__shape_renderer
```

4.13.3.8: Memento Pattern

The memento pattern can be used to save the internal state of an object to another object called the memento object.

Let's have an example with a `TextEditor` class. First, we define a `TextEditorState` protocol and its implementation. Then, we define a `TextEditorStateMemento` class for storing a memento of the text editor's state.

Figure 4.249

```
from typing import Protocol


class TextEditorState(Protocol):
    def clone(self) -> 'TextEditorState':
        pass


class TextEditorStateImpl(TextEditorState):
    # Implement text editor state here ...


class TextEditorStateMemento:
    def __init__(self, state: TextEditorState):
        self.__state = state.clone()

    @property
    def state(self) -> TextEditorState:
        return self.__state
```

The `TextEditor` class stores mementos of the text editor's state. It provides methods to save a state, restore a state, or restore the previous state:

Figure 4.250

```
from typing import Final

from TextEditorStateImpl import TextEditorStateImpl
from TextEditorStateMemento import TextEditorStateMemento


class TextEditor:
    def __init__(self):
        self.__state_mementos:
Final[list[TextEditorStateMemento]] = []
```

```
        self.__current_state = TextEditorStateImpl(...)
        self.__current_version = 1

    def save_state(self) -> None:
        self.__state_mementos.append(
            TextEditorStateMemento(self.__current_state)
        )

        self.__current_version += 1

    def restore_state(self, version: int) -> None:
        if 1 <= version <= len(self.__state_mementos):
            self.__current_state = self.__state_mementos[version
- 1].state
            self.__current_version += 1

    def restore_previous_state(self) -> None:
        if self.__current_version > 1:
            self.restore_state(self.__current_version - 1)
```

In the above example, we can add a memento for the text editor's state by calling the `save_state` method. We can recall the previous version of the text editor's state using the `restore_previous_state` method and any version of the text editor's state using the `restore_state` method.

4.13.3.9: Visitor Pattern

> ***Visitor pattern allows adding functionality to a class (like adding new methods) without modifying the class. This is useful, for example, with library classes that you cannot modify.***

First, let's have an example with classes that we can modify:

Figure 4.251

```
from typing import Final, Protocol


class Shape(Protocol):
    def draw(self) -> None:
        pass


class CircleShape(Shape):
    def __init__(self, radius: int):
        self.__radius: Final = radius

    def draw(self) -> None:
        # Draw circle ...

    @property
```

```
    def radius(self) -> int:
        return self.__radius


class RectangleShape(Shape):
    def __init__(self, width: int, height: int):
        self.__width: Final = width
        self.__height: Final = height

    def draw(self) -> None:
        # Draw rectangle ...

    @property
    def width(self) -> int:
        return self.__width

    @property
    def height(self) -> int:
        return self.__height
```

Let's assume we need to calculate the total area of shapes in a drawing. Currently, we are in a situation where we can modify the shape classes, so let's add `calculate_area` methods to the classes:

Figure 4.252

```
import math
from typing import Protocol


class Shape(Protocol):
    # ...

    def calculate_area(self) -> float:
        pass


class CircleShape(Shape):
    # ...

    def calculate_area(self) -> float:
        return math.pi * self.__radius**2


class RectangleShape(Shape):
    # ...

    def calculate_area(self) -> float:
        return self.__width * self.__height
```

Adding a new method to an existing class may be against the *open-closed principle*. In the above case, adding the `calculate_area` methods is safe because the shape classes are immutable. And even if they were not, adding the `calculate_area` methods would be safe because they are

read-only, i.e., they don't modify the object's state. Also, we don't have to worry about thread safety if we agree that our example application is not multithreaded.

Now that we have the area calculation methods added, we can use a common algorithm to calculate the total area of shapes in a drawing:

Figure 4.253

```
from functools import reduce

shapes = [CircleShape(1), RectangleShape(2, 2)]

total_shapes_area = reduce(
    lambda accum_shapes_area, shape: accum_shapes_area +
shape.calculate_area(),
    shapes,
    0.0
)

print(total_shapes_area) # Prints 7.141592653589793
```

But what if the shape classes, without the area calculation capability, were in a 3rd party library that we cannot modify? We would have to do something like this:

Figure 4.254

```
import math
from functools import reduce

def shapes_area(accum_shapes_area: float, shape: Shape) ->
float:
    if isinstance(shape, CircleShape):
        shape_area = math.pi * shape.radius**2
    elif isinstance (shape, RectangleShape):
        shape_area = shape.width * shape.height
    else:
        raise ValueError('Invalid shape')

    return accum_shapes_area + shape_area


total_shapes_area = reduce(
    shapes_area,
    shapes,
    0.0
)
```

The above solution is complicated and needs updating every time a new type of shape is introduced. The above example does not follow object-oriented design principles: it contains

an if/elif structure with `isinstance` checks, called the *chain of instanceof checks* code smell.

We can use the visitor pattern to replace the above conditionals with polymorphism. First, we introduce a visitor protocol that can be used to provide additional behavior to the shape classes. Then, we introduce an `execute` method in the `Shape` protocol. In the shape classes, we implement the `execute` methods so that additional behavior provided by a concrete visitor can be executed:

Figure 4.255

```
from typing import Any, Final, Protocol


# This is our visitor protocol that
# provides additional behaviour to the shape classes
class ShapeBehavior(Protocol):
    def execute_for_circle(self, circle: 'CircleShape') -> Any:
        pass

    def execute_for_rectangle(self, rectangle: 'RectangleShape')
-> Any:
        pass

    # Add methods for possible other shape classes here ...


class Shape(Protocol):
    # ...

    def execute(self, behavior: ShapeBehavior) -> Any:
        pass


class CircleShape(Shape):
    def __init__(self, radius: int):
        self.__radius: Final = radius

    @property
    def radius(self) -> int:
        return self.__radius

    def execute(self, behavior: ShapeBehavior) -> Any:
        return behavior.execute_for_circle(self)


class RectangleShape(Shape):
    def __init__(self, width: int, height: int):
        self.__width: Final = width
        self.__height: Final = height

    @property
    def width(self) -> int:
        return self.__width

    @property
```

```
    def height(self) -> int:
        return self.__height

    def execute(self, behavior: ShapeBehavior) -> Any:
        return behavior.execute_for_rectangle(self)
```

Suppose our application is multithreaded and the shape classes were mutable and made thread-safe. We would have to define the `execute` methods with appropriate synchronization to make them also thread-safe:

Figure 4.256

```
from threading import Lock


class CircleShape(Shape):
    # Constructor that initializes
    # self.__lock = Lock()

    def execute(self, behavior: ShapeBehaviour) -> Any:
        with self.__lock:
            return behavior.execute_for_circle(self)

    # Rest of methods ...


class RectangleShape(Shape):
    # Constructor that initializes
    # self.__lock = Lock()

    def execute(self, behavior: ShapeBehaviour) -> Any:
        with self.__lock:
            return behavior.execute_for_rectangle(self)

    # Rest of methods ...
```

Let's implement a concrete visitor for calculating areas of different shapes:

Figure 4.257

```
class AreaCalculationShapeBehavior(ShapeBehavior):
    def execute_for_circle(self, circle: CircleShape) -> Any:
        return math.pi * circle.radius**2

    def execute_for_rectangle(self, rectangle: RectangleShape) -
> Any:
        return rectangle.width * rectangle.height
```

Now, we can implement the calculation of shapes' total area by using a common algorithm, and we get rid of the conditionals. Below, we execute the `area_calculation` behavior for each shape.

Figure 4.258

```
shapes = [CircleShape(1), RectangleShape(2, 2)]
area_calculation = AreaCalculationShapeBehavior()

total_shapes_area = reduce(
    lambda accum_shapes_area, shape:
        accum_shapes_area + shape.execute(area_calculation),
    shapes,
    0.0
)
```

You can add more behavior to the shape classes by defining a new visitor. Let's define a `PerimeterCalculationShapeBehaviour` class:

Figure 4.259

```
class PerimeterCalculationShapeBehavior(ShapeBehavior):
    def execute_for_circle(self, circle: CircleShape) -> Any:
        return 2 * math.pi * circle.radius

    def execute_for_rectangle(self, rectangle: RectangleShape) -
> Any:
        return 2 * rectangle.width + 2 * rectangle.height
```

Notice that we did not need to use the *visitor* term in our code examples. Adding the design pattern name to the names of software entities (class/function names, etc.) often does not bring any real benefit but makes the names longer. However, there are some design patterns, like the *factory pattern* and *builder pattern* where you always use the design pattern name in a class name.

If you develop a third-party library and want the behavior of its classes to be extended by its users, you should make your library classes accept visitors who can perform additional behavior. Using the visitor pattern allows to add behavior to existing classes without modifying them, i.e., in accordance with the open-closed principle. However, there is one drawback to using the visitor pattern. You must create getters and setters for class attributes to allow visitors to add behavior. Adding getters and setters breaks the class encapsulation, as was discussed earlier in this chapter.

4.13.3.10: Null Object Pattern

A null object is an object that does nothing.

Use the null object pattern to implement a class for null objects that don't do anything. A null object can be used in place of a real object that does something.

Let's have an example with a `Shape` protocol:

Figure 4.260

```
from typing import Protocol


class Shape(Protocol):
    def draw(self) -> None:
        pass
```

We can easily define a class for *null* shape objects:

Figure 4.261

```
from Shape import Shape


class NullShape(Shape):
    def draw(self) -> None:
        # Intentionally no operation
```

A *null* shape does not draw anything. We can use an instance of the `NullShape` class everywhere where a concrete implementation of the `Shape` protocol is wanted.

4.14: Don't Ask, Tell Principle

> ***Don't ask, tell principle states that you should tell another object what to do, and not ask about the other object's state and then do the work by yourself in your object.***

If your object asks many things from another object using, e.g., multiple getters, you might be guilty of the *feature envy* design smell. Your object is envious of a feature that the other object should have.

Let's have an example and define a cube shape class:

Figure 4.262

```
from typing import Final, Protocol


class ThreeDShape(Protocol):
```

```
    # ...


class Cube3DShape(ThreeDShape):
    def __init__(self, width: int, height: int, depth: int):
        self.__width: Final = width
        self.__height: Final = height
        self.__depth: Final = depth

    @property
    def width(self, ) -> int:
        return self.__width

    @property
    def height(self) -> int:
        return self.__height

    @property
    def depth(self) -> int:
        return self.__depth
```

Next, we define another class, `CubeUtils`, that contains a method for calculating the total volume of cubes:

Figure 4.263

```
from typing import final

from Cube3DShape import Cube3DShape


@final
class CubeUtils:
    @staticmethod
    def calculate_total_volume(cubes: list[Cube3DShape]) -> int:
        total_volume = 0
        for cube in cubes:
            total_volume += cube.width * cube.height *
cube.depth
        return total_volume
```

In the `calculate_total_volume` method, we ask about a cube object's state three times. This is against the *don't ask, tell principle*. Our method is envious of the volume calculation feature and wants to do it by itself rather than telling a `Cube3DShape` object to calculate its volume.

Let's correct the above code so that it follows the *don't ask, tell principle*:

Figure 4.264

```
from typing import Final, Protocol, final


class ThreeDShape(Protocol):
    def calculate_volume(self) -> int:
```

```
        pass


class Cube3DShape(ThreeDShape):
    def __init__(self, width: int, height: int, depth: int):
        self.__width: Final = width
        self.__height: Final = height
        self.__depth: Final = depth

    def calculate_volume(self) -> int:
        return self.__height * self.__width * self.__depth


@final
class ThreeDShapeUtils:
    @staticmethod
    def calculate_total_volume(three_d_shapes:
list[ThreeDShape]) -> int:
        total_volume = 0
        for three_d_shape in three_d_shapes:
            total_volume += three_d_shape.calculate_volume()
        return total_volume
```

Our `calculate_total_volume` method does not ask anything about a cube object. It just tells a cube object to calculate its volume. We also removed the *asking* methods (getters/properties) from the `Cube3DShape` class because they are no longer needed.

Below is another example of asking instead of telling:

Figure 4.265

```
import time


class AnomalyDetectionEngine:
    def run(self) -> None:
        while self.__is_running:
            now = time.time()

            if
self.__anomaly_detector.anomalies_should_be_detected(now):
                anomalies =
self.__anomaly_detector.detect_anomalies()
                # Do something with the detected anomalies ...

            time.sleep(1)
```

In the above example, we ask the anomaly detector if we should detect anomalies now. Then, depending on the result, we call another method on the anomaly detector to detect anomalies. This could be simplified by making the `detect_anomalies` method to check if anomalies should be

detected using the `anomalies_should_be_detected` method. Then, the `anomalies_should_be_detected` method can be made private, and we can simplify the above code as follows:

Figure 4.266

```
class AnomalyDetectionEngine:
    def run(self) -> None:
        while self.__is_running:
            anomalies =
self.__anomaly_detector.detect_anomalies()
            # Do something with the detected anomalies ...
            time.sleep(1)
```

4.15: Law of Demeter

> ***A method on an object received from another object's method call should not be called.***

The below statements are considered to break the law because of the chained method calls:

Figure 4.267

```
user.get_account().get_balance()
user.get_account().withdraw(...)
```

The above statements can be corrected by moving functionality to a different class or by making the second object to act as a facade (remember the *facade pattern*) between the first and the third object.

Below is an example of the latter solution, where we introduce two new methods in the `User` class and remove the `get_account` method:

Figure 4.268

```
user.get_account_balance()
user.withdraw_from_account(...)
```

In the above example, the `User` class is a facade in front of the `Account` class that we should not access directly from our object.

However, you should always check if the first solution alternative could be used instead. It makes the code more

object-oriented and does not require creating additional methods.

Below is an example that uses `User` and `SalesItem` entities and is not obeying the law of Demeter:

Figure 4.269

```
from SalesItem import SalesItem
from User import User


def purchase(user: User, sales_item: SalesItem) -> None:
    account = user.get_account()

    # Breaks the law
    account_balance = account.get_balance()

    sales_item_price = sales_item.get_price()

    if account_balance >= sales_item_price:
        # Breaks the law
        account.withdraw(sales_item_price);

    # ...
```

We can resolve the problem in the above example by moving the `purchase` method to the correct class, in this case, the `User` class:

Figure 4.270

```
from Account import Account
from SalesItem import SalesItem


class User:
    def __init__(self, account: Account):
        self.__account = account

    def purchase(self, sales_item: SalesItem) -> None:
        account_balance = self.__account.get_balance()
        sales_item_price = sales_item.get_price()

        if account_balance >= sales_item_price:
            self.__account.withdraw(sales_item_price)

        # ...
```

4.16: Avoid Primitive Obsession Principle

Avoid primitive obsession by defining semantic types for function parameters and function return value.

Some of us have experienced a situation where we have supplied arguments to a function in the wrong order. This can be easy if the function, for example, takes two integer parameters, but you accidentally give those two integer parameters in the wrong order. You don't get a compilation error.

Another problem with primitive types as function arguments is that the argument values are not necessarily validated. You may have to implement the validation logic in your function. However, the validation should typically happen when reading untrusted input and only already validated values are supplied to other functions in the software component.

Suppose you accept an integer parameter for a port number in a function. In that case, you might get any integer value as the parameter value, even though the valid port numbers are from 1 to 65535. Suppose you also had other functions in the same codebase accepting a port number as a parameter. In that case, you could do the same validation logic in multiple places and have duplicate code in your codebase.

Let's have a simple example of using this principle:

Figure 4.271

```
from Shape import Shape


class RectangleShape(Shape):
    def __init__(self, width: int, height: int):
        self.__width = width
        self.__height = height
```

In the above example, the constructor has two parameters with the same primitive type (int). It is possible to give `width` and `height` in the wrong order. But if we refactor the code to use objects instead of primitive values, we can make the likelihood of giving the arguments in the wrong order much smaller:

Figure 4.272

```
from typing import TypeVar, Generic, Final

from Shape import Shape
```

```
T = TypeVar('T')


class Value(Generic[T]):
    def __init__(self, value: T):
        self.__value: Final = value

    @property
    def value(self) -> T:
        return self.__value


class Width(Value[int]):
    pass


class Height(Value[int]):
    pass


class RectangleShape(Shape):
  def __init__(self, width: Width, height: Height):
          self.__width = width.value
          self.__height = height.value


width = Width(20)
height = Height(50)

# OK
rectangle = RectangleShape(width, height)

# Does not pass type check, parameters are in wrong order
rectangle2 = RectangleShape(height, width)

# Does not pass type check, first parameter is not a width
rectangle3 = RectangleShape(height, height)

# Does not pass type check, second parameter is not a height
rectangle4 = RectangleShape(width, width)

# Does not pass type check,
# Width and Height objects must be used
# instead of primitive types
rectangle5 = RectangleShape(20, 50)
```

In the above example, `Width` and `Height` are simple data classes. They don't contain any behavior. You can use concrete data classes as function parameter types. There is no need to create an interface for a data class. So, the *program against interfaces* principle does not apply here.

In Python, we have another way to safeguard against giving the same type of parameters in the wrong order: named

parameters. Without the named parameters, the creation of a new rectangle would look like the following:

Figure 4.273

```
rectangle = RectangleShape(20, 50)
```

In the above example, we must be sure that the first parameter is width and the second is height. When using named parameters, we don't have to remember the correct order of the parameters:

Figure 4.274

```
rectangle = RectangleShape(width=20, height=50)
rectangle2 = RectangleShape(height=50, width=20)
```

Let's have another simple example where we have the following function signature:

Figure 4.275

```
def do_something(namespaced_name: str, ...):
    # ...
```

The above function signature allows function callers to supply a non-namespaced name accidentally. By using a custom type for the namespaced name, we can formulate the above function signature to the following:

Figure 4.276

```
from typing import Final


class NamespacedName:
    def __init__(self, namespace: str, name: str):
        self.__namespaced_name: Final = (
            (namespace + '.' + name) if namespace else name
        )

    def get(self) -> str:
        return self.__namespaced_name


def do_something(namespaced_name: NamespacedName, ...):
    # ...
```

Let's have a more comprehensive example with an `HttpUrl` class. The class constructor has several parameters that should be validated upon creating an HTTP URL:

Figure 4.277

```
from typing import Final


class HttpUrl:
    def __init__(
        self,
        scheme: str,
        host: str,
        port: int,
        path: str,
        query: str
    ):
        self.__url_string: Final = (
              scheme
              + "://"
              + host
              + ":"
              + str(port)
              + path
              + "?"
              + query
        )
```

Let’s introduce an abstract class for validated values:

Figure 4.278

```
from abc import abstractmethod
from typing import Final, Generic, TypeVar

from Optional import Optional

T = TypeVar('T')


class AbstractValidatedValue(Generic[T]):
    def __init__(self, value: T):
        self._value: Final = value

    def get(self) -> Optional[T]:
        return (
            Optional.of(self._value)
            if self._is_valid()
            else Optional.empty()
        )

    class GetError(Exception):
        pass

    def try_get(self) -> T:
        if self._is_valid():
            return self._value
        else:
            raise self.GetError('Invalid ' +
self.__class__.__name__)

    @abstractmethod
        def _is_valid(self) -> bool:
            pass
```

Let's create a class for validated HTTP scheme objects:

Figure 4.279

```
from functools import cache

from AbstractValidatedValue import AbstractValidatedValue


class HttpScheme(AbstractValidatedValue[str]):
    # Because instances are immutable, we can cache the
validation result
    # This would be especially beneficial in cases where
    # the validation logic is complex and this method is
    # called many times
    @cache
    def _is_valid(self) -> bool:
        lowercase_value = self._value.lower()
        return lowercase_value == 'https' or lowercase_value ==
'http'
```

Let's create a `Port` class (and similar classes for the host, path, and query should be created):

Figure 4.280

```
from functools import cache

from AbstractValidatedValue import AbstractValidatedValue


class Port(AbstractValidatedValue[int]):
    # Because instances are immutable, we can cache the
validation result
    @cache
    def _is_valid(self) -> bool:
        return 1 <= self._value <= 65535


# Implement Host class ...
# Implement Path class ...
# Implement Query class ...
```

Let's create a utility class, `OptionalUtils`, with a method for mapping a result for five optional values:

Figure 4.281

```
from collections.abc import Callable
from typing import TypeVar, final

from Optional import Optional

T = TypeVar('T')
U = TypeVar('U')
V = TypeVar('V')
X = TypeVar('X')
Y = TypeVar('Y')
R = TypeVar('R')
```

```
@final
class OptionalUtils:
    @staticmethod
    def map_all(
      opt1: Optional[T],
      opt2: Optional[U],
      opt3: Optional[V],
      opt4: Optional[X],
      opt5: Optional[Y],
      mapper: Callable[[T, U, V, X, Y], R]
  ) -> Optional[R]:
        if (
            opt1.is_present()
            and opt2.is_present()
            and opt3.is_present()
            and opt4.is_present()
            and opt5.is_present()
         ):
            return Optional.of(
                map(
                    opt1.try_get(),
                    opt2.try_get(),
                    opt3.try_get(),
                    opt4.try_get(),
                    opt5.try_get(),
                )
            )
        else:
            return Optional.empty()
```

Next, we can reimplement the `HttpUrl` class to contain two alternative factory methods for creating an HTTP URL:

Figure 4.282

```
# Imports ...

T = TypeVar("T")

class PrivateConstructor(type):
    def __call__(
        cls: type[T], *args: tuple[Any, ...], **kwargs:
dict[str, Any]
    ):
        raise TypeError('Constructor is private')

    def _create(
        cls: type[T], *args: tuple[Any, ...], **kwargs:
dict[str, Any]
    ) -> T:
        return super().__call__(*args, **kwargs)


class HttpUrl(metaclass=PrivateConstructor):
    def __init__(self, url_string: str):
        self.__url_string: Final = url_string

    @property
```

```
    def url_string(self) -> str:
        return self.__url_string

    # Factory method that returns an optional HttpUrl
    @classmethod
    def create(
        cls,
        scheme: HttpScheme,
        host: Host,
        port: Port,
        path: Path,
        query: Query
    ) -> Optional['HttpUrl']:
        return OptionalUtils.map_all(
            scheme.get(),
            host.get(),
            port.get(),
            path.get(),
            query.get(),
            lambda scheme_val, host_val, port_val, path_val,
query_val: cls._create(
                scheme_val
                + '://'
                + host_val
                + ':'
                + port_val
                + path_val
                + '?'
                + query_val
            )
        )

    class CreateError(Exception):
        pass

    # Factory method that returns a valid HttpUrl or
    # raises an error
    @classmethod
    def try_create(
        cls,
        scheme: HttpScheme,
        host: Host,
        port: Port,
        path: Path,
        query: Query
    ) -> 'HttpUrl':
        try:
            return cls._create(
                scheme.try_get()
                + '://'
                + host.try_get()
                + ':'
                + str(port.try_get())
                + path.try_get()
                + '?'
                + query.try_get()
            )
        except AbstractValidatedValue.GetError as error:
            raise cls.CreateError(error)


maybe_http_url = HttpUrl.create(
```

```
    HttpScheme('https'),
    Host('www.google.com'),
    Port(443),
    Path('/query'),
    Query('search=jee')
)

# Prints https://www.google.com:443/query?search=jee
print(maybe_http_url.try_get().url_string)

# Raises an error: Invalid Port
http_url2 = HttpUrl.try_create(
    HttpScheme('https'),
    Host('www.google.com'),
    Port(443222),
    Path('/query'),
    Query('search=jee')
)
```

Notice how we did not hardcode the URL validation inside the `HttpUrl` class, but we created small validated value classes: `HttpScheme`, `Host`, `Port`, `Path`, and `Query`. These classes can be further utilized in other parts of the codebase if needed and can even be put into a shared validation library for broader usage.

An application typically receives unvalidated input data from the following external sources:

- Reading command line arguments
- Reading environment variables
- Reading standard input
- Reading files from the file system
- Reading data from a socket (network input)
- Receiving input from a user interface (UI)

Make sure that you validate any data received from the sources mentioned above. Use a ready-made validation library or create your own validation logic if needed. Validate the input immediately after receiving it from an untrustworthy source and only pass valid values to other functions in the codebase. In this way, other functions in the codebase can trust the input they receive, and they don't have to validate it again. If you pass unvalidated data freely around in your application,

you may need to implement validation logic in every function, which is unreasonable.

4.17: Dependency Injection (DI) Principle

> ***Dependency injection (DI) allows changing the behavior of an application based on static or dynamic configuration.***

When using dependency injection, the dependencies are injected only after the application startup. The application can first read its configuration and then decide what objects are created for the application. In many languages, dependency injection is crucial for unit tests, too. When executing a unit test using DI, you can inject mock dependencies into the tested code instead of using the standard dependencies of the application.

Below is an example of using the *singleton pattern* without dependency injection:

Figure 4.283

```
from enum import Enum

from Logger import Logger


class LogLevel(Enum):
    ERROR = 1
    WARN = 2
    INFO = 3
    DEBUG = 4
    TRACE = 5


class StdOutLogger(Logger):
    @staticmethod
    def log(log_level: LogLevel, message: str) -> None:
        # Log to standard output


class Application:
    def run(self):
        StdOutLogger.log(LogLevel.INFO, 'Starting application')
        # ...
```

In the above example, we use a static method of the hard-coded `StdOutLogger` class. It is difficult to change the logger later (assuming the static log method is used in many places in the codebase).

We should refactor the above code not to use a static method but use dependency injection instead:

Figure 4.284. Logger.py

```
from enum import Enum
from typing import Protocol


class LogLevel(Enum):
    ERROR = 1
    WARN = 2
    INFO = 3
    DEBUG = 4
    TRACE = 5


class Logger(Protocol):
    def log(self, log_level: LogLevel, message: str) -> None:
        pass
```

Figure 4.285. StdOutLogger.py

```
from Logger import Logger, LogLevel


class StdOutLogger(Logger):
    def log(self, log_level: LogLevel, message: str) -> None:
        # Log to standard output
```

Figure 4.286. Application.py

```
from dependency_injector.wiring import Provide, inject
from Logger import Logger
from LogLevel import LogLevel


class Application:
    @inject
    def __init__(self, logger: Logger = Provide['logger']):
        self.__logger = logger

    def run(self):
        self.__logger.log(LogLevel.INFO, 'Starting application')
        # ...
```

Then we need to define the DI container:

Figure 4.287. DiContainer.py

```
from dependency_injector import containers, providers
from StdOutLogger import StdOutLogger
```

```
class DiContainer(containers.DeclarativeContainer):
    wiring_config = containers.WiringConfiguration(
        modules=['Application']
    )

    logger = providers.Singleton(StdOutLogger)
```

Now, changing the code to use a different logger is easy. Let's say we want to log into a file instead of the standard output. We can introduce a new class for the file-based logger (following the *open-closed principle*)

Figure 4.288. FileLogger.py

```
from typing import Final

from Logger import Logger, LogLevel


class FileLogger(Logger):
    def __init__(self, log_file_directory: str):
        self.__log_file_directory: Final = log_file_directory

    def log(self, log_level: LogLevel, message: str):
        # Log to a file in self.__log_file_directory
```

Then, we can change the DI container to use the file-based logger instead of the `StdOutLogger`:

Figure 4.289. DiContainer.py

```
import os

from dependency_injector import containers, providers
from FileLogger import FileLogger


class DiContainer(containers.DeclarativeContainer):
    wiring_config = containers.WiringConfiguration(modules=
['Application'])

    logger = providers.Singleton(
        FileLogger, os.environ.get('LOG_FILE_DIRECTORY',
'/var/log')
    )
```

We can also change the logging behavior dynamically based on the environment where the application is running:

Figure 4.290. DiContainer.py

```
import os

from dependency_injector import containers, providers
from FileLogger import FileLogger
from StdOutLogger import StdOutLogger
```

```
class DiContainer(containers.DeclarativeContainer):
    wiring_config = containers.WiringConfiguration(modules=
['Application'])

    if os.environ.get('LOG_DESTINATION') == 'file':
        logger = providers.Singleton(
            FileLogger, os.environ.get('LOG_FILE_DIRECTORY',
'/var/log')
        )
    else:
        logger = providers.Singleton(StdOutLogger)
```

If you have a very simple microservice with few dependencies, you might think dependency injection is an overkill. One thing that is sure in software development is change. Change is inevitable, and you cannot predict the future. For example, your microservice may start growing larger. Introducing DI in a late phase of a project might require substantial refactoring. Therefore, consider using DI in all non-trivial applications from the beginning.

For all full-stack Python developers, below is a TypeScript example of a *data-visualization-web-client* where the noicejs NPM library is used for dependency injection. This library resembles the famous Google Guice library. Below is a `FakeServicesModule` class that configures dependencies for different backend services that the web client uses. As you can notice, all the services are configured to use fake implementations because this DI module is used when the backend services are not yet available. A `RealServicesModule` class can be implemented and used when the backend services become available. In the `RealServicesModule` class, the services are bound to their actual implementation classes instead of fake implementations.

Figure 4.291

```
import { Module } from 'noicejs';
import FakeDataSourceService from ...;
import FakeMeasureService from ...;
import FakeDimensionService from ...;
import FakeChartDataService from ...;

export default class FakeServicesModule extends Module {
  override async configure(): Promise<void> {
    this.bind('dataSourceService')
      .toInstance(new FakeDataSourceService());

    this.bind('measureService')
      .toInstance(new FakeMeasureService());
```

```
    this.bind('dimensionService')
      .toInstance(new FakeDimensionService());

    this.bind('chartDataService')
      .toInstance(new FakeChartDataService());
    );
  }
}
```

With the *noicejs* library, you can configure several DI modules and create a DI container from the wanted modules. The module approach lets you divide dependencies into multiple modules, so you don't have a single big module and lets you instantiate a different module or modules based on the application configuration.

In the below example, the DI container is created from a single module, an instance of the `FakeServicesModule` class:

Figure 4.292. diContainer.ts

```
import { Container } from 'noicejs';
import FakeServicesModule from './FakeServicesModule';

const diContainer = Container.from(new FakeServicesModule());

export default diContainer;
```

In the development phase, we could create two separate modules, one for fake services and another one for real services, and control the application behavior based on the web page's URL query parameter:

Figure 4.293. diContainer.ts

```
import { Container } from 'noicejs';
import FakeServicesModule from './FakeServicesModule';
import RealServicesModule from './RealServicesModule';

const diContainer = (() => {
  if (location.href.includes('useFakeServices=true')) {
    // Use fake services if web page URL
    // contains 'useFakeServices=true'
    return Container.from(new FakeServiceModule());
  } else {
    // Otherwise use real services
    return Container.from(new RealServicesModule());
  }
})();

export default diContainer;
```

Then, you must configure the `diContainer` before the dependency injection can be used. In the below example, the `diContainer` is configured before a React application is rendered:

Figure 4.294. app.ts

```
import React from 'react';
import ReactDOM from 'react-dom';
import diContainer from './diContainer';
import AppView from './app/view/AppView';

diContainer.configure().then(() => {
  ReactDOM.render(<AppView />, document.getElementById('root'));
});
```

Then, in Redux actions, where you need a service, you can inject the required service with the `@Inject` decorator. You specify the name of the service you want to inject. The service will be injected as the class constructor argument's property (with the same name).

Figure 4.295. StartFetchChartDataAction.ts

```
// Imports ...

type ConstructorArgs = {
  chartDataService: ChartDataService,
  chart: Chart,
  dispatch: Dispatch;
};

export default
@Inject('chartDataService')
class StartFetchChartDataAction extends AbstractChartAreaAction
{
  private readonly chartDataService: ChartDataService;
  private readonly chart: Chart;

  constructor({ chart,
                chartDataService,
                dispatch }: ConstructorArgs) {
    super(dispatch);
    this.chartDataService = chartDataService;
    this.chart = chart;
  }

  perform(currentState: ChartAreaState): ChartAreaState {
    this.chartDataService
      .fetchChartData(
        this.chart.dataSource,
        this.chart.getColumns(),
        this.chart.getFilters(),
        this.chart.getSortBys()
      )
      .then((columnNameToValuesMap: ColumnNameToValuesMap) => {
        this.dispatch(
```

```
          new FinishFetchChartDataAction(columnNameToValuesMap,
                                         this.chart.id)
        );
      })
      .catch((error) => {
        // Handle error
      });

    this.chart.isFetchingChartData = true;
    return ChartAreaStateUpdater
            .getNewStateForChangedChart(currentState,
this.chart);
  }
}
```

To be able to dispatch the above action, a controller should be implemented:

Figure 4.296

```
import diContainer from './diContainer';
import StartFetchChartDataAction from
'./StartFetchChartDataAction';
import Controller from './Controller';
import store from './store';

class ChartAreaController extends Controller {
  readonly actionDispatchers = {
    startFetchChartData: (chart: Chart) =>
      // the 'chart' is given as a property to
      // StartFetchChartDataAction class constructor
      this.dispatchWithDi(diContainer,
                          StartFetchChartDataAction,
                          { chart });
  }
}

export const controller = new
ChartAreaController(store.dispatch);
export type ActionDispatchers = typeof
controller.actionDispatchers;
```

The following base classes are also defined:

Figure 4.297. AbstractAction.ts

```
export default abstract class AbstractAction<S> {
  abstract perform(state: S): S;
}
```

Figure 4.298. AbstractDispatchingAction.ts

```
// Imports...

export default abstract class AbstractDispatchingAction<S>
    extends AbstractAction<S> {
  constructor(protected readonly dispatch: Dispatch) {}
}
```

Figure 4.299. AbstractChartAreaAction.ts

```
// Imports...

export default abstract class AbstractChartAreaAction
         extends AbstractDispatchingAction<ChartAreaState> {
}
```

Figure 4.300. Controller.ts

```
export type ReduxDispatch =
  (reduxActionObject: { type: AbstractAction<any> }) => void;

export default class Controller {
  protected readonly dispatch:
    (action: AbstractAction<any>) => void;

  constructor(reduxDispatch: ReduxDispatch) {
    this.dispatch = (action: AbstractAction<any>) =>
      reduxDispatch({ type: action });
  }

  dispatchWithDi(
    diContainer: { create: (...args: any[]) => Promise<any> },
    ActionClass:
      abstract new (...args: any[]) => AbstractAction<any>,
    otherArgs: {}
  ) {
    // diContainer.create will create a new object of
    // class ActionClass.
    // The second parameter of the create function defines
    // additional properties supplied to ActionClass
constructor.
    // The create method is asynchronous. When it succeeds,
    // the created action object is available in the 'then'
    // function and it can be now dispatched

    diContainer
      .create(ActionClass, {
        dispatch: this.dispatch,
        ...otherArgs
      })
      .then((action: any) => this.dispatch(action));
  }
}
```

4.18: Avoid Code Duplication Principle

> ***At the class level, when you spot duplicated code in two different classes implementing the same interface, you should create a new base class to accommodate the common functionality and let the classes extend the new base class.***

Below is an `AvroBinaryKafkaInputMessage` class that implements the `InputMessage` protocol:

Figure 4.301. InputMessage.py

```
from typing import Protocol

from DecodedMessage import DecodedMessage
from Schema import Schema


class InputMessage(Protocol):
    def try_decode_schema_id(self) -> int:
        pass

    def try_decode(self, schema: Schema) -> DecodedMessage:
        pass
```

Figure 4.302. AvroBinaryKafkaInputMessage.py

```
from typing import Final

from DecodedMessage import DecodedMessage
from InputMessage import InputMessage
from KafkaMessage import KafkaMessage
from Schema import Schema


class AvroBinaryKafkaInputMessage(InputMessage):
    def __init__(self, kafka_message: KafkaMessage):
        self.__kafka_message: Final = kafka_message

    def try_decode_schema_id(self) -> int:
        # Try decode schema id from the beginning of
        # the Avro binary Kafka message

    def try_decode(self, schema: Schema) -> DecodedMessage:
        return schema.try_decode_message(
            self.__kafka_message.payload,
            self.__kafka_message.length
        )
```

If we wanted to introduce a new Kafka input message class for JSON, CSV, or XML format, we could create a class like the `AvroBinaryKafkaInputMessage` class. But then we could notice the code duplication in the `try_decode` method. We can notice that the `try_decode` method is the same regardless of the input message source and format. According to this principle, we should move the duplicate code to a common base class, `AbstractInputMessage`. We could make the `try_decode` method a template method according to the *template method pattern* and create abstract methods for getting the message data and its length:

Figure 4.303. AbstractInputMessage.py

```
from abc import abstractmethod
from typing import final

from DecodedMessage import DecodedMessage
from InputMessage import InputMessage
from Schema import Schema


class AbstractInputMessage(InputMessage):
    @abstractmethod
    def try_decode_schema_id(self) -> int:
        pass

    # Template method
    @final
    def try_decode(self, schema: Schema) -> DecodedMessage:
        return schema.try_decode_message(
            self._get_data(),
            self._get_length()
        )

    @abstractmethod
        def _get_data(self) -> bytearray:
            pass

    @abstractmethod
        def _get_length(self) -> int:
            pass
```

Next, we should refactor the `AvroBinaryKafkaInputMessage` class to extend the new `AbstractInputMessage` class and implement the `_get_data` and `_get_length` methods. But we can realize these two methods are the same for all Kafka input message data formats. We should not implement those two methods in the `AvroBinaryKafkaInputMessage` class because we would need to implement them as duplicates if we needed to add a Kafka input message class for another data format. Once again, we can utilize this principle and create a new base class for Kafka input messages:

Figure 4.304. AbstractKafkaInputMessage.py

```
from abc import abstractmethod
from typing import Final

from AbstractInputMessage import AbstractInputMessage
from KafkaMessage import KafkaMessage


class AbstractKafkaInputMessage(AbstractInputMessage):
    def __init__(self, kafka_message: KafkaMessage):
        self.__kafka_message: Final = kafka_message

    @abstractmethod
    def try_decode_schema_id(self) -> int:
        pass
```

```
    def _get_data(self) -> bytearray:
        return self.__kafka_message.payload

    def _get_length(self) -> int:
        return self.__kafka_message.length
```

Finally, we can refactor the `AvroBinaryKafkaInputMessage` class to contain no duplicated code:

Figure 4.305. AvroBinaryKafkaInputMessage.py

```
from AbstractKafkaInputMessage import AbstractKafkaInputMessage


class AvroBinaryKafkaInputMessage(AbstractKafkaInputMessage):
    def try_decode_schema_id(self) -> int:
        # Try decode the schema id from the beginning of
        # the Avro binary Kafka message
        # Use base class _get_data() and _get_length()
        # methods to achieve that
```

4.19: Inheritance in Cascading Style Sheets (CSS)

This last section of this chapter is for full-stack Python developers interested in how inheritance works in CSS. In HTML, you can define classes (class names) for HTML elements in the following way:

Figure 4.306

```
<span class="icon pie-chart-icon">...</span>
```

In a CSS file, you define CSS properties for CSS classes, for example:

Figure 4.307

```
.icon {
  background-repeat: no-repeat;
  background-size: 1.9rem 1.9rem;
  display: inline-block;
  height: 2rem;
  margin-bottom: 0.2rem;
  margin-right: 0.2rem;
  width: 2rem;
}

.pie-chart-icon {
  background-image: url('pie_chart_icon.svg');
}
```

The problem with the above approach is that it is not correctly object-oriented. In the HTML code, you must list all the class names to combine all the needed CSS properties. It is easy to forget to add a class name. For example, you could specify `pie-chart-icon` only and forget to specify the `icon`.

It is also difficult to change the inheritance hierarchy afterward. Suppose you wanted to add a new class `chart-icon` for all the chart icons:

Figure 4.308

```
.chart-icon {
  // Define properties here...
}
```

You would have to remember to add the `chart-icon` class name to all places in the HTML code where you are rendering chart icons:

Figure 4.309

```
<span class="icon chart-icon pie-chart-icon">...</span>
```

The above-described approach is very error-prone. What you should do is introduce proper object-oriented design. You need a CSS preprocessor that makes extending CSS classes possible. In the below example, I am using SCSS:

Figure 4.310

```
<span class="pieChartIcon">...</span>
```

Figure 4.311

```
.icon {
  background-repeat: no-repeat;
  background-size: 1.9rem 1.9rem;
  display: inline-block;
  height: 2rem;
  margin-bottom: 0.2rem;
  margin-right: 0.2rem;
  width: 2rem;
}

.chartIcon {
  @extend .icon;

  // Other chart icon related properties...
}

.pieChartIcon {
  @extend .chartIcon;
```

```
  background-image:
url('../../../../../assets/images/icons/chart/pie_chart_icon.svg
');
}
```

In the above example, we define only one class for the HTML element. The inheritance hierarchy is defined in the SCSS file using the `@extend` directive. We are now free to change the inheritance hierarchy in the future without any modification needed in the HTML code.

5: Coding Principles

This chapter presents principles for coding. The following principles are presented:

- Uniform variable naming principle
- Uniform source code repository structure principle
- Domain-based source code structure principle
- Avoid comments principle
- Function single return statement principle
- Prefer statically typed language principle
- Refactoring principle
- Static code analysis principle
- Error/Exception handling principle
- Avoid off-by-one errors principle
- Be critical when googling principle
- Make one change at a time principle
- Use appropriate data structure principle
- Optimization principle

5.1: Uniform Variable Naming Principle

> ***A good variable name should describe the variable's purpose and its type.***

Writing your code with great names, at best, makes it read like prose. And remember that code is more often read than written, so code must be easy to read and understand.

Naming variables with names that convey information about the variable's type is crucial in untyped languages and beneficial in typed languages because modern typed languages use automatic type deduction, and you won't always see the actual variable type. But when the variable's name tells its type, it does not matter if the type name is not visible.

If a variable name is 20 or more characters long, consider making it shorter. Try to abbreviate one or more words in the variable name, but only use meaningful and well-known abbreviations. If such abbreviations don't exist, then don't abbreviate at all. For example, if you have a variable named `environment_variable_name`, you should try to shorten it because it is over 20 characters long. You can abbreviate *environment* to *environ* and *variable* to *var*, resulting in a variable name `environ_var_name` that is short enough. Both abbreviations `environ` and `var` are commonly used and well understood. Let's have another example with a variable named `loyalty_bonus_percentage`. You cannot abbreviate *loyalty*. You cannot abbreviate *bonus*. But you can abbreviate *percentage* to *percent* or even *pct*. I would rather use *percent* instead of *pct*. Using *percent* makes the variable name shorter than 20 characters (underscores are not counted as variable name characters).

If a variable name is less than 20 characters long, you don't need to shorten it. If you have several variables named in a similar wording and one or more of them needs abbreviating, you can use that abbreviation in all the variable names for consistency. For example, if you have variables `configuration_file` and `configuration_file_parser`, you can abbreviate the both to `config_file` and `config_file_parser`.

In the following sections, naming conventions for different types of variables are proposed.

5.1.1: Naming Integer Variables

Some variables are intrinsically integers, like *age* or *year*. Everybody immediately understands that the type of an *age* or *year* variable is a number and, more specifically, an integer.

So, you don't have to add anything to the variable's name to indicate its type. It already tells you its type.

One of the most used categories of integer variables is a count or number of something. You see those kinds of variables in every piece of code. I recommend using the following convention for naming those variables: *number_of_<something>* or alternatively *<something>_count*. For example, *number_of_failures* or *failure_count*. You should not use a variable named *failures* to designate a failure count. The problem with that variable name is it does not clearly specify the variable type and thus can cause some confusion. This is because a variable named *failures* can be misunderstood as a collection variable (e.g., a list of failures).

If the unit of a variable is not self-evident, always add information about the unit to the end of the variable name. For example, instead of naming a variable *tooltip_show_delay*, you should name it *tooltip_show_delay_in_millis* or *tooltip_show_delay_in_ms*. If you have a variable whose unit is self-evident, unit information is not needed. So, there is no need to name an *age* variable as *age_in_years*. But if you are measuring age in months, you must name the respective variable as *age_in_months* so that people don't assume that age is measured in years.

5.1.2: Naming Floating-Point Number Variables

Floating-point numbers are not as common as integers, but sometimes you need them too. Some values are intrinsically floating-point numbers, like most un-rounded measures (e.g., price, height, width, or weight). A floating-point variable would be a safe bet if you need to store a measured value.

If you need to store an amount of something that is not an integer, use a variable named *<something>_amount*, like *rainfall_amount*. When you see the "amount of something" in code, you can automatically think it is a floating-point number. If you need to use a number in arithmetic, depending on the application, you might want to use either floating-point or integer arithmetic. In the case of money, you should use

integer arithmetic to avoid rounding errors. Instead of a floating-point *money_amount* variable, you should have an integer variable, like *money_in_cents*.

If the unit of a variable is not self-evident, add information about the unit to the end of the variable name, like *rainfall_amount_in_mm*, *width_in_inches*, *angle_in_degrees* (values 0-360), *failure_percent* (values 0-100), or *failure_ratio* (values 0-1).

5.1.3: Naming Boolean Variables

Boolean variables can have only one of two values: true or false. The name of a boolean variable should form a statement where the answer is true or false, or yes or no. Typical boolean variable naming patterns are: *is_<something>*, *has_<something>*, *did_<something>*, *should_<something>*, *can_<something>, or *will_<something>*. Some examples of variable names following the above patterns are *is_disabled*, *has_errors*, *did_update*, *should_update*, and *will_update*.

The verb in the boolean variable name does not have to be at the beginning. It can and should be in the middle if it makes the code read better. Boolean variables are often used in if-statements where changing the word order in the variable name can make the code read better. Remember that, at best, code reads like beautiful prose and is read more often than written.

Below is a code snippet where we have a boolean variable named `is_pool_full`:

Figure 5.1

```
# ...

is_pool_full = len(self.__pooled_messages) >= 200
if is_pool_full:
    # ...
else:
    # ...
```

We can change the variable name to `pool_is_full` to make the if-statement read more fluently. In the below example, the if-statement reads “if pool_is_full” instead of “if is_pool_full”:

Figure 5.2

```
# ...

pool_is_full = len(self.__pooled_messages) >= 200
if pool_is_full:
    # ...
else:
    # ...
```

Don't use boolean variable names in the form of *<passive-verb>_something*, like *inserted_field*, because this can confuse the reader. It is unclear if the variable name is a noun (a field that was inserted) that names an object or a boolean statement. Instead, use either *did_insert_field* or *field_was_inserted*.

Below is an example of the incorrect naming of a variable used to store a function return value. The `drop_redundant_tables` function returns a boolean. Someone might think the `tables_dropped` variable means a list of dropped table names. So, the name of the variable is obscure and should be changed.

Figure 5.3

```
tables_dropped = drop_redundant_tables(
    prefix,
    vms_data,
    config.database,
    hive_client,
    logger
)

if tables_dropped:
    # ...
```

Below is the above example modified so that the variable name is changed to indicate a boolean statement:

Figure 5.4

```
tables_were_dropped = drop_redundant_tables(
    prefix,
    vms_data,
    config.database,
    hive_client,
    logger
)

if tables_were_dropped:
    # ...
```

You could have used a variable named `did_drop_tables`, but the `tables_were_dropped` makes the if-statement more readable. If the return value of the `drop_redundant_tables` function were a list of dropped table names, I would name the return value receiving variable as `dropped_table_names`.

When you read code containing a negated boolean variable, it usually reads terrible, for example:

Figure 5.5

```
app_was_started = app.start()

if not app_was_started:
    # ...
```

To improve the readability, you can mentally move the `not` word to the correct place to make the sentence read like proper English. For example: *if app was not started*

The other option is to negate the variable. That is done by negating both sides of the assignment by adding `not` on both sides of the assignment operator. Here is an example:

Figure 5.6

```
app_was_not_started = not app.start()

if app_was_not_started:
    # ...
```

5.1.4: Naming String Variables

String variables are prevalent, and many things are intrinsically strings, like *name*, *title*, *city*, *country*, or *topic*. When you need to store numerical data in a string variable, tell the code reader clearly that it is a question about a number in string format, and use a variable name in the following format: *<someValue>_string* or *<someValue>_as_string*. It makes the code more prominent and easier to understand. For example:

Figure 5.7

```
try:
    year = int(year_as_string)
except ValueError:
    # ...
```

If you have a variable that could be confused with an object variable, like `schema` (could be confused with an instance of `Schema` class), but it is a string, add *string* to the end of the variable name, e.g., `schema_string`. Here is an example:

Figure 5.8

```
schema = schema_parser.parse(schema_string)
```

5.1.5: Naming Enum Variables

Name enum variables with the same name as the enum type. For example, a `CarType` enum variable should be named `car_type`. If the name of an enum type is very generic, like `Result`, you might benefit from declaring an enum variable with some detail added to the variable name. Below is an example of a very generic enum type name:

Figure 5.9

```
# Returns enum type 'Result'
result = pulsar_client.create_producer(...)

if result == Result.Ok:
    # ...
```

Let's add some detail and context to the `result` variable name:

Figure 5.10

```
producer_create_result = pulsar_client.create_producer(...)

if producer_create_result == Result.Ok:
    # ...
```

5.1.6: Naming Collection (List and Set) Variables

When naming lists and sets, you should use the plural form of a noun, like *customers*, *errors*, or *tasks*. These kinds of names work well in the code, for example:

Figure 5.11

```
def process(customers: list[Customer]) -> list[Customer]:
    # ...

customers = [...]

for customer in customers:
    # ...
```

```
processed_customers = process(customers)

def is_even(integer: int) -> bool:
    return integer % 2 == 0

integers = [1, 2, 3, 4, 5]
even_integers = filter(is_even, integers)
```

This plural noun naming convention is usually enough because you don't necessarily need to know the underlying collection implementation. Using this naming convention allows you to change the type of a collection variable without changing the variable name. If you are iterating over a collection, it does not matter if it is a list or set. Thus, it does not bring any benefit if you add the collection type name to the variable name, for example, *customer_list* or *task_set*. Those names are just longer. You might want to specify the collection type in some special cases. Then, you can use the following kinds of variable names: *queue_of_tasks*, *stack_of_cards*, or *set_of_timestamps*.

Below is an example where the function is named correctly to return a collection (of categories), but the variable receiving the return value is not named according to the collection variable naming convention:

Figure 5.12

```
vms_data = vms_client.get_categories(vms_url, logger)
```

Correct naming would be:

Figure 5.13

```
vms_categories = vms_client.get_categories(vms_url, logger)
```

5.1.7: Naming Dictionary Variables

Dictionaries are accessed by requesting a *value* for a certain *key*. This is why I recommend naming dictionaries using the pattern *key_to_value*, *key_to_value_map* or *key_to_value_dict*. Let's say we have a dictionary containing order counts for customer ids. This dictionary should be named `customer_id_to_order_count`, `customer_id_to_order_count_map`, or

`customer_id_to_order_count_dict`. If we have a list of suppliers for product names, the mapping variable should be named `product_name_to_suppliers`, `product_name_to_suppliers_map`, or `product_name_to_suppliers_dict`. Below is an example of accessing dicts:

Figure 5.14

```
order_count = customer_name_to_order_count.get(customer_name)
suppliers = product_name_to_suppliers.get(product_name)
```

Below are examples of iterating over a dict:

Figure 5.15

```
customer_name_to_order_count = {
    'John': 10,
    'Peter': 5
}

for customer_name in customer_name_to_order_count:
    print(customer_name)

for customer_name in customer_name_to_order_count.keys():
    print(customer_name)

for order_count in customer_name_to_order_count.values():
    print(order_count)

for (
    customer_name,
    order_count,
) in customer_name_to_order_count.items():
    print(customer_name, order_count)
```

5.1.8: Naming Pair and Tuple Variables

A variable containing a pair should be named using the pattern *variable1_and_variable2*. For example: `height_and_width`. And for tuples, the recommended naming pattern is *variable1_variable2_...and_variableN*. For instance: `height_width_and_depth`. If the tuple does not have values that should be individually named, you can treat it as an immutable list and name it according to the collection variable naming convention.

Below is an example of using pairs and tuples:

Figure 5.16

```
height_and_width = (100, 200)
height, width = height_and_width
```

```
height_width_and_depth = (100, 200, 40)
height, *, depth = height_width_and_depth
numbers = (1, 2, 3, 4, 5)
print(numbers[-1]) # Prints 5
print(numbers[0:2]) # Prints (1,2)
print(numbers[:2]) # Prints (1,2)
print(numbers[1:]) # Prints (2, 3, 4, 5)
```

5.1.9: Naming Object Variables

Object variables refer to an instance of a class. Class names are nouns written in CapWords, like *Person*, *CheckingAccount*, or *Task*. Object variable names should contain the related class name: a *person* object of the *Person* class, an *account* object of the *Account* class, etc. You can freely decorate the object's name, for example, with an adjective: *completed_task*. Including the class name or at least a significant part of it at the end of the variable name is useful because looking at the end of the variable name tells what kind of object is in question.

Sometimes, you might want to name an object variable so that the name of its class is implicit, for example:

Figure 5.17

```
# The class of the function parameters, 'Location', is implicit
drive(from=home, to=destination)
```

In the above example, the classes of `home` and `destination` objects are not explicit. In most cases, it is preferable to make the class name explicit in the variable name when it does not make the variable name too long. This is because of the variable type deduction. The types of variables are not necessarily visible in the code, so the variable name should communicate the type of a variable. Below is an example where the types of function parameters are explicit.

Figure 5.18

```
# The class of the function parameters, 'Location', is now
explicit
drive(from=home_location, to=dest_location)
```

5.1.10: Naming Optional Variables

If you are using the `Optional[T]` class (presented earlier in this book), name variables of this type using the following pattern: *maybe_<something>*:

Figure 5.19

```
maybe_logged_in_user.if_present(lambda logged_in_user:
logged_in_user.logout())
current_user = maybe_logged_in_user.or_else(guest_user)
```

When you create optional types using a type union, you don't need any prefixes in optional variable names. In the below example, the `discount` parameter is optional:

Figure 5.20

```
def add_tax(price: float, discount: int | None = None) -> float:
    return 1.2 * (price - (0 if discount is None else discount))

price_with_tax = add_tax(price_without_tax)
```

5.1.11: Naming Function Variables (Callbacks)

Callback functions are supplied to other functions to be called at some point. If a callback function returns a value, it can be named according to the returned value, but it should still contain a verb. If the callback function does not return a value, you should name the callback function like any other function: Indicating what the function does.

Figure 5.21

```
from functools import reduce

def doubled(number: int | float):
    return 2 * number


def squared(number: int | float):
    return number**2


def is_even(number: int | float):
    return number % 2 == 0


values = [1, 2, 3, 4, 5]
doubled_values = [doubled(value) for value in values]
doubled_values2 = list(map(doubled, values))
squared_values = [squared(value) for value in values]
squared_values2 = list(map(squared, values))
even_values = [value for value in values if is_even(value)]
even_values2 = list(filter(is_even, values))
```

```
def trimmed(string: str):
    return string.strip()


strings = [" string1", "string2 "]
trimmed_strings = [trimmed(string) for string in strings]
trimmed_strings2 = list(map(trimmed, strings))

def sum(accum_sum: int | float, number: int | float) -> int |
float:
    return accum_sum + number

sum_of_values = reduce(sum, values, 0)
```

If the callback function is very simple and short, like the *doubled* and *squared* functions are, we can inline them in Python list/set/dict comprehensions and generator expressions, making them a bit shorter:

Figure 5.22

```
doubled_values = [2 * value for value in values]
squared_values = [value**2 for value in values]
distinct_values = {value for value in values}
value_to_squared_value = {value: value**2 for value in values}
generator_of_doubled_values = (2 * value for value in values)
first_doubled_value = generator_of_doubled_values.__next__()
tuple_of_doubled_values = tuple(generator_of_doubled_values)
```

Let's have a quick detour with an example written in Clojure:

Figure 5.23

```
(defn print-first-n-doubled-integers [n]
  (println (take n (map (fn [x] (* 2 x)) (range)))))
```

To understand what happens in the above code, you should start reading from the innermost function call and proceed toward the outermost function call. A function call is inside parenthesis. When traversing the function call hierarchy, the difficulty lies in storing and retaining information about all the nested function calls in short-term memory.

We could simplify reading the above example by naming the anonymous function and introducing variables (constants) for intermediate function call results. Of course, our code becomes more prolonged, but coding is not a competition to write the shortest possible code but to write the shortest, most readable, and understandable code for other people and your

future self. It is a compiler's job to compile the longer code below into code that is as efficient as the shorter code above.

Below is the above code refactored:

Figure 5.24

```
(defn print-first-n-doubled-integers [n]
  (let [doubled (fn [x] (* 2 x))
        doubled-integers (map doubled (range))
        first-n-doubled-integers (take n doubled-integers)]
          (println first-n-doubled-integers)))
```

Let's think hypothetically: if Clojure's `map` function took parameters in a different order and the `range` function was named `integers` and the `take` function was named `take-first` (like there is the `take-last` function), we would have an even more explicit version of the original code:

Figure 5.25

```
(defn print-first-n-doubled-integers [n]
  (let [doubled (fn [x] (* 2 x))
        doubled-integers (map (integers) doubled)
        first-n-doubled-integers (take-first n doubled-
integers)]
          (println first-n-doubled-integers)))
```

There is a reason why the `map` function takes the parameters in that order. It is to make function partial application possible.

5.1.12: Naming Class Attributes

Class attributes should be named so that the class name is not repeated in the attribute names. Below is an example of incorrect naming:

Figure 5.26

```
class Order:
    def __init__(self, order_id: int, order_state: OrderState):
        self.__order_id = order_id
        self.__order_state = order_state
```

Below is the above code with corrected names:

Figure 5.27

```
class Order:
    def __init__(self, id_: int, state: OrderState):
        self.__id = id_
        self._state = state
```

If you have an attribute to store a callback function (e.g., event handler or lifecycle callback), you should name it so that it tells on what occasion the stored callback function is called. Name attributes storing event handlers using the following pattern: `on` + `<event-type>`, e.g., `on_click` or `on_submit`. Name attributes storing lifecycle callbacks in a similar way you would name a lifecycle method, for example: `on_init`, `after_mount`, or `before_mount`.

5.1.13: General Naming Rules

5.1.13.1: Use Short, Common Names

When picking a name for something, use the most common shortest name. If you have a function named *relinquish_something*, consider a shorter and more common name for the function. You could rename the function to *release_something*, for example. The word "release" is shorter and more commonly used than the "relinquish" word. Use Google to search for word synonyms, e.g., "relinquish synonym", to find the shortest and most common similar term.

5.1.13.2: Pick One Term And Use It Consistently

Let's assume that you are building a data exporter microservice and you are currently using the following terms in the code: *message*, *report*, *record* and *data*. Instead of using four terms to describe the same thing, you should pick just one term, like *message*, and use it consistently throughout the microservice code. One way to avoid end up having multiple terms for the same thing is to conduct domain-driven design. It enables you to create a ubiquitous language.

5.1.13.3: Avoid Obscure Abbreviations

Many abbreviations are commonly used, like *str* for a string, *num/nbr* for a number, *prop* for a property, or *val* for a value. Most programmers use these, and I use them to make long names shorter. If a variable name is short, the full name should

be used instead, like *number_of_items* instead of *nbr_of_items*. Use abbreviations in cases where the variable name otherwise becomes too long (20 or more characters). What you should especially try to avoid is using uncommon abbreviations. For example, do not abbreviate *amount* to *amnt* or *discount* to *dscnt* because those abbreviations are not really used.

5.1.13.4: Avoid Too Short Or Meaningless Names

Names that are too short do not communicate what the variable is about. Avoid using a single-character variable name like in the following example, where five threads are started:

Figure 5.28

```
for i in range(1, 6):
    start_thread(i)
```

Instead, use a proper variable name to indicate what the loop counter is for:

Figure 5.29

```
for thread_number in range(1, 6):
    start_thread(thread_number)
```

If you don't need to use the loop counter value inside the loop, you can use an underscore as the loop variable name to indicate that it is not used. The below loop loops `object_count` times:

Figure 5.30

```
for _ in range(object_count):
    objects.append(acquire_object())
```

Even if you don't agree with all naming conventions presented here and in the previous chapter, I recommend you create rules for naming code entities, like classes, functions, and variables. That would make your code look consistent and professional. It makes the code look pretty bad if no naming conventions are used, and naming inside a single module or even a function varies dramatically. For example, a 'customers' variable is used somewhere, while a `customer_list` variable is used elsewhere, and the `customers` variable is used to store a list of customers in some place and the number of customers in

another place. It is preferable if a whole development team or, even better, all development teams could share a common set of naming conventions.

5.2: Uniform Source Code Repository Structure Principle

> ***Structuring a source code repository systematically in a certain way makes it easy for other developers to discover wanted information quickly.***

You can create source code repositories for starter projects. Create a source code repository containing a starter project per each technology stack to ensure the uniformity of the repository structures. Below is an example of structuring a source code repository for a Python microservice. In the below example, a containerized (Docker) microservice deployed to a Kubernetes cluster is assumed. Your CI/CD tool might require the CI/CD pipeline code to reside in a specific directory. But if not, place the CI/CD pipeline code in a *cicd* directory.

Figure 5.31

```
my-python-service
├── cicd
│   └── Jenkinsfile
├── docker
│   ├── Dockerfile
│   └── docker-compose.yml
├── docs
├── env
│   ├── .env.dev
│   └── .env.ci
├── helm
│   └── my-python-service
│       ├── templates
│       ├── .helmignore
│       ├── Chart.yaml
│       ├── values.schema.json
│       └── values.yaml
├── integration-tests
│   ├── features
│   │   └── feature1.feature
│   └── steps
├── scripts
│   └── // Bash scripts here...
├── src
├── venv
├── .gitignore
```

```
├── .pylintrc
└── README.MD
```

Unit test modules should be in the same directory as source code modules, but you can also put them in a specific *test* directory.

5.3: Domain-Based Source Code Structure Principle

> ***Structure source code tree primarily by domains, not by technical details. Each source code directory should have a single responsibility at its abstraction level.***

Below is an example of a microservice's *src* directory that is *not* organized by domains but is incorrectly organized according to technical details:

Figure 5.32

```
example-service/
└── src/
    ├── controllers/
    │   ├── AController.py
    │   └── BController.py
    ├── entities/
    │   ├── AEntity.py
    │   └── BEntity.py
    ├── errors/
    │   ├── AError.py
    │   └── BError.py
    ├── dtos/
    │   ├── ADto.py
    │   └── BDto.py
    ├── repositories/
    │   ├── ARepository.py
    │   └── BRepository.py
    └── services/
        ├── AService.py
        └── BService.py
```

Below is the above example modified so that directories are organized by domains:

Figure 5.33

```
example-service/
└── src/
    ├── domainA/
    │   ├── AController.py
```

```
    │   ├── ADto.py
    │   ├── AEntity.py
    │   ├── AError.py
    │   ├── ARepository.py
    │   └── AService.py
    └── domainB/
        ├── BController.py
        ├── BDto.py
        ├── BEntity.py
        ├── BError.py
        ├── BRepository.py
        └── BService.py
```

You can have several levels of nested domains:

Figure 5.34

```
example-service/
└── src/
    ├── domainA/
    │   ├── domainA-1/
    │   │   ├── A1Controller.py
    │   │   └── ...
    │   └── domainA-2/
    │       ├── A2Controller.py
    │       └── ...
    └── domainB/
        ├── BController.py
        └── ...
```

If you want, you can create subdirectories for technical details inside a domain directory. This is the recommended approach if, otherwise, the domain directory would contain more than 5 to 7 files. Below is an example of the *salesitem* domain:

Figure 5.35

```
sales-item-service
└── src
    └── salesitem
        ├── dtos
        │   ├── InputSalesItem.py
        │   └── OutputSalesItem.py
        ├── entities
        │   └── SalesItem.py
        ├── errors
        │   ├── SalesItemServiceError.py
        │   ├── Error1.py
        │   └── Error2.py
        ├── repository
        │   ├── SalesItemRepository.py
        │   └── SalesItemRepositoryImpl.py
        ├── service
        │   ├── SalesItemService.py
        │   └── SalesItemServiceImpl.py
        └── SalesItemController.py
```

To highlight the *clean microservice design principle*, we could also use the following kind of directory layout:

Figure 5.36

```
sales-item-service
└── src
    └── salesitem
        ├── businesslogic
        │   ├── dtos
        │   │   ├── InputSalesItem.py
        │   │   └── OutputSalesItem.py
        │   ├── entities
        │   │   └── SalesItem.py
        │   ├── errors
        │   │   ├── SalesItemServiceError.py
        │   │   ├── Error1.py
        │   │   └── Error2.py
        │   ├── repository
        │   │   └── SalesItemRepository.py
        │   └── service
        │       ├── SalesItemService.py
        │       └── SalesItemServiceImpl.py
        ├── RestSalesItemController.py
        └── SqlSalesItemRepository.py
```

The *businesslogic* directory could also be named *core*, *model*, or even *usecases*, especially if you are also using the *vertical slice design* principle where you put each vertical slice (feature) in a subdirectory of the *usecases* directory.

Or if we have multiple input interface adapters (controllers) and output interface adapters (repositories):

Figure 5.37

```
sales-item-service
└── src
    └── salesitem
        ├── businesslogic
        │   ├── dtos
        │   │   ├── InputSalesItem.py
        │   │   └── OutputSalesItem.py
        │   ├── entities
        │   │   └── SalesItem.py
        │   ├── errors
        │   │   ├── SalesItemServiceError.py
        │   │   ├── Error1.py
        │   │   └── Error2.py
        │   ├── repository
        │   │   └── SalesItemRepository.py
        │   └── service
        │       ├── SalesItemService.py
        │       └── SalesItemServiceImpl.py
        ├── controllers
        │   ├── FlaskRestSalesItemController.py
        │   └── AriadneGraphQlSalesItemController.py
        └── repositories
```

```
        ├── SqlSalesItemRepository.py
        └── MongoDbSalesItemRepository.py
```

If you follow the *clean microservice design* principle and add or change an interface adapter (e.g., a controller or a repository), you should not need to make any code changes to the business logic part of the service (the *businesslogic* directory).

Below is the source code directory structure for the data exporter microservice designed in the previous chapter. There are subdirectories for the four subdomains: input, internal message, transformer, and output. A subdirectory is created for each common nominator in the class names. You can locate a particular file effortlessly by navigating the directory tree. You can grasp the contents of a directory with a glance because the number of source code files in each directory is low. The problem with directories containing many files is that finding the wanted file is not easy. For this reason, a single source code directory should ideally have 2-4 files. The absolute maximum is 5-7 files.

Note that a couple of directories are left unexpanded below to shorten the example. It should be easy for the reader to infer the contents of the unexpanded directories.

Figure 5.38

```
src
├── common
├── input
│   ├── config
│   │   ├── parser
│   │   │   ├── InputConfigParser.py
│   │   │   └── JsonInputConfigParser.py
│   │   ├── reader
│   │   │   ├── InputConfigReader.py
│   │   │   └── LocalFileSystemInputConfigReader.py
│   │   ├── InputConfig.py
│   │   └── InputConfigImpl.py
│   └── message
│       ├── consumer
│       │   ├── InputMsgConsumer.py
│       │   └── KafkaInputMsgConsumer.py
│       ├── decoder
│       │   ├── InputMsgDecoder.py
│       │   └── AvroBinaryInputMsgDecoder.py
│       ├── InputMessage.py
│       └── KafkaInputMessage.py
├── internalmessage
│   ├── field
```

```
│   ├── InternalMessage.py
│   └── InternalMessageImpl.py
├── transformer
│   ├── config
│   ├── field
│   │   ├── impl
│   │   │   ├── CopyFieldTransformer.py
│   │   │   ├── ExprFieldTransformer.py
│   │   │   ├── FilterFieldTransformer.py
│   │   │   └── TypeConvFieldTransformer.py
│   │   ├── FieldTransformer.py
│   │   ├── FieldTransformers.py
│   │   └── FieldTransformersImpl.py
│   └── message
│       ├── MsgTransformer.py
│       └── MsgTransformerImpl.py
└── output
    ├── config
    └── message
        ├── encoder
        └── producer
```

We could also structure the code according to the *clean microservice design* in the following way:

Figure 5.39

```
src
├── common
├── businesslogic
│   ├── input
│   │   ├── config
│   │   │   ├── InputConfig.py
│   │   │   ├── InputConfigImpl.py
│   │   │   ├── InputConfigParser.py
│   │   │   └── InputConfigReader.py
│   │   └── message
│   │       ├── InputMessage.java
│   │       ├── InputMsgConsumer.py
│   │       └── InputMsgDecoder.py
│   ├── internalmessage
│   │   ├── field
│   │   ├── InternalMessage.java
│   │   └── InternalMessageImpl.java
│   ├── transformer
│   │   ├── config
│   │   ├── field
│   │   │   ├── impl
│   │   │   │   ├── CopyFieldTransformer.py
│   │   │   │   ├── ExprFieldTransformer.py
│   │   │   │   ├── FilterFieldTransformer.py
│   │   │   │   └── TypeConvFieldTransformer.py
│   │   │   ├── FieldTransformer.py
│   │   │   ├── FieldTransformers.py
│   │   │   └── FieldTransformersImpl.py
│   │   └── message
│   │       ├── MsgTransformer.py
│   │       └── MsgTransformerImpl.py
│   └── output
│       ├── config
```

```
│       └── message
└── ifadapters
    ├── config
    │   ├── parser
    │   │   └── json
    │   │       ├── JsonInputConfigParser.py
    │   │       ├── JsonTransformerConfigParser.py
    │   │       └── JsonOutputConfigParser.py
    │   └── reader
    │       └── localfilesystem
    │           ├── LocalFileSystemInputConfigReader.py
    │           ├── LocalFileSystemTransformerConfigReader.py
    │           └── LocalFileSystemOutputConfigReader.py
    ├── input
    │   ├── kafka
    │   │   ├── KafkaInputMsgConsumer.py
    │   │   └── KafkaInputMessage.java
    │   └── AvroBinaryInputMsgDecoder.py
    └── output
        ├── CsvOutputMsgEncoder.py
        └── PulsarOutputMsgProducer.py
```

From the above directory structure, we can easily see the following:

- Configurations are in JSON format and read from the local file system
- For the input, Avro binary messages are read from Apache Kafka
- For the output, CSV records are produced to Apache Pulsar

Any change we want or need to make in the *ifadapters* directory should not affect the business logic part in the *businesslogic* directory.

Below is the source code directory structure for the anomaly detection microservice designed in the previous chapter. The *anomaly* directory is expanded. We can see that our implementation is using JSON for various parsing activities and self-organizing maps (SOM) is used for anomaly detection. JSON and Apache Kafka are used to publish anomaly indicators outside the microservice. Adding new concrete implementations to the below directory structure is straightforward. For example, if we wanted to add YAML support for configuration files, we could create *yaml*

subdirectories to place YAML-specific implementation classes.

Figure 5.40

```
src
├── anomaly
│   ├── detection
│   │   ├── configuration
│   │   │   ├── factory
│   │   │   │   ├── AnomalyDetectionConfigFactory.py
│   │   │   │   └── AnomalyDetectionConfigFactoryImpl.py
│   │   │   ├── parser
│   │   │   │   ├── AnomalyDetectionConfigParser.py
│   │   │   │   └── JsonAnomalyDetectionConfigParser.py
│   │   │   ├── AnomalyDetectionConfig.py
│   │   │   └── AnomalyDetectionConfigImpl.py
│   │   ├── engine
│   │   │   ├── AnomalyDetectionEngine.py
│   │   │   └── AnomalyDetectionEngineImpl.py
│   │   ├── rule
│   │   │   ├── factory
│   │   │   │   ├── AnomalyDetectionRuleFactory.py
│   │   │   │   └── AnomalyDetectionRuleFactoryImpl.py
│   │   │   ├── parser
│   │   │   │   ├── AnomalyDetectionRuleParser.py
│   │   │   │   └── AnomalyDetectionRuleParserImpl.py
│   │   │   ├── AnomalyDetectionRule.py
│   │   │   └── AnomalyDetectionRuleImpl.py
│   │   ├── AnomalyDetector.py
│   │   └── AnomalyDetectorImpl.py
│   ├── indicator
│   │   ├── factory
│   │   │   ├── AnomalyIndicatorFactory.py
│   │   │   └── AnomalyIndicatorFactoryImpl.py
│   │   ├── publisher
│   │   │   ├── AnomalyIndicatorPublisher.py
│   │   │   └── KafkaAnomalyIndicatorPublisher.py
│   │   ├── serializer
│   │   │   ├── AnomalyIndicatorSerializer.py
│   │   │   └── JsonAnomalyIndicatorSerializer.py
│   │   ├── AnomalyIndicator.py
│   │   └── AnomalyIndicatorImpl.py
│   └── model
│       ├── factory
│       │   ├── AnomalyModelFactory.py
│       │   └── AnomalyModelFactoryImpl.py
│       ├── training
│       │   ├── engine
│       │   │   ├── AnomalyModelTrainingEngine.py
│       │   │   └── AnomalyModelTrainingEngineImpl.py
│       │   ├── AnomalyModelTrainer.py
│       │   └── SomAnomalyModelTrainer.py
│       ├── AnomalyModel.py
│       └── SomAnomalyModel.py
├── common
├── measurement
├── app.py
```

For full-stack Python developers, let's have one more example with a *data-visualization-web-client*.

This web client's UI consists of the following pages, which all include a common header:

- Dashboards
- Data Explorer
- Alerts

The *Dashboards* page contains a dashboard group selector, dashboard selector, and chart area to display the selected dashboard's charts. You can select the shown dashboard by first selecting a dashboard group and then a dashboard from that group.

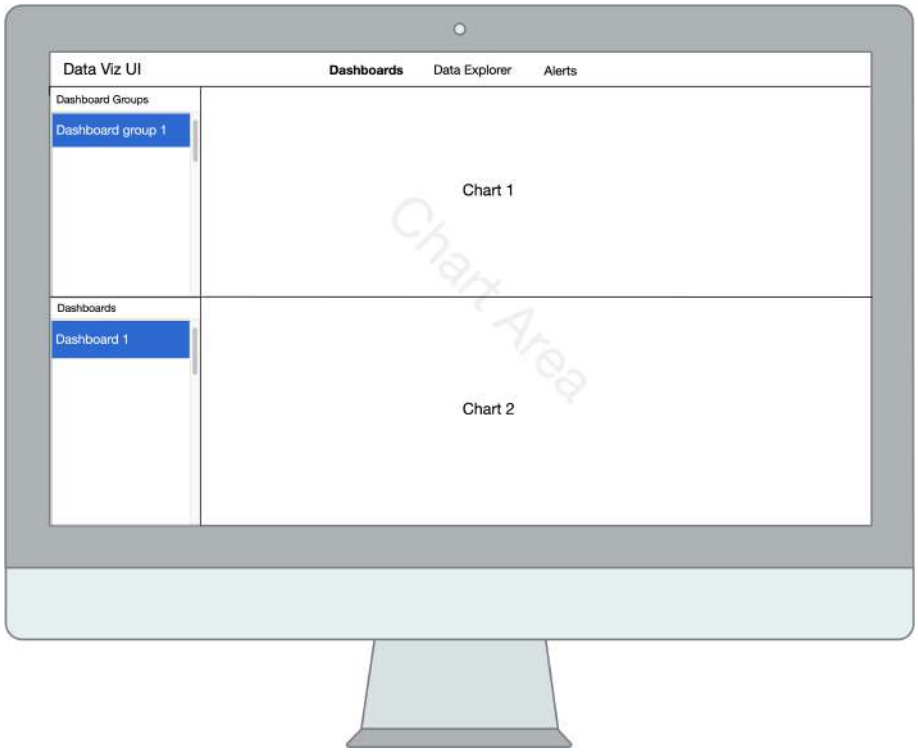

Figure 5.41. Dashboards Page

The *Data Explorer* page contains selectors for choosing a data source, measure(s), and dimension(s). The page also contains a chart area to display charts. Using the selectors, a user can change the shown measure(s) and dimension(s) for the currently selected chart in the chart area.

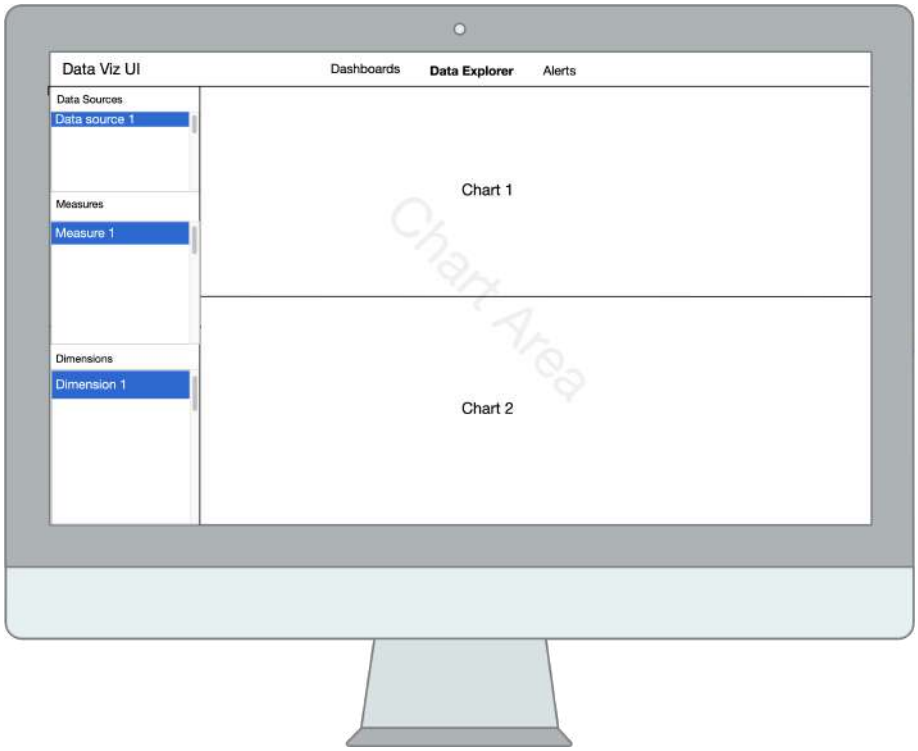

Figure 5.42. Data Explorer Page

Based on the above design, the web client can be divided into the following subdomains:

- Common UI components
  - Chart Area
    - Chart
- Header
- Pages
  - Alerts
  - Dashboards
  - Data Explorer

The source code tree should look like the following:

Figure 5.43

```
src
├── app
│   ├── common
│   │   └── chartarea
│   │       └── chart
│   ├── header
```

```
│   └── pages
│       ├── alerts
│       ├── dashboards
│       │   └── selectors
│       │       ├── dashboardgroup
│       │       └── dashboard
│       └── dataexplorer
│           └── selectors
│               ├── datasource
│               ├── dimension
│               └── measure
│
├── index.ts
└── store.ts
```

Below is an example of what a single subdomain directory would look like when using React, Redux, and SCSS modules:

Figure 5.44

```
src
├── app
│   └── header
│       ├── model
│       │   ├── actions
│       │   │   ├── AbstractHeaderAction.ts
│       │   │   └── NavigateToPageAction.ts
│       │   ├── services
│       │   └── state
│       │       ├── types
│       │       ├── HeaderState.ts
│       │       └── initialHeaderState.ts
│       ├── view
│       │   ├── navigation
│       │   │   ├── NavigationView.module.scss
│       │   │   └── NavigationView.tsx
│       │   ├── HeaderView.module.scss
│       │   └── HeaderView.tsx
│       └── headerController.ts
├── index.ts
└── store.ts
```

In the above example, we have created two directories for the technical details of the *header* domain: model and view directories. The model directory contains actions, services, and the state, and the view directory contains the view component, its possible subcomponents, and CSS definitions. The model's state directory can contain a subdirectory for types used in the subdomain state. The state directory should always contain the type definition for the subdomain's state and the initial state. The services directory contains one or more services that use the respective backend services to control the backend model.

5.4: Avoid Comments Principle

> ***Avoid comments in code. The only exception is when documenting the public API of a library.***

Comments can be problematic. You cannot trust them 100% because they can be misleading, outdated, or downright wrong. You can only trust the source code itself. Comments are often entirely unnecessary and only make the code more verbose. Allowing comments can produce unreadable code containing bad names, which are explained by attaching comments. The code typically also contains long functions where functionality blocks are described with attached comments instead of refactoring the code by extracting well-named functions. The idea behind this principle is that when writing comments is disallowed, you are forced to communicate better with the code only. The following sections describe several ways to avoid writing comments and still keep your code understandable. The following things can be done to avoid writing comments:

- Name things like classes, functions, and variables properly
    - For example, if you are using a particular algorithm, don't document that algorithm in a comment, but name the respective class/function so that it contains the algorithm name. Readers can then google the algorithm by name if unfamiliar with it
- You should not add comments about variable/function types. Use type annotations everywhere
- You don't need to comment that a function can raise an error. Use the function name *try* prefix convention described later in this chapter
- Don't add a comment to a piece of code, but extract a new well-named function
- Keep your functions small. They are easier to understand because they cannot contain too complex logic that could

justify a comment

- Don’t add as a comment information that can be obtained from the version control system
- Don’t comment out the code. Just remove the unused code. The removed code will be available in the version control system forever
- You don’t have to comment on a function’s logic. Code readers should be able to infer that information from the code itself and the related unit tests. Complex code logic and behavior do not usually need comments if you have practiced *test-driven development* and a complete set of well-named unit tests is available

Comments for a public API in a library are needed because the library needs API documentation that can be automatically generated from the comments to avoid situations where API comments and docs are out of sync. API documentation is usually unnecessary in non-library software components because you can access the API interface, implementation, and unit tests. The unit tests, for example, specify what the function does in different scenarios. The unit test name tells the scenario, and the expectations and assertions in the unit test code tell the expected behavior in the particular situation. API implementation and unit tests are not typically available for library users, and even if they are, a user should not adhere to them because they are internal details subject to change.

5.4.1: Name Things Properly

When you name things like a function poorly, you might end up attaching a comment to the function. It is imperative to focus on naming things correctly. When following the *single responsibility principle* and *uniform naming principle*, it should be easier to name things correctly and avoid comments. Below is an example of a function with a comment:

Figure 5.45

```
class MessageBuffer:
    # Return False if buffer full,
    # True if message written to buffer
```

```
    def write(self, message: Message) -> bool:
        # ...
```

If we drop the comment, we will have the following code:

Figure 5.46

```
class MessageBuffer:
    def write(self, message: Message) -> bool:
        # ...
```

Dropping the comment alone is not the best solution because some crucial information is now missing. What does that boolean return value mean? It is not 100% clear. We can assume that returning `True` means the message was successfully written, but nothing about returning `False` is communicated. We can only assume it is some error, but we are not sure what error.

In addition to removing the comment, we should give a better name for the function and rename it as follows:

Figure 5.47

```
class MessageBuffer:
    def write_if_buf_not_full(self, message: Message) -> bool:
        pass
```

Now, the purpose of the function is clear, and we can be sure what the boolean return value means. It means whether the message was written to the buffer. Now we also know why the writing of a message can fail: the buffer is full. This will give the function caller sufficient information about what to do next. It should probably wait a while so the buffer reader has enough time to read messages from the buffer and free up some space.

Below is a real-life example of C++ code where the comment and the function name do not match. Don't worry, you don't need to understand C++ to understand the example.

Figure 5.48

```
    /**
     * @brief Add new counter or get existing, if same labels
used already.
     * @param counterName Name of the counter
     * @param help Help text added for counter, if new
countername
     * @param labels Specific labels for counter.
     * @return counter pointer used when increasing counter, or
```

```
nullptr
     *         if metrics not initialized or invalid name or
labels
     */
    static prometheus::Counter* addCounter(
        std::string counterName,
        std::string help,
        const std::map<std::string, std::string>& labels);
```

In the above example, the function name tells it adds a counter, but the comment says it adds or gets an existing counter. The real problem is that once someone first reads the function name `addCounter`, they do not necessarily read the 'brief' in the comments because they immediately understand what the function does after reading its name: it should add a counter. We could improve the function's name and rename it to `addOrGetExistingCounter` as a solution.

Below is a real-life example from a book that I once read:

Figure 5.49

```
from typing import Protocol

from Person import Person


class Mediator(Protocol):
    # To register an employee
    def register(self, person: Person) -> None:
        pass

    # To send a message from one employee to another employee
    def connect_employees(
        self,
        from_person: Person,
        to_person: Person,
        msg: str
    ) -> None:
        pass

    # To display currently registered members
    def display_detail(self) -> None:
        pass
```

There are three functions in the above example, each of which has a problem. The first function is registering a person, but the comment says it is registering an employee. So, there is a mismatch between the comment and the code. In this case, I trust the code over the comment. The correction is to remove the comment because it does not bring any value. It only confuses.

The second function says in the comment that it sends a message from one employee to another. The function name speaks about connecting employees, but the parameters are persons. I assume that a part of the comment is correct: to send a message from someone to someone else. But once again, I trust the code over the comment and assume the message is sent from one person to another. We should remove the comment and rename the function.

In the third function, the comment adds information missing from the function name. The comment also discusses members, as other parts of the code speak about employees and persons. There are three different terms used: employee, person, and member. Just one term should be picked. Let's choose the term *person* and use it systematically.

Below is the refactored version without the comments:

Figure 5.50

```
class Mediator(Protocol):
    def register(self, person: Person) -> None:
        pass

    def send(message: str, sender: Person, recipient: Person) ->
None:
        pass

    def display_details_of_registered_persons(self) -> None:
        pass
```

5.4.2: Single Return Of Named Value At The End Of Function

> ***A function should have a single return statement and return a named value at the end of the function. This allows the code reader to infer the meaning of the return value by looking at the end of the function.***

Consider the following example:

Figure 5.51

```
from typing import Protocol

from CounterFamily import CounterFamily


class Metrics(Protocol):
```

```
    # ...

    def add_counter(
        self,
        counter_family: CounterFamily,
        labels: dict[str, str]
    ) -> int:
        pass

    def increment_counter(
        self,
        counter_index: int,
        increment_amount: int
    ) -> None:
        pass

    # def add_gauge ...
    # def set_gauge ...
```

What is the return value of the `add_counter` function? Someone might think a comment is needed to describe the return value because it is unclear what `int` means. Instead of writing a comment, we can introduce a named value (= variable/constant) to be returned from the function. The idea behind the named return value is that it communicates the semantics of the return value without the need for a comment. Below is the implementation for the `add_counter` function:

Figure 5.52

```
from typing import Protocol

from CounterFamily import CounterFamily


class Metrics(Protocol):
    def add_counter(
        self,
        counter_family: CounterFamily,
        labels: dict[str, str]
    ) -> int:
        # Perform adding a counter here and
        # set value for the 'counter_index' variable
        return counter_index
```

In the above implementation, we have a single return of a named value at the end of the function. All we have to do is look at the end of the function and spot the return statement, which should tell us the meaning of the mysterious `int` typed return value: It is a counter index. We can spot that the `increase_counter` function requires a `counter_index` argument, and this establishes a connection between calling

the `add_counter` function first, storing the returned counter index, and later using that stored counter index in calls to the `increase_counter` function.

5.4.3: Return Type Aliasing

In the previous example, there was the mysterious return value of type `int` in the `add_counter` function. We learned how introducing a named value returned at the end of the function helped to communicate the semantics of the return value. But there is an even better way to communicate the semantics of a return value. We can use type aliasing. Below is an example where we introduce a `CounterIndex` type alias for the `int` type:

Figure 5.53

```
from typing import Protocol

from CounterFamily import CounterFamily


CounterIndex = int

class Metrics(Protocol):
    # ...

    def add_counter(
        self,
        counter_family: CounterFamily,
        labels: dict[str, str]
    ) -> CounterIndex:
        pass

    def increment_counter(
        self,
        counter_index: CounterIndex,
        increment_amount: int
    ) -> None:
        pass

    # def add_gauge ...
    # def set_gauge_value ...
```

We can improve the above metrics example more. First of all, we should avoid the primitive obsession. We should not be returning an index from the `add_counter` method, but we should rename the method as `create_counter` and return an instance of a `Counter` class from the method. Then we should make the example more object-oriented by moving the

`increment_counter` method to the `Counter` class and renaming it just `increment`. Also, the name of the `Metrics` class should be changed to `MetricFactory`.

5.4.4: Extract Constant for Boolean Expression

By extracting a constant for a boolean expression, we can eliminate comments. Below is an example where a comment is written below an if-statement and its boolean expression:

Figure 5.54

```
from Message import Message


class MessageBuffer:
    def write_if_buf_not_full(self, message: Message) -> bool:
        message_was_written = False
        if len(self.__messages) < self.__max_length:
            # Buffer is not full
            self.__messages.append(message)
            message_was_written = True
        return message_was_written
```

By introducing a constant to be used in the "buffer is full" check, we can get rid of the "Buffer is not full" comment:

Figure 5.55

```
from Message import Message


class MessageBuffer:
    def write_if_buf_not_full(self, message: Message) -> bool:
        message_was_written = False
        buffer_is_not_full = len(self.__messages) <
self.__max_length
        if buffer_is_not_full:
            self.__messages.append(message)
            message_was_written = True
        return message_was_written
```

5.4.5: Extract Named Constant or Enumerated Type

If you encounter a magic number in your code, you should introduce either a named constant or an enumerated type (enum) for that value. In the below example, we are returning two magic numbers, 0 and 1:

Figure 5.56

```
import sys

from Application import Application


application = Application()

if application.run():
    # Application was run successfully
    sys.exit(0)

# Exit code: failure
sys.exit(1)
```

Let's introduce an enumerated type, `ExitCode`, and use it instead of magic numbers:

Figure 5.57

```
import sys
from enum import IntEnum

from Application import Application


class ExitCode(IntEnum):
    Success = 0
    Failure = 1


app_was_run = Application().run()

exit_code = (
    ExitCode.Success if app_was_run else ExitCode.Failure
)

sys.exit(exit_code)
```

It is now easy to add more exit codes with descriptive names later if needed.

5.4.6: Extract Function

If you are planning to write a comment above a piece of code, you should extract that piece of code to a new function. You don't need to write that comment when you extract a well-named function. The name of the newly extracted function serves as documentation. Below is an example with some commented code:

Figure 5.58

```
from Message import Message
```

```
class MessageBuffer:
    def write_fitting(self, messages: list[Message]) -> None:
        if len(self.__messages) + len(messages) <=
self.__max_length:
            # All messages fit in buffer
            self.__messages.extend(messages)
            messages.clear()
        else:
            # All messages do not fit, write only messages that
fit
            nbr_of_msgs_that_fit = self.__max_length -
len(messages)

self.__messages.extend(messages[:nbr_of_msgs_that_fit])
            del messages[:nbr_of_msgs_that_fit]
```

Here is the same code with comments refactored out by extracting two new methods:

Figure 5.59

```
from Message import Message


class MessageBuffer:
    def write_fitting(self, messages: list[Message]) -> None:
        all_messages_fit = (
            len(self.__messages) + len(messages) <=
self.__max_length
        )

        if all_messages_fit:
            self.__write_all(messages)
        else:
            self.__write_only_fitting(messages)

    def __write_all(self, messages: list[Message]) -> None:
        self.__messages.extend(messages)
        messages.clear()

    def __write_only_fitting(self, messages: list[Message]) ->
None:
        nbr_of_msgs_that_fit = self.__max_length - len(messages)
        self.__messages.extend(messages[:nbr_of_msgs_that_fit])
        del messages[:nbr_of_msgs_that_fit]
```

5.4.7: Avoid Comments for Regular Expression

When you have a complex regular expression, you might be tempted to attach a comment to it to explain the meaning of it. But there is an alternative way. You can split the complex regular expression into well-named parts and construct the final regular expression from those well-named parts. For example, suppose you must create a regular expression to

validate phone numbers in the international format. In that case, you can split the regular expression into the following parts: plus sign, country code, area code, and the rest of the phone number (local number). You can define the regular expression using the parts: `intl_phone_nbr_reg_expr = plus_sign + country_code + area_code + local_number`

5.4.8: Avoiding Comments in Bash Shell Scripts

Many programmers, myself included, don't enjoy the mysterious syntax of Linux shell commands and scripts. Even the syntax of the simplest expressions can be hard to understand and remember if you don't regularly work with scripts and commands. Of course, the best thing is to avoid writing complex Linux shell scripts and use a proper programming language like Python instead. But sometimes, performing some actions using a shell script is easier. Because the syntax and commands in shell scripts can be hard to understand, many developers tend to solve the problem by adding comments to scripts.

Next, alternative ways to make scripts more understandable without comments are presented. Let's consider the below example from one real-life script I have bumped into:

Figure 5.60

```
create_network() {
  #create only if not existing yet
  if [[ -z "$(docker network ls | grep $DOCKER_NETWORK_NAME )"
]];
  then
    echo Creating $DOCKER_NETWORK_NAME
    docker network create $DOCKER_NETWORK_NAME
  else
    echo Network $DOCKER_NETWORK_NAME already exists
  fi
}
```

Below is the same example with the following changes:

- The comment was removed, and the earlier commented expression was moved to a well-named function

- The negation in the expression was removed, and the contents of the *then* and *else* branches were swapped
- Variable names were made camel case to enhance readability. All-caps names are harder to read.

Figure 5.61

```
dockerNetworkExists() { [[ -n "$(docker network ls | grep $1 )"
]]; }

createDockerNetwork() {
  if dockerNetworkExists $networkName; then
    echo Docker network $networkName already exists
  else
    echo Creating Docker network $networkName
    docker network create $networkName
  fi
}
```

If your script accepts arguments, give the arguments proper names, for example:

Figure 5.62

```
dataFilePathName=$1
schemaFilePathName=$2
```

The script reader does not have to remember what `$1` or `$2means,` and you don't have to insert any comments to clarify the meaning of the arguments.

If you have a complex command in a Bash shell script, you should not attach a comment to it but extract a function with a proper name to describe the command.

The below example contains a comment:

Figure 5.63

```
# Update version in Helm Chart.yaml file
sed -i "s/^version:.*/version: $VERSION/g"
helm/service/Chart.yaml
```

Here is the above example refactored to contain a function and a call to it:

Figure 5.64

```
updateHelmChartVersionInChartYamlFile() {
  sed -i "s/^version:.*/version: $1/g" helm/service/Chart.yaml
}

updateHelmChartVersionInChartYamlFile $version
```

Here is another example:

Figure 5.65

```
getFileLongestLineLength() {
  echo $(awk '{ if (length($0) > max) max = length($0) } END {
print max }' $1)
}

configFileLongestLineLength = $(getFileLongestLineLength
$configFilePathName)
```

5.5: Function Single Return Principle

> ***Prefer a single return statement at the end of a function to clearly communicate the return value's meaning and make refactoring the function easier.***

A single return statement with a named value at the end of a function clearly communicates the return value semantics if the return value type does not directly communicate it. For example, if you return a value of a primitive type like an integer or boolean from a function, what the return value means is not necessarily 100% clear. But when you return a named value at the end of the function, the name of the returned variable communicates the semantics.

You might think that being unable to return a value in the middle of a function would make the function less readable because of lots of nested if statements. This is possible, but one should remember that a function should be small. Aim to have a maximum of 5-9 lines of statements in a single function. Following that rule, you never have *a hell of nested if-statements* inside a single function.

Having a single return statement at the end of a function makes refactoring the function easier. You can use automated refactoring tools provided by your IDE. It is always harder to extract a new function from code containing a return statement. The same applies to loops with a *break* or *continue* statement. It is easier to refactor code inside a loop that does not contain a break or continue statement.

In some cases, returning a single value at the end of a function makes the code more straightforward and requires fewer lines. Below is an example of a function with two return locations:

Figure 5.66

```
from threading import Thread

from InputMessage import InputMessage


class TransformThread(Thread):
    # ...

    def transform(self, input_message: InputMessage) -> bool:
        output_message =
self.__output_message_pool.acquire_message()

        (
            msg_was_transformed,
            msg_is_filtered_in,
        ) = self.__message_transformer.transform(
            input_message, output_message
        )

         if msg_was_transformed and msg_is_filtered_in:
             self.__output_messages.append(output_message)
         else:

self.__output_message_pool.return_message(output_message)
             if not msg_was_transformed:
                 return False

         return True
```

When analyzing the above function, we notice that it transforms an input message into an output message. We can conclude that the function returns *True* on successful message transformation. We can shorten the function by refactoring it to contain only one return statement. After refactoring, it is also 100% clear what the function return value means.

Figure 5.67

```
from threading import Thread

from InputMessage import InputMessage


class TransformThread(Thread):
    # ...

    def transform(self, input_message: InputMessage) -> bool:
        output_message =
self.__output_message_pool.acquire_message()

         (
```

```
            msg_was_transformed,
            msg_is_filtered_in,
        ) = self.__message_transformer.transform(
            input_message, output_message
        )

        if msg_was_transformed and msg_is_filtered_in:
            self.__output_messages.append(output_message)
        else:

self.__output_message_pool.return_message(output_message)

        return msg_was_transformed
```

As an exception to this rule, you can have multiple return statements in a function when the function has an optimal length (max 5-9 statements) and would become too long if it is refactored to contain a single return statement. Additionally, it is required that the semantic meaning of the return value is evident from the function name or the return type of the function. Below is an example of a function with multiple return statements. It is also evident from the function name what the return value means. Also, the length of the function is optimal: seven statements.

Figure 5.68

```
from typing import Protocol, TypeVar

T = TypeVar('T')


class MyIterator(Protocol[T]):
    def has_next_item(self) -> bool:
        pass

    def get_next_item(self) -> T:
        pass

def are_equal(
    iterator: MyIterator[T],
    another_iterator: MyIterator[T]
) -> bool:
    while iterator.has_next_item():
        if another_iterator.has_next_item():
            if (
                iterator.get_next_item()
                != another_iterator.get_next_item()
            ):
                return False
        else:
            return False

    return True
```

If we refactored the above code to contain a single return statement, the code, as shown below, would become too long (10 statements) to fit in one optimal-sized function. In this case, we should prefer the above code over the below code.

Figure 5.69

```
def are_equal(
    iterator: MyIterator[T],
    another_iterator: MyIterator[T]
) -> bool:
    iters_are_equal = True

    while iterator.has_next_item():
        if another_iterator.has_next_item():
            if (
                iterator.get_next_item()
                != another_iterator.get_next_item()
            ):
                iters_are_equal = False
                break
        else:
            iters_are_equal = False
            break

    return iters_are_equal
```

As the second exception to this rule, you can use multiple return locations in a factory because you know from the factory name what type of objects it creates. Below is an example factory with multiple return statements:

Figure 5.70

```
from enum import Enum


class CarType(Enum):
    AUDI = 1
    BMW = 2
    MERCEDES_BENZ = 3


class Car:
    # ...


class Audi(Car):
    #...


class Bmw(Car):
    # ...


class MercedesBenz(Car):
    # ...
```

```
class CarFactory:
    def create_car(self, car_type: CarType) -> Car:
        match car_type:
            case CarType.AUDI:
                return Audi()
            case CarType.BWM:
                return Bmw()
            case CarType.MERCEDES_BENZ:
                return MercedesBenz()
            case _:
                raise ValueError('Invalid car type')
```

5.6: Use Type Annotations for Production Code Principle

> ***Use type annotations when implementing production software. You can use untyped Python for non-production code like integration, end-to-end, and automated non-functional tests.***

You can manage with a trivial software component without types, but when it grows bigger, and more people are involved, the benefits of static typing become evident.

Let's analyze what potential problems using an untyped language might incur:

- Function arguments might be given in the wrong order
- Function argument might be given with the wrong type
- Function return value type might be misunderstood
- Refactoring code is more difficult
- Forced to write public API comments to describe function signatures
- Type errors are not necessarily found in testing

5.6.1: Function Arguments Might Be Given in Wrong Order

When not using type annotations for functions, you can accidentally give function arguments in the wrong order.

When you use type annotations, this kind of error is less common. Modern IDEs can display inlay hints for parameters in a function call. This is a feature you should consider enabling in your IDE. Those parameter hints might reveal cases where arguments for a function are not given in the correct order.

5.6.2: Function Argument Might Be Given with Wrong Type

You can give a function argument with the wrong type when type annotations are not used. For example, a function requires a string representation of a number, but you provide a number. Naming function arguments correctly can help. Instead of naming a string argument, *amount*, the argument should be named as *amount_string* or *amount_as_string*.

5.6.3: Function Return Value Type Might Be Misunderstood

Determining the function return value type can be difficult. It is not necessarily 100% clear from the name of the function. For example, if you have a function named `get_value`, the return value type is not 100% clear. It might be apparent only if you know the context of the function well. As an improvement, the function should be appropriately named, for example: `get_value_as_string()`, if the returned value is always a string. If the return value type is unclear when looking at the function name, you must analyze the function's source code to determine the return value type. That is unnecessary and error-prone manual work that can be avoided using function return type annotations.

5.6.4: Refactoring Code Is More Difficult

Refactoring code is usually more difficult if you don't have type annotations. But when you have type annotations and make a change to, e.g., a function argument type, you will get type check errors in parts of code where this function is called.

Then, it is easy to refactor those parts. But if you didn't have type annotations and made this same change, you would have to manually find all the places where a change is needed, which is clearly more error-prone.

5.6.5: Forced to Write Public API Comments

You might be forced to document a public API using comments when not using type annotations. This is additional work that could be avoided by using type annotations. Writing API documentation with comments is error-prone. You can accidentally write wrong information in the API documentation or forget to update the documentation when you make changes to the API code itself. Similarly, the API documentation readers can make mistakes. They might not read the API documentation at all. Or they have read it earlier but later misremember it.

5.6.6: Type Errors Are Not Found in Testing

This is the biggest problem. You might think that if you have mistakes in your code related to having correct function arguments with the correct types, testing will reveal those mistakes. This is typically a wrong assumption. Unit tests won't find the issues because you mock other classes and their methods. You can only find the issues in integration testing when you integrate the software component (i.e., test functions calling other real functions instead of mocks). According to the testing pyramid, integration tests only cover a subset of the codebase, less than unit tests. Depending on the code coverage of the integration tests, some function argument order or argument/return value type correctness issues may be left untested and escape to production.

5.7: Refactoring Principle

> ***You cannot write the perfect code on the first try, so you should always reserve some time for future refactoring.***

You must refactor even if you write code for a new software component. Refactoring is not related to legacy codebases only. If you don't refactor, you let technical debt grow in the software. The main idea behind refactoring is that no one can write the perfect code on the first try. Refactoring means that you change code without changing the actual functionality. After refactoring, most tests should still pass, the code is organized differently, and you have a better object-oriented design and improved naming of things. Refactoring does not usually affect integration tests but can affect unit tests depending on the type and scale of refactoring. Keep this in mind when estimating refactoring effort. When you practice test-driven development (TDD), you are bound to refactor. Refactoring is a practice built into the TDD process. In TDD, the final function implementation is a result of a series of refactorings. This is one of the biggest benefits of TDD. We will discuss TDD more in the next chapter.

Software developers don't necessarily reserve any or enough time for refactoring when they plan things. When we provide work estimates for epics, features, and user stories, we should be conscious of the need to refactor and add some extra time to our initial work estimates (which don't include refactoring). When you use TDD, you start automatically reserving some time for refactoring because refactoring is an integral part of the TDD process. Refactoring is work that is not necessarily understood clearly by the management. The management should support the need to refactor even if it does not bring clear added value to an end user. But it brings value by not letting the codebase rot and removing technical debt. If you have software with lots of accumulated technical debt, developing new features and maintaining the software is costly. Also, the quality of the software is lower, which can manifest in several bugs and lowered customer satisfaction.

Below is a list of the most common code smells and refactoring techniques to resolve them:

Code Smell	Refactoring Solution
Non-descriptive name	Rename
Long method	Extract method
Large class	Extract class
Complex expression	Extract constant
Long switch-case or if/elif/else statement	Replace conditionals with polymorphism
Long parameter list	Introduce parameter object
Shotgun surgery	Replace conditionals with polymorphism
Negated boolean condition	Invert if-statement
Anemic object	Creating rich object

There are other refactoring techniques, but the ones explained here are the most relevant and valuable. You can find other refactoring techniques in *Martin Fowler*'s book *Refactoring*. Some of the refactorings presented in that book are pretty self-evident for experienced OOP practitioners; others are also handled in this book but scattered around and described as part of different principles and patterns, and some refactorings are primarily for legacy code only. You don't need them when writing new code with appropriate linter settings active. One example of such a refactoring is "Remove Assignments to Parameters". You don't need that when writing new code because you should have a linting rule that disallows assignments to parameters.

5.7.1: Rename

This is probably the single most used refactoring technique. You often don't get the names right on the first try and need to do renaming. Modern IDEs offer tools that help rename things in the code: interfaces, classes, functions, and variables. The IDE's renaming functionality is always better than the plain old search-and-replace method. When using the search-and-replace method, you can accidentally rename something you do not want to be renamed or don't rename something that should have been renamed.

5.7.2: Extract Method

This is probably the second most used refactoring technique. When you implement a public method of a class, the method quickly grows in the number of code lines. A function should contain a maximum of 5-9 statements to keep it readable and understandable. When a public method is too long, you should extract one or more private methods and call these private methods from the public method. Every modern IDE has an *extract method* refactoring tool that allows you to extract private methods easily. Select the code lines you want to extract to a new method and press the IDE's shortcut key for the *extract method* functionality. Then, give a descriptive name for the extracted method, and you are done. (An additional thing you can do is to organize the arguments of the extracted method in better order before completing the extraction in the IDE). In some cases, the refactoring is not automatic. For example, if the code to be extracted contains a *return*, *break*, or *continue* statement, that affects the execution flow of the function (causing multiple return points). To keep your code refactorable, avoid using *break* and *continue* statements and have only a single return statement at the end of the function.

5.7.3: Extract Class

This refactoring technique is used to make a large class smaller. A large class contains several hundreds of lines of code or tens of methods. To make a class smaller, you need to extract one or more behavioral classes from it and use them in the original class. This refactoring technique allows you to use the *strategy pattern* described in the previous chapter. Let's have a simple example with a `Bird` class, which we assume to be large:

Figure 5.71

```
class Bird:
    def move(self):
        # This method uses the below helper methods
        # ...

    def make_sound(self):
        # This method uses the below helper methods
```

```
        # ...

    # Possibly some other public methods ...

    def __move_helper1(self):
        # ...

    # ...

    def __move_helper4(self):
        # ...

    def make_sound_helper1(self):
        # ...

    # ...

    def make_sound_helper4(self):
        # ...
```

To make the above class smaller, we can extract two behavioral classes: `BirdMover` and `BirdSoundMaker`. After refactoring, we have the following classes:

Figure 5.72

```
from typing import Protocol

class BirdMover(Protocol):
    def move(self):
        pass


class BirdMoverImpl(BirdMover):
    def move(self):
        # ...

    def __move_helper1(self):
            # ...

    # ...

    def __move_helper4(self):
        # ...


class BirdSoundMaker:
    def make_sound(self):
        pass


class BirdSoundMakerImpl(BirdSoundMaker):
    def make_sound(self):
        # ...

    def make_sound_helper1(self):
        # ...

    # ...
```

```
    def make_sound_helper4(self):
        # ...


class Bird:
    def __init__(self, mover: BirdMover, sound_maker:
BirdSoundMaker):
        self.__mover: Final = mover
        self.__sound_maker: Final = sound_maker

    def move(self):
        # Call self.__mover.move()

    def make_sound(self):
        # Call self.__sound_maker.make_sound()

     # Possibly some other public methods ...
```

The above solution allows us to use the *strategy pattern*. We can introduce new classes that implement the `BirdMover` and `BirdSoundMaker` protocols and supply them to the `Bird` class constructor. We can now modify the behavior of the `Bird` class using the *open-closed principle*.

5.7.4: Extract Constant

If you have a complex expression (boolean or numeric), assign the value of the expression to a constant. The name of the constant conveys information about the expression. Below is an example where we make the if-statements read better by extracting expressions to constants:

Figure 5.73

```
# ...

if (
    data_source_selector_is_open
    and measure_selector_is_open
    and dimension_selector_is_open
):
    data_source_selector.style.height = f'{0.2 *
available_height}px'
    measure_selector.style.height =  f'{0.4 *
available_height}px'
    dimension_selector.style.height = f'{0.4 *
availableHeight}px'
elif (
     not data_source_selector_is_open
     and not measure_selector_is_open
     and dimension_selector_is_open
):
    dimension_selector.style.height = f'{available_height}px'
```

Let's extract constants:

Figure 5.74

```
# ...

all_selectors_are_open = (
    data_source_selector_is_open
    and measure_selector_is_open
    and dimension_selector_is_open
)

only_dimension_selector_is_open = (
    not data_source_selector_is_open
    and not measure_selector_is_open
    and dimension_selector_is_open
)

if all_selectors_are_open:
    data_source_selector.style.height = f'{0.2 *
available_height}px'
    measure_selector.style.height =  f'{0.4 *
available_height}px'
    dimension_selector.style.height = f'{0.4 *
availableHeight}px'
elif only_dimension_selector_is_open:
    dimension_selector.style.height = f'{available_height}px'
```

Below is an example where we return a boolean expression:

Figure 5.75

```
class AvroFieldSchema:
    # ...

    def equals(self, other_field_schema: 'AvroFieldSchema') ->
bool:
        return (
            self.__type == other_field_schema.type
            and self.__name[self.__name.find('.') + 1 :]
            == other_field_schema.name[
                other_field_schema.name.find('.') + 1 :
            ]
        )
```

It can be challenging to understand what the boolean expression means. We could improve the function by adding a comment: (We assume that each field name has a root namespace that cannot contain a dot character)

Figure 5.76

```
class AvroFieldSchema:
    # ...

    def equals(self, other_field_schema: 'AvroFieldSchema') ->
bool:
        # Field schemas are equal if field types are equal and
        # field names without the root namespace are equal
```

```
        return (
            self.__type == other_field_schema.type
            and self.__name[self.__name.find('.') + 1 :]
            == other_field_schema.name[
                other_field_schema.name.find('.') + 1 :
            ]
        )
```

But we should not write comments because comments are never 100% trustworthy. It is possible that a comment and the related code are not in synchrony: someone has changed the function without updating the comment or modified only the comment but did not change the function. Let's refactor the above example by removing the comment and extracting multiple constants. The below function is longer than the original, but it is, of course, more readable. If you look at the last two statements of the method, you can understand in what case two field schemas are equal. It should be the compiler's job to make the longer version of the function below as performant as the original function.

Figure 5.77

```
class AvroFieldSchema:
    # ...

    def equals(self, other_field_schema: 'AvroFieldSchema') ->
bool:
        name_without_root_ns = self.__name[self.__name.find('.')
+ 1 :]

        other_field_name_without_root_ns =
other_field_schema.name[
            other_field_schema.name.find('.') + 1 :
        ]

        types_and_names_without_root_ns_are_equal = (
            self.__type == other_field_schema.type
            and name_without_root_ns ==
other_field_name_without_root_ns
        )

        return types_and_names_without_root_ns_are_equal
```

5.7.5: Replace Conditionals with Polymorphism

Suppose you encounter a sizeable match-case statement or if/elif-structure in your code (not considering code in factories). It means your software component does not have a

proper object-oriented design. You should replace the conditionals with polymorphism. When you introduce proper OOD in your software component, you move the functionality from a match statement's case branches to different classes that implement a particular interface. And similarly, you move the code from if and elif-statements to different classes that implement a specific interface. This way, you can eliminate the match-case and if/elif-statements and replace them with polymorphic method calls.

Below is an example of a non-object-oriented design:

Figure 5.78

```
def do_something_with(chart: Chart):
    if chart.type == 'column':
        # do this ...
    elif chart.type == 'pie':
        # do that ...
    elif chart.type == 'geographic-map':
        # do a third thing ...
```

Let's replace the above conditionals with polymorphism:

Figure 5.79

```
from typing import Protocol


class Chart(Protocol):
    def do_something(self) -> None:
        pass


class ColumnChart(Chart):
    def do_something(self) -> None:
        # do this ...


class PieChart(Chart):
    def do_something(self) -> None:
        # do that ...


class GeographicMapChart(Chart):
    def do_something(self) -> None:
        # do a third thing


def do_something_with(chart: Chart):
    chart.do_something()
```

Suppose you are implementing a data visualization application and have many places in your code where you check the chart

type and need to introduce a new chart type. It could mean you must add a new *case* or *elif* statement in many places in the code. This approach is very error-prone. It is called *shotgun surgery* because you need to find all the places in the codebase where the code needs to be modified. What you should do is conduct proper object-oriented design and introduce a new chart class containing the new functionality instead of introducing that new functionality by modifying code in multiple places.

5.7.6: Introduce Parameter Object

If you have more than 5-7 parameters for a function, you should introduce a parameter object to reduce the number of parameters to keep the function signature more readable. Below is an example constructor with too many parameters:

Figure 5.80

```
class KafkaConsumer:
    def __init__(
        self,
        brokers: list[str],
        topics: list[str],
        extra_config_entries: list[str],
        tls_is_used: bool,
        cert_should_be_verified: bool,
        ca_file_path_name: str,
        cert_file_path_name: str,
        key_file_path_name: str
   ):
        # ...
```

Let's group the Transport Layer Security (TLS) related parameters to a parameter class named `TlsOptions`:

Figure 5.81

```
class TlsOptions:
    def __init__(
        self,
        tls_is_used: bool,
        cert_should_be_verified: bool,
        ca_file_path_name: str,
        cert_file_path_name: str,
        key_file_path_name: str
    ):
        # ...
```

Now we can modify the `KafkaConsumer` constructor to utilize the `TlsOptions` parameter class:

Figure 5.82

```
from TlsOptions import TlsOptions


class KafkaConsumer:
    def __init__(
        self,
        brokers: list[str],
        topics: list[str],
        extra_config_entries: list[str],
        tls_options: TlsOptions
   ):
       # ...
```

5.7.7: Invert If-Statement

This is a refactoring that a modern IDE can do for you.

Below is a Python example with a negated boolean expression in the if-statement condition. Notice how difficult the boolean expression reads: `host_mount_folder is not None`. It is a double-negative statement and thus can be difficult to read.

Figure 5.83

```
import os


def get_behave_test_folder(relative_test_folder: str = ''):
    host_mount_folder = os.environ.get("HOST_MOUNT_FOLDER")

    if host_mount_folder is not None:
        final_host_mount_folder = host_mount_folder
        if host_mount_folder.startswith('/mnt/c/'):
            final_host_mount_folder = host_mount_folder.replace(
                '/mnt/c/', '/c/', 1
            )
        behave_test_folder = (
            final_host_mount_folder + '/' + relative_test_folder
        )
    else:
        behave_test_folder = os.getcwd()

    return behave_test_folder
```

Let's refactor the above code so that the if and else statements are inverted:

Figure 5.84

```
def get_behave_test_folder(relative_test_folder: str = ''):
    host_mount_folder = os.environ.get("HOST_MOUNT_FOLDER")
```

```
    if host_mount_folder is None:
        behave_test_folder = os.getcwd()
    else:
        final_host_mount_folder = host_mount_folder
        if host_mount_folder.startswith('/mnt/c/'):
            final_host_mount_folder = host_mount_folder.replace(
                '/mnt/c/', '/c/', 1
            )
        behave_test_folder = (
            final_host_mount_folder + '/' + relative_test_folder
        )

    return behave_test_folder
```

Below is another example:

Figure 5.85

```
if name != 'some name':
    # Do thing 1 ...
else:
    # Do thing 2 ...
```

We should not have a negation in the condition of the if-statement. Let's refactor the above example:

Figure 5.86

```
if name == 'some name':
    # Do thing 2 ...
else:
    # Do thing 1 ...
```

5.7.8: Creating Rich Object

An object can be a so-called anemic object with little or no behavior. The object class might only contain attributes and getters and setters for them, making it a kind of data class only. Sometimes, this is okay if there is no business logic related to the object, but many times, there exists some business logic, but that logic is not implemented in the anemic object class itself but in other places in the code. This kind of software "design" (a better term is lack of design) is deemed to be problematic when you need to change the anemic object class. You might need to make changes to various unrelated places in the code. Making all these changes is manual and error-prone work. Making those changes is called shotgun surgery. What you should do is make an anemic object a rich object. This is done by moving behavior from other classes to

the anemic class, making it a rich class. A rich class typically has attributes and behavior but it lacks getters and setters to enforce proper encapsulation of the object state.

Let's have an example of a anemic class:

Figure 5.87

```
class Rectangle:
    def __init__(self, width: float, height: float):
        self.__width = width
        self.__height = height

    @property
    def width(self) -> float:
        return self.__width

    @property
    def height(self) -> float:
        return self.__height

    @width.setter
    def width(self, width: float) -> None:
        self.__width = width

    @height.setter
    def height(self, height: float) -> None:
        self.__height = height


class Drawer:
    # ...

    def draw_rectangle(rectangle: Rectangle) -> None:
        # ...

class Calculator:
    # ...

    def calc_rect_area(rectangle: Rectangle) -> float:
        # ...

    def calc_rect_perimeter(rectangle: Rectangle) -> float:
        # ...

class Sizer:
    # ...

    def increase_rect_width(
        rectangle: Rectangle,
        percentage: float
    ) -> None:
        # ...

     def increase_rect_height(
        rectangle: Rectangle,
        percentage: float
    ) -> None:
        # ...
```

What we should do is to make the `Rectangle` objects rich objects by moving functionality from various classes into the `Rectangle class` and removing the getters and setters:

Figure 5.88

```
class Rectangle:
    def __init__(self, width: float, height: float):
        self.__width = width
        self.__height = height

    def draw(self) -> None:
        # Implement drawing here ...

    def calculate_area(self) -> float:
        # Calculate area here ...

    def calculate_perimeter(self) -> float:
        # Calculate perimeter here ...

    def increase_width(self, percentage: float) -> None:
        # Increase width here ...

    def increase_height(self, percentage: float) -> None:
        # Increase height here ...
```

Let's imagine that the methods added to the `Rectangle` class are large, and the whole class becomes too large. In that case, we can use the *bridge pattern* and refactor the classes as shown below. We still have a rich `Rectangle` class with attached behaviour and we do not need any getters or setters because we are not passing the `self` to any outside object of the `Rectangle` class.

Figure 5.89

```
class Drawer:
    # ...

    def draw_rectangle(self, width: float, height: float) ->
None:
        pass


class Calculator:
    # ...

    def calc_rect_area(self, width: float, height: float) ->
float:
        pass

    def calc_rect_perimeter(self, width: float, height: float) -
> float:
        pass


class Sizer:
```

```
    # ...

    def increase(self, length: float, percentage: float) ->
float:
        pass


class Rectangle:
    def __init__(
        self,
        width: float,
        height: float,
        drawer: Drawer,
        calculator: Calculator,
        sizer: Sizer,
    ):
        self.__width = width
        self.__height = height
        self.__drawer = drawer
        self.__calculator = calculator
        self.__sizer = sizer

    def draw(self) -> None:
        self.__drawer.draw_rectangle(self.__width,
self.__height)

    def calculate_area(self) -> float:
        return self.__calculator.calc_rect_area(
            self.__width, self.__height
        )

    def calculate_perimeter(self) -> float:
        return self.__calculator.calc_rect_perimeter(
            self.__width, self.__height
        )

    def increase_width(self, percentage: float) -> None:
        self.__width = self.__sizer.increase(self.__width,
percentage)

    def increase_height(self, percentage: float) -> None:
        self.__height = self.__sizer.increase(self.__height,
percentage)
```

You can see from the above code that we no longer need to perform *shotgun surgery* if we need to change the `Rectangle` class. For example, let's say that our rectangles should always have a height double the width. We can refactor the `Rectangle` class to the following:

Figure 5.90

```
class Rectangle:
    def __init__(
        self,
        width: float,
        drawer: Drawer,
        calculator: Calculator,
        sizer: Sizer,
```

```
    ):
        self.__width = width
        self.__height = 2 * width
        self.__drawer = drawer
        self.__calculator = calculator
        self.__sizer = sizer

    def draw(self) -> None:
        self.__drawer.draw_rectangle(self.__width,
self.__height)

    def calculate_area(self) -> float:
        return self.__calculator.calc_rect_area(
            self.__width, self.__height
        )

    def calculate_perimeter(self) -> float:
        return self.__calculator.calc_rect_perimeter(
            self.__width, self.__height
        )

    def increase_width(self, percentage: float) -> None:
        self.__width = self.__sizer.increase(self.__width,
percentage)
        self.__height = 2 * self.__width

    def increase_height(self, percentage: float) -> None:
        self.__height = self.__sizer.increase(self.__height,
percentage)
        self.__width = self.__height / 2
```

5.8: Static Code Analysis Principle

> ***Let the computer find bugs and issues in the code for you.***

Static code analysis tools find bugs and design-related issues on your behalf. Use multiple static code analysis tools to get the full benefit. Different tools might detect different issues. Using static code analysis tools frees people's time in code reviews to focus on things that automation cannot tackle.

Below is a list of some popular static code analysis tools for Python:

- PyLint
- Ruff
- SonarLint

- SonarQube/SonarCloud
- Black (Code formatter)
- Blue (Code formatter)
- Mypy
- Jetbrains PyCharm inspections

Infrastructure and deployment code should be treated the same way as source code. Remember to run static code analysis tools on your infrastructure and deployment code, too. Several tools are available for analyzing infrastructure and deployment code, like Checkcov, which can be used for analyzing Terraform, Kubernetes, and Helm code. Helm tool contains a linting command to analyze Helm chart files, and Hadolint is a tool for analyzing *Dockerfiles* statically.

5.8.1: Common Static Code Analysis Issues

Issue	Description/Solution
Chain of instance of checks	This issue indicates a chain of conditionals in favor of object-oriented design. Use the *replace conditionals with polymorphism* refactoring technique to solve this issue.
Feature envy	Use the *don't ask, tell principle* from the previous chapter to solve this issue.
Use of concrete classes	Use the *program against interfaces* principle from the previous chapter to solve this issue.
Assignment to a function argument	Don't modify function arguments but introduce a new variable.
Commented-out code	Remove the commented-out code. If you need that piece of code in the future, it is available in the version control system forever.
Const correctness	Make attributes and variables `Final` whenever possible to achieve immutability and avoid accidental modifications.
Nested match statement	Use match statements mainly only in factories. Do not nest them.
Nested conditional expression (ternary operator)	Conditional expression should not be nested because it greatly hinders the code readability.
Overly complex boolean expression	Split the boolean expression into parts and introduce constants to store the parts and the final expression.
Expression can be simplified	This can be refactored automatically by an IDE.
Match statement without default branch	Always introduce a default branch and raise an exception there. Otherwise, when you use a match statement with an enum, you might encounter strange problems after adding a new enum value not handled by the match statement.

Issue	Description/Solution
Law of Demeter	The object knows too much. It is coupled to the dependencies of another object, which creates additional coupling and makes code harder to change. Making code easy to change is an objective, but not needing to change code is an even greater objective. (Remember the *open-closed principle*)
Reuse of local variable	Introduce a new variable instead of reusing a variable for a different purpose. That new variable can be named appropriately to describe its purpose.
Scope of variable is too broad	Introduce a variable only just before it is needed.
Protected field	Subclasses can modify the protected state of the superclass without the superclass being able to control that. This is an indication of breaking the encapsulation and should be avoided.
Breaking the encapsulation: Return of modifiable/mutable field	Use the *don't leak modifiable internal state outside an object principle* from the previous chapter to solve this issue.
Breaking the encapsulation: Assignment from a method parameter to a modifiable/mutable field	Use the *don't assign from a method parameter to a modifiable field principle* from the previous chapter to solve this issue.
Non-constant public field	Anyone can modify a public field. This breaks the encapsulation and should be avoided.

Issue	Description/Solution
Overly broad except-block	This can indicate a wrong design. Don't catch the language's base exception class if you should only catch your application's base error class, for example. Read more about handling exceptions in the next section.

5.9: Error/Exception Handling Principle

Python and many other languages like C++, Java, and JavaScript/TypeScript have an exception-handling mechanism that can handle errors and exceptional situations. First of all, I want to make a clear distinction between these two words:

> ***An error can happen, and one should be prepared for it.***
> ***An exception is something that should never happen.***

You define errors in your code and raise them in your functions. For example, if you try to write to a file, you must be prepared for the error that the disk is full, or if you are reading a file, you must be prepared for the error that the file does not exist (anymore).

Some errors are recoverable. You can delete files from the disk to free up some space to write to a file. Or, in case a file is not found, you can give a "file not found" error to the user, who can then retry the operation using a different file name, for example.

You usually don't need to define your own exceptions in your application, but the system raises built-in exceptions in *exceptional situations*, like when a programming error is encountered. An exception can be raised, for example, when memory is low, and memory allocation cannot be performed, or when a programming error results in an array index out of bounds or an object does not have a specific attribute. When an exception is raised, the program cannot continue executing

normally and might need to terminate. This is why many exceptions can be categorized as unrecoverable errors. In some cases, it is possible to recover from exceptions. Suppose a web service encounters an exception while handling an HTTP request. In that case, you can terminate the handling of the current request, return an error response to the client, and continue handling further requests normally. It depends on the software component how it should handle exceptional situations. This is important to consider when designing a new software component. You should define (and document) how the software component should handle exceptions. Should it terminate the process or perhaps do something else?

Errors define situations where the execution of a function fails for some reason. Typical examples of errors are a file not found error, an error in sending an HTTP request to a remote service, or a failure to parse a configuration file. Suppose a function can raise an error. The function caller can decide how to handle the error depending on the error. With transient errors, like a failing network request, the function caller can wait a while and call the function again. Or, the function caller can use a default value. For example, if a function tries to load a configuration file that does not exist, it can use some default configuration instead. In some cases, the function caller cannot do anything but leave the error unhandled or expect the error but raise another error at a higher level of abstraction. Suppose a function tries to load a configuration file, but the loading fails, and no default configuration exists. In that case, the function cannot do anything but pass the error to its caller. Eventually, this error bubbles up in the call stack, and the whole process is terminated due to the inability to load the configuration. This is because the configuration is needed to run the application. Without configuration, the application cannot do anything but exit.

When defining error classes, define a base error class for your software component. You can name the base error class according to the name of the software component. For example, for the data exporter microservice, you can define a `DataExporterError` (or `DataExporterServiceError)` base error class. For *common-utils-lib*, you can define

`CommonUtilsError` (or `CommonUtilsLibError`), and for *sales-item-service*, you can define `SalesItemServiceError`. The popular requests library implements this convention. It defines a `requests.RequestException`, which is the base class for all other errors the library methods can raise. Many other Python libraries also define a common base error class.

For each function that can raise an error, define a base error class at the same abstraction level as the function. That error class should extend the software component's base error class. For example, if you have a `parse(config_str)` function in the `ConfigParser` class, define a base error class for the function inside the class with the name `ParseError`, i.e., `ConfigParser.ParseError`. If you have a `read_file` function in the `FileReader` class, define a base error class in the `FileReader` class with the name `ReadFileError`, i.e., `FileReader.ReadFileError`. If all the methods in a class can raise the same error, it is enough to define only one error at the class level. For example, if you have a `HttpClient` class where all methods like `get`, `post`, `put` etc., can raise an error, you can only define a single `Error` error class in the `HttpClient` class.

Below is an example of errors defined for the data exporter microservice:

Figure 5.91

```
# Base error class for the software component
class DataExporterError(Exception):
    pass


class FileReader:
    # Base error class for the 'read_file' method
    # Extends the base error class of the software component
    class ReadFileError(DataExporterError):
        pass

    def read_file(self, file_path_name: str):
        # ...
        # raise self.ReadFileError()


class ConfigParser:
    # Base error class for the 'parse' method
    # Extends the base error class of the software component
    class ParserError(DataExporterError):
        pass

    def parse(self, config_str: str):
```

```
        # ...
        # raise self.ParseError()
```

Following the previous rules makes catching errors in the code easy because you can infer the error class name from the called method name. In the below example, we can infer the `ReadFileError` error class name from the `read_file` method name:

Figure 5.92

```
try:
    file_contents = file_reader.read_file(...)
except FileReader.ReadFileError as error:
    # Handle error ...
```

You can also catch all user-defined errors using the software component's base error class in the *except* clause.

Figure 5.93

```
try:
    config_string = file_reader.read_file(...)
    return config_parser.parse(config_string)
except DataExporterError as error:
    # Handle error situation
```

Don't catch the language's base exception class or another too-generic exception class because that will catch exceptions and all user-defined errors. It would catch exceptions like `MemoryError` or `ZeroDivisionError`, which is probably not what you want. So, do not catch a too-generic exception class like this:

Figure 5.94

```
try:
    config_string = file_reader.read_file(...)
    return config_parser.parse(config_string)
except BaseException as error:
    # Don't do this!
```

Catch all exceptions only in a special place(s) in your code, like in the main function or the main loop, like the loop in a web server processing HTTP requests or the main loop of a thread. Below is an example of correctly catching the language's base exception class in the main function. When you catch an unrecoverable exception in the main function, log it and exit the process with an appropriate error code. When

you catch an unrecoverable error in a main loop, log it and continue the loop if possible.

Figure 5.95

```
try:
    application.run(...)
except BaseException as exception:
    logger.log(exception)
    sys.exit(1)
else:
    sys.exit(0)
```

Using the above-described rules, you can make your code future-proof or forward-compatible so that adding new errors to be raised in a function in the future is possible. Let's say that you are using a `fetch_config` function like this:

Figure 5.96

```
try:
    configuration = config_fetcher.fetch_config(url)
except ConfigFetcher.FetchConfigError as error:
    # Handle error ...
```

Your code should still work if a new type of error is raised in the `fetch_config` function. Let's say that the following new errors could be raised in the `fetch_config` function:

- Malformed URL error
- Server not found error
- Connection timeout error

When classes for these new errors are implemented, they must extend the function's base error class, in this case, the `FetchConfigError` class. Below are the error classes defined:

Figure 5.97

```
from Config import Config
From DataExporterError import DataExporterError


class ConfigFetcher:
    # Base error class for 'fetch_config' method
    # Extends the base error class of the software component
    class FetchConfigError(DataExporterError):
        pass

    # An additional error for 'fetch_config' method
    # Extends the base error class of the method
    class MalformedUrlError(FetchConfigError):
```

```
        pass

    # An additional error for 'fetch_config' method
    # Extends the base error class of the method
    class ServerNotFoundError(FetchConfigError):
        pass

    # An additional error for 'fetch_config' method
    # Extends the base error class of the method
    class ConnectionTimeoutError(FetchConfigError):
        pass

    def fetch_config(self, url: str) -> Config:
        # ...
        # raise self.MalformedUrlError()
        # raise self.ServerNotFoundError()
        # raise self.ConnectionTimeoutError()
```

You can enhance your code at any time to handle different errors raised from the `fetch_config` method differently. For example, you might want to handle a `ConnectionTimeoutError` so that the function will wait a while and then retry the operation because the error is usually transient:

Figure 5.98

```
try:
    configuration = config_fetcher.fetch_config(url)
except ConfigFetcher.ConnectionTimeoutError as error):
    # Retry after a while
except ConfigFetcher.MalformedUrlError as error:
    # Inform caller that URL should be checked
except ConfigFetcher.ServerNotFoundError as error:
    # Inform caller that URL host/port cannot be reached
except ConfigFetcher.FetchConfigError as error:
    # Handle possible other error situations
    # This will catch any error that could be raised
    # in 'fetch_config' method now and in the future
```

In the above examples, we handled raised errors correctly, but you can easily forget to handle a raised error. This is because nothing in the function signature tells you whether the function can raise an error. The only way to find out is to check the documentation (if available) or investigate the source code (if available). This is one of the biggest problems regarding exception handling because you must know and remember that a function can raise an error, and you must remember to catch and handle errors. You don’t always want to handle an error immediately, but still, you must be aware that the error will

bubble up in the call stack and should be dealt with eventually somewhere in the code.

Below is an example extracted from the documentation of the popular *requests* library:

Figure 5.99

```
import requests

r = requests.get('https://api.github.com/events')
r.json()
# [{'repository': {'open_issues': 0, 'url':
'https://github.com/...
```

Did you know that both `requests.get` and `r.json` can raise an error? Unfortunately, the above documentation extract does not include error handling at all. If you copy-paste the above code sample directly into your production code, you forget to handle errors. If you go to the API reference documentation of the *requests* library, you can find the documentation for the `get` method. That documentation (when writing this book) does not tell that the method can raise an error. The documentation only speaks about the method parameters, return value, and its type. Scrolling down the documentation page, you will find a section about exceptions. But what if you don't scroll down? You might think that the method does not raise an error. The `get` method documentation should be corrected to tell that the method can raise an error and contain a link to the section where possible errors are described.

The above-described problem can be mitigated, at least on some level, when practicing *test-driven development* (TDD). TDD will be described in the next chapter, which covers testing-related principles. In TDD, you define the tests before the implementation, forcing you to think about error scenarios and make tests for them. When you have tests for error scenarios, leaving those scenarios unhandled in the actual implementation code is impossible.

One great solution to the problem that error handling might be forgotten is to make raising errors more explicit:

> ***Use a 'try' prefix in a function name if the function can raise an error.***

This is a straightforward rule. If a function can raise an error, name the function so that its name starts with `try`. This makes it clear to every caller that the function can raise an error, and the caller should be prepared for that. For the caller of the function, there are three alternatives to deal with a raised error:

1. Catch the base error class of the called function (or software component) and handle the error, e.g., catch `DataFetcher.FetchDataError` if you are calling a method named `try_fetch_data` in a class named `DataFetcher`.
2. Catch the base error class of the called function (or software component) and raise a new error on a higher level of abstraction. You must also name the calling function with a `try` prefix.
3. Don't catch errors. Let them propagate upwards in the call stack. You must also name the calling function with a `try` prefix.

Here is an example of alternative 1:

Figure 5.100

```
from Config import Config
from ConfigParser import ConfigParser
from DataFetcher import DataFetcher


class ConfigFetcher:
    def fetch_config(self, url: str) -> Config:
        try:
            config_str = self.__data_fetcher.try_fetch_data(url)
            return self.__config_parser.try_parse(config_str)
        except (
            DataFetcher.FetchDataError,
            ConfigParser.ParseError
        ) as error:
            # You could also catch errors in two different
            # except blocks
            # You could also catch the base error class
'DataExporterError'
            # of the software component
```

And here is an example of alternative 2:

Figure 5.101

```
from Config import Config
from ConfigParser import ConfigParser
from DataFetcher import DataFetcher
```

```
class ConfigFetcher:
    class FetchConfigError(DataExporterError):
        pass

    def try_fetch_config(self, url: str) -> Config:
        try:
            config_str = self.__data_fetcher.try_fetch_data(url)
            return self.__config_parser.try_parse(config_str)
        except (
            DataFetcher.FetchDataError,
            ConfigParser.ParseError
        ) as error:
            # Error on higher level of abstraction is raised
            # This function must be named with the 'try' prefix
            # to indicate that it can raise an error
            raise self.FetchConfigError(error)
```

And here is an example of alternative 3:

Figure 5.102

```
from Config import Config


class ConfigFetcher:
    def try_fetch_config(self, url: str) -> Config:
        # No try-except, all raised errors from both
try_fetch_data
        # and try_parse method calls propagate
        # to the caller and
        # this function must be named with the 'try' prefix
        # to indicate that it can raise an error
        config_str = self.__data_fetcher.try_fetch_data(url)
        return self.__config_parser.try_parse(config_str)
```

Figure 5.103

```
from DataExporterError import DataExporterError


class DataExporter:
    def initialize(self) -> None:
        try:
            config = self.__config_fetcher.try_fetch_config(url)
        except DataExporterError as error:
            # In this case you must catch the base error class
of
            # the software component (DataExporterError),
because
            # you don't know what errors try_fetch_config can
            # raise, because no FetchConfigError class
            # has been defined in the ConfigFetcher class
```

If we go back to the *requests* library usage example, the error-raising methods `requests.get` and `Response.json` could be renamed to `requests.try_get` and

`Response.try_parse_json`. That would make the earlier example look like the following:

Figure 5.104

```
import requests

r = requests.try_get('https://api.github.com/events')
r.try_parse_json()
# [{'repository': {'open_issues': 0, 'url':
'https://github.com/...
```

Now, we can see that the two methods can raise an error. It is easier to remember to put them inside a try/except-block:

Figure 5.105

```
import requests

try:
    r = requests.try_get('https://api.github.com/events')
    r.try_parse_json()
    # [{'repository': {'open_issues': 0, 'url':
'https://github.com/...
except ...
    # ...
```

To make the try-prefix convention even better, a linting rule that enforces the correct naming of error-raising functions could be developed. The rule should force the function name to have a *try* prefix if the function raises or propagates errors. A function propagates errors when it calls an error-raising (try-prefixed) method outside a try-except block.

You can also create a library that has try-prefixed functions that wrap error-raising functions that don't follow the try-prefix rule:

Figure 5.106

```
import json


class JsonParser:
    class ParseError(Exception):
        pass

    @staticmethod
    def try_parse(json_string: str, **kwargs):
        try:
            return json.loads(json_string, **kwargs)
        except json.JSONDecodeError as error:
            raise JsonParser.ParseError(error)
```

Now, if you use the `JsonParser's try_parse` method, you can easily infer the class name of the possibly raised errors without the need to consult any documentation.

When using a web framework, the framework usually provides an error-handling mechanism. The framework catches all possible errors and exceptions when processing a request and maps them to HTTP responses with HTTP status codes indicating a failure. Typically, the default status code is 500 *Internal Server Error*. When you utilize the web framework's error-handling mechanism, there is no significant benefit in naming error-raising functions with the try-prefix because it won't be problematic if you forget to catch an error. So, you can opt out of the try-prefix rule. Many times, this is what you want to do: pass the error to the web framework's error handler. Usually, you provide your own error handler instead of using the default one, so you get responses in the format you want. We will discuss API error handling again in the *API design principles* chapter.

It is usually a good practice to document the error handling mechanism used in the software component's documentation.

The best way to avoid forgetting to handle errors is to practice rigorous *test-driven development* (TDD) described in the next chapter. Another great way to not forget to handle errors is to walk through the code line by line and check if the particular line can produce an error. If it can produce an error, what kind of error, and are there multiple errors the line can produce? Let's have an example with the following code (we focus only on possible errors, not what the function does):

Figure 5.107

```
from typing import Any

import requests
from jwt import PyJWKClient, decode

class JwtAuthorizer:
    # ...

    def __try_get_jwt_claims(
            self, auth_header: str | None
        ) -> dict[str, Any]:
            if not self.__jwks_client:
                oidc_config_response =
requests.get(self.__oidc_config_url)
```

```
                oidc_config = oidc_config_response.json()
                self.__jwks_client =
PyJWKClient(oidc_config['jwks_uri'])

            jwt = auth_header.split('Bearer ')[1] if auth_header
else ''
            signing_key =
self.__jwks_client.get_signing_key_from_jwt(jwt)
            jwt_claims = decode(jwt, signing_key.key,
algorithms=['RS256'])
            return jwt_claims
```

The code on the first line cannot produce an error. On the second line, the `requests.get` method can raise an error on connection failure, for example. Can it produce other errors? It can produce the following errors:

- Malformed URL (`requests.URLRequired`)
- Connection error (`requests.ConnectionError`)
- Connection timeout (`requests.ConnectTimeout`)
- Read timeout (`requests.ReadTimeout`)

It can also produce an error response, e.g., an internal server error. Our code does not handle that currently, which is why we should add the following line after the `requests.get` method call: `oidc_config_response.raise_for_status()`, which will raise an `HttpError` if the response status code is >= 400. The third line can raise a `JSONDecodeError` if the response is not valid JSON. The fourth line can raise a `KeyError` because it is possible that the key `jwks_uri` does not exist in the response JSON. The fifth line can raise an `IndexError` because the list returned by the `split` does not necessarily have an element at index one. Also, the sixth line can raise an error when the JWKS client cannot connect to the IAM system or the JWT is invalid. The second last line can raise an `InvalidTokenError` when the JWT is invalid. In summary, all the lines in the above code can produce at least one kind of error except the first and last lines.

Let's modify the code to implement error handling instead of passing all possible errors and exceptions to the caller:

Figure 5.108

```
from typing import Any

import requests
from jwt import PyJWKClient, PyJWKClientError, decode
from jwt.exceptions import InvalidTokenError


class JwtAuthorizer:
    class GetJwtClaimsError(Exception):
        pass

    def __try_get_jwt_claims(
        self, auth_header: str | None
    ) -> dict[str, Any]:
        try:
            if not self.__jwks_client:
                oidc_config_response =
requests.get(self.__oidc_config_url)
                oidc_config_response.raise_for_status()
                oidc_config = oidc_config_response.json()
                self.__jwks_client =
PyJWKClient(oidc_config['jwks_uri'])

            jwt = auth_header.split('Bearer ')[1] if auth_header
else ''
            signing_key =
self.__jwks_client.get_signing_key_from_jwt(jwt)
            jwt_claims = decode(jwt, signing_key.key,
algorithms=['RS256'])
            return jwt_claims
        except (
            # RequestException is the base error for all errors
            # in the requests library
            requests.RequestException,
            KeyError,
            IndexError,
            PyJWKClientError,
            # Base error when decode() fails on a token
            InvalidTokenError,
        ) as error:
            raise self.GetJwtClaimsError(error)
```

Make yourself a habit of walking through the code of a function line by line once you think it is ready to find out if you have accidentally missed handling some error.

5.9.1: Returning Errors

As an alternative to raising errors, it is possible to communicate erroneous behavior to the function caller using a return value. Using an exception-handling mechanism provides some advantages over returning errors. When a function can return an error, you must always check for the

error right after the function call. This can cause the code to contain nested if-statements, which hinders code readability. The exception-handling mechanism allows you to propagate an error to a higher level in the call stack. You can also execute multiple function calls that can fail inside a single `try` block and provide a single error handler in the `except` block.

5.9.1.1: Returning Failure Indicator

You can return a failure indicator from a failable function when the function does not need to return any additional value. It is enough to return a failure indicator from the function when there is no need to return any specific error code or message. This can be because there is only one reason the function can fail, or function callers are not interested in error details. To return a failure indicator, return a boolean value from the function: *True* means a successful operation, and *False* indicates a failure:

Figure 5.109

```
def perform_task(...) -> bool:
    # Perform the task and set the value of
    # the 'task_was_performed' variable

    return task_was_performed
```

5.9.1.2: Returning an Optional Value

Suppose a function should return a value, but the function call can fail, and there is precisely one cause why the function call can fail. In this case, return an optional value from the function. In the below example, getting a value from the cache can only fail when no value for a specific key is stored in the cache. We don’t need to return any error code or message.

Figure 5.110

```
from typing import Protocol, TypeVar

TKey = TypeVar('TKey')
TValue = TypeVar('TValue')


class Cache(Protocol[TKey, TValue]):
    def add(self, key: TKey, value: TValue) -> None:
        pass
```

```
    def get(self, key: TKey) -> TValue | None:
        pass
```

Or, if you want to use a more functional approach, return an `Optional` object. (The `Optional` class was defined in the previous chapter)

Figure 5.111

```
from typing import Protocol, TypeVar

from Optional import Optional

TKey = TypeVar('TKey')
TValue = TypeVar('TValue')


class Cache(Protocol[TKey, TValue]):
    def add(self, key: TKey, value: TValue) -> None:
        pass

    def get(self, key: TKey) -> Optional[TValue]:
        pass
```

5.9.1.3: Returning an Error Object

When you need to provide details about an error to the function caller, you can return an error object from the function:

Figure 5.112

```
from dataclasses import dataclass


@dataclass
class BackendError:
    http_status_code: int
    error_code: int
    message: str
```

If a function does not return any value but can produce an error, you can return either an error object or *None*:

Figure 5.113

```
from typing import Awaitable, TypeVar

from BackendError import BackendError
from Entity import Entity

T = TypeVar('T', bound=Entity)
```

```
class DataStore:
    async def update_entity(
        self,
        id_: int,
        entity: T
    ) -> Awaitable[BackendError | None]:
        # ...
```

Alternatively, return an optional error, as shown below. (The `Optional` class was defined in the previous chapter)

Figure 5.114

```
from typing import Awaitable, TypeError

from BackendError import BackendError
from Entity import Entity
from Optional import Optional

T = TypeVar('T', bound=Entity)


class DataStore:
    async def update*entity(
        self,
        id*: int,
        entity: T
    ) -> Awaitable[Optional[BackendError]]:
        # ...
```

Suppose a function needs to return a value or an error. In that case, you can use a 2-tuple (i.e., a pair) type, where the first value in the tuple is the actual value or *None* in case of an error and the second value in the tuple is an error object or *None* value in case of a successful operation. Below is an example.

Figure 5.115

```
from typing import Awaitable, TypeVar, Union

from BackendError import BackendError
from Entity import Entity

T = TypeVar('T', bound=Entity)


class DataStore:
    async def create_entity(
        self,
        entity: T
    ) ->  Awaitable[Union[(T, None), (None, BackendError)]]:
        # ...
```

If we want to make our method more functional, we should return an `Either` type from it, but Python does not have that. Either type contains one of two values, either a left value or a right value. The `Either` type can be defined as follows. (The `Optional` class was defined in the previous chapter).

Figure 5.116

```
from collections.abc import Callable
from typing import Any, TypeVar, Generic

from Optional import Optional

TLeft = TypeVar('TLeft')
TRight = TypeVar('TRight')
T = TypeVar('T')
U = TypeVar('U')


class PrivateConstructor(type):
    def __call__(
        cls: type[T],
        *args: tuple[Any, ...],
        **kwargs: dict[str, Any]
     ):
        raise TypeError('Constructor is private')

    def _create(
        cls: type[T],
        *args: tuple[Any, ...],
        **kwargs: dict[str, Any]
    ) -> T:
        return super().__call__(*args, **kwargs)


class Either(Generic[TLeft, TRight],
metaclass=PrivateConstructor):
    def __init__(
        self,
        maybe_left_value: Optional[TLeft],
        maybe_right_value: Optional[TRight]
    ):
        self.__maybe_left_value = maybe_left_value
        self.__maybe_right_value = maybe_right_value

    @classmethod
    def with_left(cls, value: TLeft) -> 'Either[TLeft, TRight]':
        return cls._create(Optional.of(value), Optional.empty())

    @classmethod
    def with_right(cls, value: TRight) -> 'Either[TLeft,
TRight]':
        return cls._create(Optional.empty(), Optional.of(value))

    def has_left_value(self) -> bool:
        return self.__maybe_left_value.is_present()

    def has_right_value(self) -> bool:
            return self.__maybe_right_value.is_present()
```

```
    def map_left(
        self,
        to_value: Callable[[TLeft], U]
    ) -> 'Either[U, TRight]':
        return Either._create(
            self.__maybe_left_value.map(to_value),
            self.__maybe_right_value
        )

    def map_right(
        self,
        to_value: Callable[[TRight], U]
    ) -> 'Either[TLeft, U]':
        return Either._create(
            self.__maybe_left_value,
            self.__maybe_right_value.map(to_value)
        )

    def map(
        self,
        left_to_value: Callable[[TLeft], U],
        right_to_value: Callable[[TRight], U]
    ) -> U:
        return
self.__maybe_left_value.map(left_to_value).or_else_get(
            lambda:
self.__maybe_right_value.map(right_to_value).try_get()
        )

    def apply(
        self,
        consume_left_value: Callable[[TLeft], None],
        consume_right_value: Callable[[TRight], None]
    ) -> None:
        self.__maybe_left_value.if_present(consume_left_value)
        self.__maybe_right_value.if_present(consume_right_value)
```

Below are some examples of how to use the `Either` class:

Figure 5.117

```
class Error(Exception):
    pass


int_or_error: Either[int, Error] = Either.with_left(3)
int_or_error2: Either[int, Error] = Either.with_right(Error())

print(int_or_error.has_left_value()) # Prints True
print(int_or_error2.has_right_value()) # Prints True
print(
    int_or_error.map_left(lambda number: number *
2).has_left_value()
)
# Prints True

print(int_or_error.map(lambda number: number * 2, lambda error:
0))
# Prints 6

print(int_or_error2.map(lambda number: number * 2, lambda error:
```

```
0))
# Prints 0
```

Now, we can use the new `Either` type and rewrite the example as follows:

Figure 5.118

```
from typing import Awaitable, TypeVar

from BackendError import BackendError
from Entity import Entity

T = TypeVar('T', bound=Entity)


class DataStore:
    async def create_entity(
        self,
        entity: T
    ) ->  Awaitable[Either[T, BackendError]]:
        # ...
```

5.9.1.4: Adapt to Wanted Error Handling Mechanism

You can adapt to a desired error-handling mechanism by creating an adapter class. For example, if a library has an error-raising method, you can create an adapter class with a method returning an optional value. The below `Url` class has a `try_create_url` factory method that can raise an error:

Figure 5.119

```
class Url:
    # ...

    class CreateUrlError(Exception):
        pass

    @classmethod
    def try_create_url(
        cls,
        scheme: str,
        host: str,
        port: int,
        path: str,
        query: str
    ) -> 'Url':
        # ...
        # Potentially raise a CreateUrlError here ...
```

We can create a `UrlFactory` adapter class with a non-error-raising method, `create_url`.

Figure 5.120

```
from Url import Url


class UrlFactory:
    def create_url(
        self,
        scheme: str,
        host: str,
        port: int,
        path: str,
        query: str
    ) -> Url | None:
        try:
            return Url.try_create_url(scheme, host, port, path,
query)
        except Url.CreateUrlError:
            return None
```

If the code using the `UrlFactory` is interested in the error details, we can also create a method that does not raise an error but returns either a value or an error:

Figure 5.121

```
from typing import Union

from Url import Url


class UrlFactory:
    def create_url_or_error(
        self,
        scheme: str,
        host: str,
        port: int,
        path: str,
        query: str
    ) -> Union[(Url, None), (None, Url.CreateUrlError)]:
        try:
            return (
                Url.try_create_url(scheme, host, port, path,
query),
                None,
            )
        except Url.CreateUrlError as error:
            return None, error
```

5.9.1.5: Functional Exception Handling

The below `Failable` class can be used in functional error handling. A `Failable` object represents either a value of type `T` or an instance of the `Exception` class, i.e., `Failable[T]` is the same as `Either[T, Exception]`

Figure 5.122

```python
from collections.abc import Callable
from typing import Any, Generic, TypeVar

from Either import Either

T = TypeVar('T')


class PrivateConstructor(type):
    def __call__(
        cls: type[T],
        *args: tuple[Any, ...],
        **kwargs: dict[str, Any]
     ):
        raise TypeError('Constructor is private')

    def _create(
        cls: type[T],
        *args: tuple[Any, ...],
        **kwargs: dict[str, Any]
    ) -> T:
        return super().__call__(*args, **kwargs)


TError = TypeVar('TError', bound=Exception)
U = TypeVar('U')


class Failable(Generic[T], metaclass=PrivateConstructor):
    def __init__(self, value_or_error: Either[T, Exception]):
        self.__value_or_error = value_or_error

    @classmethod
    def with_value(cls, value: T) -> 'Failable[T]':
        return cls._create(Either.with_left(value))

    @classmethod
    def with_error(cls, error: Exception) -> 'Failable[T]':
        return cls._create(Either.with_right(error))

    def or_raise(self, error_cls: type[TError]) -> T:
        return self.__value_or_error.map(
            lambda value: value,
            lambda error: self.__raise(error_cls(error))
        )

    def or_else(self, other_value: T) -> T:
        return self.__value_or_error.map(
            lambda value: value,
            lambda error: other_value
        )

    def map_value(
        self,
        to_value: Callable[[T], U]
    ) -> 'Failable[U]':
        return
Failable._create(self.__value_or_error.map_left(to_value))

    def map_error(
        self,
        to_error: Callable[[Exception], Exception]
```

```
    ) -> 'Failable[T]':
        if self.__value_or_error.has_left_value():
            error = to_error(Exception())
            return Failable.with_error(error)
        else:
            return Failable._create(
                self.__value_or_error.map_right(to_error)
            )

    def __raise(self, error: Exception) -> None:
            raise error
```

In the below example, the `read_config` method returns a `Failable[Configuration]`. The `try_initialize` method either obtains an instance of `Configuration` or raises an error of type `Application.InitializeError`.

Figure 5.123

```
from DataExporterError import DataExporterError


class Application:
    # ...

    class InitializeError(DataExporterError):
        pass

    def try_initialize(self) -> None:
        configuration = self.__config_reader \
            .read_config(...) \
            .or_raise(self.InitializeError)
```

The benefit of the above functional approach is that it is shorter than an entire try-catch block. The above functional approach is also as understandable as a try-catch block. Remember that one should write the shortest, most understandable code. When a method returns a `Failable` instance, you don't have to name the method with the *try* prefix because the method does not raise an error. The call to the `or_raise` method on `Failable` converts the functional code back to imperative.

You can also use other methods of the `Failable` class. For example, a default value can be returned with the `or_else` method:

Figure 5.124

```
from DefaultConfig import DefaultConfig


class Application:
```

```
    # ...

    def initialize(self) -> None:
        configuration =
self.__config_reader.read_config(...).or_else(
            DefaultConfig()
        )
```

You can also transform multiple imperative error-raising statements into functional, failable statements. For example, instead of writing:

Figure 5.125

```
from DataExporterError import DataExporterError


class Application:
    # ...

    class InitializeError(DataExporterError):
        pass

    def try_initialize(self) -> None:
        try:
            config_json = self.__data_fetcher.try_fetch_data(
                self.__config_url
            )

            configuration =
self.__config_parser.try_parse(config_json)
        except DataExporterError as error:
            raise self.InitializeError(error)
```

You can write:

Figure 5.126

```
from DataExporterError import DataExporterError


class Application:
    # ...

    class InitializeError(DataExporterError):
        pass

    def try_initialize(self) -> None:
        configuration = (
           self.__data_fetcher.fetch_data(self.__config_url)
           .map_value(self.__config_parser.parse)
           .or_raise(self.InitializeError)
       )
```

The above functional code is shorter than the same imperative code, but it is slightly less readable, for which reason you

might want to use the imperative approach instead of the functional approach.

It can be error-prone to use error-raising imperative code together with functional programming constructs. For example, let's assume we have the below code that reads and parses multiple configuration files to a single configuration object using a functional programming construct `reduce`. We have named the config reading function `try_read_config` with the try-prefix because it can raise an error. When we use the `reduce` function, we must remember to surround it with a try-except block because the `reduce` function will call the `try_read_config` function that can throw.

Figure 5.127

```
import json
from functools import reduce
from typing import Any


def try_read_config(
    accumulated_config: dict[str, Any],
    config_file_path_name: str
):
    with open(config_file_path_name) as config_file:
        config_json = config_file.read()

    config = json.loads(config_json)
    return accumulated_config | config


def get_config(config_file_path_names: list[str]) -> dict[str,
Any]:
    try:
        return reduce(try_read_config, config_file_path_names,
{})
    except:
        # ...
```

We could turn the above example more functional by making the `get_config` function return a `Failable` instance:

Figure 5.128

```
import json
from functools import reduce
from typing import Any

from Failable import Failable


def to_config_or_error(
    accum_config_or_error: Failable[dict[str, Any]],
    config_file_path_name: str
```

```
) -> Failable[dict[str, Any]]:
    try:
       with open(config_file_path_name) as config_file:
               config_json = config_file.read()

       config = json.loads(config_json)

       return accum_config_or_error.map_value(
            lambda accum_config: accum_config | config
        )
    except (OSError, json.JSONDecodeError) as error:
        return accum_config_or_error.map_error(
            lambda accum_error: RuntimeError(
                f'{str(accum_error)}\n{config_file_path_name}:
{str(error)}'
            )
        )


def get_config(
    config_file_path_names: list[str]
) -> Failable[dict[str, Any]]:
    return reduce(
        to_config_or_error,
        config_file_path_names,
        Failable.with_value({})
    )
```

If we have a *config1.json* file with the following contents:

Figure 5.129

```
{
    "foo": 1,
    "bar": 2
}
```

and we have a *config2.json* file with the following contents:

Figure 5.130

```
{
    "xyz": 3
}
```

Then, we can run the following code:

Figure 5.131

```
config_file_path_names = ['config1.json', 'config2.json']
maybeConfig = get_config(config_file_path_names)
print(maybeConfig.or_raise(RuntimeError))
# Prints {'foo': 1, 'bar': 2, 'xyz': 3}
```

Let's introduce an error (a missing comma after the first property) in the *config1.json* file:

Figure 5.132

```
{
    "foo": 1
    "bar": 2
}
```

Let's also try to provide a non-existing configuration file *config3.json*:

Figure 5.133

```
config_file_path_names = ['config1.json', 'config3.json']
maybeConfig = get_config(config_file_path_names)
print(maybeConfig.or_raise(RuntimeError))
# Raises a RuntimeError with the following message:
# config1.json: Expecting ',' delimiter: line 3 column 3 (char
16)
# config3.json: [Errno 2] No such file or directory:
'config3.json'
```

5.10: Avoid Off-By-One Errors Principle

Off-by-one errors usually result from the fact that collections in programming languages are indexed with zero-based indexes. Zero-based indexing is unnatural for human beings but excellent for computers. However, programming languages should be designed with humans in mind. People never speak about getting the zeroth value of an array. We speak of getting the first value in the array. As the null value was called a billion-dollar mistake, I would call the zero-based indexing another billion-dollar mistake. Let's hope that someday we get a programming language with one-based indexing! But then we must unlearn the zero-based indexing habit… And that's another problem!

In some languages, you can create for-loops with a loop counter. Below are two examples of programming errors in JavaScript that are easy to make if you are not careful enough:

Figure 5.134

```
for (let index = 0; index <= values.length; index++) {
  // ...
}

for (let index = 0; index < values.length - 1; index++) {
  // ...
}
```

In the first example, there should be '<' instead of '<=', and in the latter example, there should be '<=' instead of '<'. Fortunately, the above mistakes can be avoided in Python:

Figure 5.135

```
for value in values:
  # ...
```

In Python's `range` function, you must remember it starts from zero, and the end of the range is exclusive. You will have an off-by-one error if you assume the start to be one or the end of the range is inclusive. The off-by-one errors are caused by the fact that, given a range, people assume it is inclusive at both ends by default. So, a `range(6)` gives values from 0 to 5, not from 1 to 6. And `range(1, 6)` gives values from 1 to 5, not from 1 to 6. The same thing is with slices, e.g. `values[:6]` starts from index 0 and ends at index 5. If you want a slice that is all but the last item, you can use negative indexing: `values[:-1]` gives values except the last one. Using -1 is much safer than using `values[:len(values) - 1]`, which might produce an off-by-one error if you forget the -1. And similarly using `values[:-2]` is less error-prone than using `values[:len(values) - 2]`. You can also use negative indexing, e.g., to get the last value with `values[-1]` instead of `values[len(values) - 1]`. A negative index is a one-based index starting from the end of a list.

Additionally, unit tests are your friend when trying to spot off-by-one errors. So remember to write unit tests for the edge cases, too.

5.11: Be Critical When Googling or Using Generative AI Principle

> ***You should always analyze code taken from the web to ensure it meets the criteria for production code. Don't let the AI be the master, but an apprentice.***

We all have done it, and we have done it hundreds of times: googled for answers. Usually, you find good resources by

googling, but the problem often is that examples in the Google results are not necessarily production quality. One specific thing missing in them is error handling. If you copy and paste code from a website, it is possible that errors are not handled appropriately. You should always analyze the copy-pasted code to see if error handling needs to be added.

When you provide answers for other people, try to make the code as production-like as possible. In Stack Overflow, you find the most up-voted answer right below the question. If the answer is missing error handling, you can comment on that and let the author improve their answer. You can also up-vote an answer that seems the most production-ready. The most up-voted answers tend to be pretty old. For this reason, it is useful to scroll down to see if a more modern solution fits your needs better. You can also up-vote that more modern solution to make it eventually rank higher in the list of answers.

Regarding open-source libraries, the first examples in their documentation can describe only the “happy path” usage scenario, and error handling is described only in later parts of the documentation. This can cause problems if you copy-paste code from the “happy path” example and forget to add error handling. For this reason, open-source library authors should give production-quality examples early in the documentation.

Regarding generative AI, e.g., ChatGPT, I have a couple of experiences. I asked ChatGPT to generate simple Django code. The generated code was about 95% correct, but it did not work. The problem was that ChatGPT forgot to provide code for generating the database tables (makemigrations, migrate). If you are inexperienced with the Django framework, that problem might be challenging to solve. In that case, continue the discussion with ChatGPT and ask it to solve the problem for you.

My other experiment with ChatGPT was to generate GraphQL server code using the *Ariadne* library. The ChatGPT-generated code was for an old version of Ariadne and did not work correctly with a newer version of the Ariadne library. (Notice that the data used to train ChatGPT contains more older than newer data. ChatGPT could not prioritize the less and newer

data over the older and more data.) It also generated some lines of code in the wrong order, which made the GraphQL API not work at all. It took quite a lot of debugging for such a small program to find out what was wrong: The executable schema was created before the query resolver. It should have been created only after defining the resolver.

You should familiarize yourself with the AI-generated code when using ChatGPT or other generative AI tools. Otherwise, you don't know what your program is doing, and if the AI-generated code contains bug(s), those will be hard to find because you don't clearly understand what the code is actually doing. Don't let the AI be the master, but an apprentice.

The best way to prevent bugs related to code taken from the web is to practice *test-driven development* (TDD). TDD is better described in the next chapter. The idea behind TDD is to specify the function first and write unit test cases for different scenarios: edge/corner cases, error scenarios, and security scenarios. For example, let's say you are new to Python and google for a code snippet to perform an HTTP request to an API endpoint. You copy-paste the code into your function. Now, error scenarios are not handled. What you should do is practice TDD and write unit test cases for different scenarios, like, what if the remote server cannot be contacted or the contact results in timeout, or what if the remote server responds with an error (an HTTP response with a status code greater than or equal to 400). What if you need to parse the result from the API (e.g., parse JSON), and it fails? Once you have written a unit test case for all those scenarios, you can be sure that error handling in the function implementation is not forgotten.

5.12: Make One Change At A Time Principle

> ***Don't try to make multiple distinct changes at the same time.***

If you try to make multiple unrelated changes simultaneously, you are focusing on too many things, and you are more likely to introduce a bug or bugs. Don't try to implement two distinct features at the same time. Don't try to make two distinct refactorings at the same time. Don't try to implement a new feature and do refactoring simultaneously. Try to make a single change as small and isolated as possible. I have violated this principle so many times. I have thought, okay, this is a small change I can do together with this other change. But later, I realized that the changes were not so small after all and that I had created some bugs. I didn't know which of the many changes caused the bug. That made the bug hunting more difficult than it should have been. So, resist the urge to make multiple distinct changes simultaneously. Have all changes that need to be made as separate user stories in the team backlog or list small changes to be done in a TODO.MD file so that they are not forgotten.

If you need to implement a new feature, analyze if you should refactor the code first to make the feature implementation easier. If the refactoring is not necessary for the feature and the feature is important, you can implement the feature first and refactor the code later. If you gain benefit by refactoring first and the feature is not time-critical, refactor first and only after that implement the feature.

5.13: Use Appropriate Data Structure Principle

Next, the main built-in Python data structures with their use cases will be presented.

5.13.1: List

> *Use a list when you need an ordered collection or should be able to store the same item multiple times.*

The implementation of a Python list is similar to C++'s `std::vector` or the `ArrayList` class in Java. Lists are mutable

and can contain duplicate elements. Each element can be accessed by its index in a performant way. If you need to add or remove elements at the beginning of the list, that is not a cheap operation because all existing elements in the list must be moved by one index to the right or left. Similarly, if you add an element to the end of a list and the allocated space for the list is full, new space for the list must be allocated, and the list elements must be copied to the newly allocated space.

5.13.2: Dictionary

> *Use a dictionary when you need quick access to a value by a key.*

Python dictionary is similar to a `HashMap` in Java. Accessing a value by its key is always a cheap operation. You can iterate over a dictionary using one of the following methods: `keys()`, `values()`, or `items()`. Combining two dictionaries is easy with the unpacking operator (`{**dict1, **dict2}`) or with the merge operator (`dict1 | dict2`).

5.13.3: Tuple

> *Use a tuple instead of a list when you need an immutable ordered collection.*

In a tuple, each element can be accessed by its index in a performant way. Like a list, a tuple can contain various type of elements and the same element multiple times.

5.13.4: Set

> *Use a set when you don't need an ordered collection, and duplicate elements are not allowed.*

Accessing a set by a value is cheap because Python sets are implemented as hash tables. On the contrary, if you have a list, the whole list may need to be gone through to find a specific

value. If you have a list and want to remove duplicates from it, you can convert the list into a set: `set(my_list)`. Python also offers frozen sets with the `frozenset` constructor. These sets are immutable.

5.13.5: String

Use a string to store an immutable ordered collection of characters.

In Python, strings are lists of Unicode characters. Strings are always immutable. Accessing a character by its index is always a cheap operation O(1).

5.13.6: Bytearray

Use a byte array to store an ordered collection of values between 0 and 256.

A byte array is created with the `bytearray` constructor. Accessing a byte by its index is always a cheap operation O(1). Here are two examples of creating a byte array: `bytearray((255, 50, 20)` or `bytearray(b'\xFF\x32\x14')`.

5.13.7: Counter

Use a counter to store counts for various elements.

A counter is a dictionary where elements are stored as keys, and their counts are stored as values. Counter objects differ from traditional dictionary objects when a value for a missing key is requested, value zero is returned instead of a `KeyError`.

5.13.8: OrderedDict

Use the `OrderedDict` class if the dictionary needs to remember the insertion order of entries.

When you iterate over an `OrderedDict` object, you get the entries in the insertion order.

5.13.9: Deque (Double Ended Queue)

Use a deque when you need cheap insertion and removal of elements at either or both ends of the collection.

Adding and removing elements at the ends of a deque are cheap (O(1)) operations because the deque is implemented as a doubly linked list. The drawback of the deque is that randomly accessing an element at a specific index is slow (O(n)).

5.13.10: Customized Built-In Collections

If you need to customize the behavior of a built-in list, string, or dictionary, make your custom class extend one of the following classes: `UserList`, `UserString`, or `UserDict`.

5.13.11: Stack (LIFO Queue)

A stack can be implemented using a deque.

You can push an element into the stack with the deque's `append` method and pop an element out of the stack with the deque's `pop` method. You can also implement a stack using a list, but operations can be slower sometimes, especially if the stack is large. When you implement a stack using a list, you use the `append` and `pop` methods like with a deque.

5.13.12: Queue (FIFO Queue)

A queue can be implemented using a deque.

You can add an element to a queue with the deque's `append` method and pop an element from the queue with the deque's `popleft` method. You can also implement a queue using a list, but operations can be slower sometimes, especially if the queue is large. When you implement a queue using a list, you use the `append` and `pop(0)` methods.

5.13.13: Priority Queue

> *Use a priority queue to retrieve elements based on their priority.*

You can implement a priority queue using the `PriorityQueue` class from the `queue` module. That class is also thread-safe.

5.13.14: Synchronized Queues (LIFO/FIFO)

- Use a synchronized queue for implementing thread-safe multi-consumer, multi-producer queues.*

Python provides two thread-safe queue implementations in the `queue` module: `Queue` and `LifoQueue`.

5.14: Optimization Principle

Code optimization makes code run faster and/or consume less memory. Faster code improves the end-user experience, and optimization reduces the need for computing resources (CPU/memory), making operating the software cheaper.

> ***Avoid premature optimization. Premature optimization may hinder crafting a proper object-oriented design for a software component.***

Measure unoptimized performance first. Then, decide if optimization is needed. Implement optimizations individually and measure the performance after each optimization to

determine if the particular optimization matters. You can then utilize the knowledge you gained in future projects only to make optimizations that boost performance significantly enough. Sometimes, you can make performance optimization in the early phase of a project if you know that a particular optimization is needed (e.g., from previous experience), and the optimization can be implemented without negatively affecting the object-oriented design.

5.14.1: Optimization Patterns

The following optimization patterns are described in this section:

- Optimize busy loops only pattern
- Remove unnecessary functionality pattern
- Object pool pattern
- Use optimal data structures pattern
- Algorithm complexity reduction pattern
- Cache function results pattern
- Buffer file I/O pattern
- Share identical objects, a.k.a flyweight pattern

5.14.1.1: Optimize Busy Loops Only Pattern

Optimizations should primarily target only the busy loop or loops in a software component. Busy loops are the loops in threads that execute over and over again, possibly thousands or more iterations in a second. Performance optimization should not target functionality that executes only once or a couple of times during the software component's lifetime, and running that functionality does not take a long time. For example, an application can execute configuration reading and parsing functionality when it starts. This functionality takes a short time to execute. It is not reasonable to optimize that functionality because it runs only once. It does not matter if

you can read and parse the configuration in 200 or 300 milliseconds, even if there is a 50% difference in performance.

Let's use the data exporter microservice as an example. Our data exporter microservice consists of input, transformer, and output parts. The input part reads messages from a data source. We cannot affect the message reading part if we use a 3rd party library for that purpose. Of course, if multiple 3rd party libraries are available, it is possible to craft performance tests and evaluate which library offers the best performance. If several 3rd party libraries are available for the same functionality, we tend to use the most popular library or a library we know beforehand. If performance is an issue, we should evaluate different libraries and compare their performances.

The data exporter microservice has the following functionality in its busy loop: decode an input message to an internal message, perform transformations, and encode an output message. Decoding an input message requires decoding each field in the message. Let's say there are 10000 messages handled per second, each with 100 fields. During one second, 100,000 fields must be decoded. This reveals that the optimization of the decoding functionality is crucial. The same applies to output message encoding. We at Nokia have implemented the decoding and encoding Avro binary fields ourselves. We were able to make them faster than what was provided by a 3rd party library.

5.14.1.2: Remove Unnecessary Functionality Pattern

Removing unnecessary functionality is something that will boost performance. Every now and then, you should stop and think critically about your software component: Is my software component doing only the necessary things considering all circumstances?

Let's consider the data exporter's functionality in a special case. It is currently decoding an input message to an internal message. This internal message is used when making various transformations to the data. Transformed data is encoded to a

wanted output format. The contents of the final output message can be a small subset of the original input message. This means that only a tiny part of the decoded message is used. In that case, it is unnecessary to decode all the fields of an input message if, for example, only 10% of the fields are used in the transformations and output messages. By removing unnecessary decoding, we can improve the performance of the data exporter microservice.

5.14.1.3: Object Pool Pattern

In garbage-collected languages like Python, the benefit of using an object pool is evident from the garbage-collection point of view. Objects are created only once in the object pool pattern and then reused. This will take pressure away from garbage collection. If we didn't use an object pool, new objects could be created in a busy loop repeatedly, and soon after they were created, they could be discarded. This would cause many objects to be made available for garbage collection in a short time. Garbage collection takes processor time, and if the garbage collector has a lot of garbage to collect, it can slow the application down for an unknown duration at unknown intervals.

5.14.1.4: Use Optimal Data Structures Pattern

If you are performing number crunching in your application, do not use the regular Python data structures, but find a suitable library, like numpy, that contains optimized data structures for a particular use case.

5.14.1.5: Algorithm Complexity Reduction Pattern

Choose an algorithm with reduced complexity as measured using the Big-O notation. This usually boosts the performance. In the below example, we are using the find algorithm with a list:

Figure 5.136

```
values = [1, 2, 3, 4, 5, ..., 2000]
if 2000 in values:
    print("Value 2000 found")
```

The above algorithm must traverse the list, which makes it slower compared to the find algorithm with a set:

Figure 5.137

```
values = {1, 2, 3, 4, 5, ..., 2000}
if 2000 in values:
    print("Value 2000 found")
```

The below algorithm (list comprehension) will generate a list of 20,000 values:

Figure 5.138

```
values = [value for value in range(20_000)]
```

If we don't need all the 20,000 values in the memory at the same time, we could use a different algorithm (generator expression) which consumes much less memory, because not all the 20,000 values are in the memory:

Figure 5.139

```
values = (value for value in range(20_000))
```

The type of the `values` object in the above example is `Generator`, which inherits from `Iterator`. You can use the `values` anywhere an iterator is expected.

5.14.1.6: Cache Function Results Pattern

You can benefit from caching the function results if you have an expensive pure function that always returns the same result for the same input without any side effects. You can cache function results using the `@cache` or `@lru_cache` decorator. Here is an example:

Figure 5.140

```
from functools import lru_cache

# Results of 500 most recent calls to the function
# will be cached
@lru_cache(maxsize=500)
def make_expensive_calc(value: int):
    # ...
```

```
print(make_expensive_calc(1))
# After the first call,
# the function result for the input value 1
# will be cached

print(make_expensive_calc(1))
# The result of function call is fetched from the cache
```

`@cache` is the same as `@lru_cache(maxsize=None)`, i.e., the cache does not have a maximum size limit.

5.14.1.7: Buffer File I/O Pattern

You can benefit from setting custom buffer sizes if you are reading/writing large files. The below examples set buffer sizes to 1MB:

Figure 5.141

```
with open('data.json', 'r', buffering=1_048_576) as data_file:
    data = data_file.read()

with open('data.json', 'w', buffering=1_048_576) as data_file:
    data_file.write(data)
```

5.14.1.8: Share Identical Objects a.k.a Flyweight Pattern

If your application has many objects with some identical properties, those parts of the objects with identical properties are wasting memory. You should extract the common properties to a new class and make the original objects reference a shared object of that new class. Now, your objects share a single common object, and possibly significantly less memory is consumed. This design pattern is called the *flyweight pattern* and was described in more detail in the earlier chapter.

6: Testing Principles

Testing is traditionally divided into two categories: functional and non-functional testing. This chapter will first describe the functional testing principles and then the non-functional testing principles.

6.1: Functional Testing Principles

Functional testing is divided into three phases:

- Unit testing
- Integration testing
- End-to-end (E2E) testing

Functional test phases can be described with the testing pyramid:

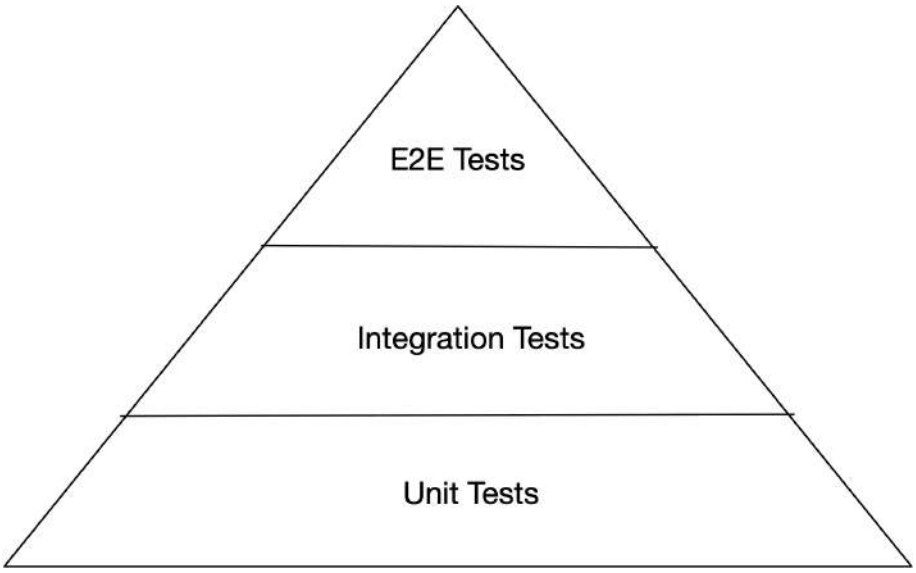

Figure 6.1. Testing Pyramid

The testing pyramid depicts the relative number of tests in each phase. Most tests are unit tests. The second most tests are integration tests; the fewest are E2E tests. Unit tests should cover the whole codebase of a software component. Unit

testing focuses on testing individual public functions as units (of code). Software component integration tests cover the integration of the unit-tested functions to a complete working software component, including testing the interfaces to external services. External services include a database, a message broker, and other microservices. E2E testing focuses on testing the end-to-end functionality of a complete software system.

There are various other terms used to describe different testing phases:

- Module testing (an older term for unit testing)
- (Software) Component testing (same as integration testing)
- System (Integration) testing (same as E2E testing)

The term *component testing* is also used to denote only the integration of the unit-tested modules in a software component without testing the external interfaces. In connection with the *component testing* term, there is the term *integration testing* used to denote the testing of external interfaces of a software component. Here, I use the term *integration testing* to denote both the integration of unit-tested modules and external interfaces.

6.1.1: Unit Testing Principle

> ***Unit tests should test the functionality of public functions as isolated units with as high coverage as possible. The isolation means that dependencies (other classes/modules/services) are mocked.***

Unit tests should be written for public functions only. Do not try to test private functions separately. They should be tested indirectly when testing public functions. Unit tests should test the function specification, i.e., what the function is expected to do in various scenarios, not how the function is implemented. When you unit test only public functions, you can easily

refactor the function implementation, e.g., rewrite the private functions that the public function uses without modifying the related unit tests (or with minimal changes).

Below is an example of a public function using a private function:

Figure 6.2

```
from other_module import do_something

def __read_file(...):
# ...

def parse_config(...):
# ...
# __read_file(...)
# do_something(...)
# ...
```

In the above *parse_config.py* module, there is one public function, `parse_config`, and one private function, `__read_file`. In unit testing, you should test the public `parse_config` function in isolation and mock the `do_something` function, which is imported from another module. And you indirectly test the private `__read_file` function when testing the public `parse_config` function.

Below is the same example rewritten using classes. You test the class-based version in a similar way as the above version. You write unit tests for the public `parse_config` method only. Those tests will test the private `__read_file` method indirectly. You must supply a mock instance of the `OtherClass` class for the `ConfigParser` constructor.

Figure 6.3

```
class OtherClass:
    # ...

    def do_something(self, ...) -> None:
        # ...


class ConfigParser:
    def __init__(self, other_class: OtherClass):
        self.__other_class = other_class

    # ...

    def parse_config(self, ...):
        # ...
        # self.__read_file(...)
```

```
        # self.__other_class.do_something(...)
        # ...

    def __read_file(self, ...):
        # ...
```

If you have a public function using many private methods, testing the public method can become complicated, and the test method becomes long, possibly with many expectations on mocks. It can be challenging to remember to test every scenario that exists in the public and related private methods. What you should do is refactor all or some private methods into public methods of one or more new classes (This is the *extract class* refactoring technique explained in the previous chapter). Then, the test method in the original class becomes shorter and more straightforward, with less expectation of mocks. This is an essential refactoring step that should not be forgotten. It helps keep both the source code and unit tests readable and well-organized. The unit test code must be the same high quality as the source. Unit test code should use type annotations and the same linter rules as the source code itself. The unit test code should not contain duplicate code. A unit test method should be a maximum of 5-9 statements long. Aim for a single assertion or put assertions in one well-named private method. If you have more than 5-6 expectations on mocks, you need to refactor the source code as described above to reduce the number of expectations and to make the test method shorter.

Unit tests should test all the functionality of a public function: happy path(s), possible failure situations, security issues, and edge/corner cases so that each code line of the function is covered by at least one unit test. Security issues in functions are mostly related to the input the function gets. Is that input secure? If your function receives unvalidated input data from an end-user, that data must be validated against a possible attack by a malicious end-user.

Below are some examples of edge/corner test cases listed:

- Is the last loop counter value correct? This test should detect possible off-by-one errors

- Test with an empty array
- Test with the smallest allowed value
- Test with the biggest allowed value
- Test with a negative value
- Test with a zero value
- Test with a very long string
- Test with an empty string
- Test with floating-point values having different precisions
- Test with floating-point values that are rounded differently
- Test with an extremely small floating-point value
- Test with an extremely large floating-point value

Unit tests should not test the functionality of dependencies. That is something to be tested with integration tests. A unit test should test a function in isolation. If a function has one or more dependencies on other functions defined in different classes (or modules), those dependencies should be mocked. A *mock* is something that mimics the behavior of a real object or function. Mocking will be described in more detail later in this section.

Testing functions in isolation has two benefits. It makes tests faster. This is a real benefit because you can have a lot of unit tests, and you run them often, so the execution time of the unit tests must be as short as possible. Another benefit is that you don’t need to set up external dependencies, like a database, a message broker, and other microservices, because you are mocking the functionality of the dependencies.

Unit tests give you protection against introducing accidental bugs when refactoring code. Unit tests ensure that the implementation code meets the function specification. It should be remembered that it is hard to write the perfect code on the first try. You are bound to practice refactoring to keep your code base clean and free of technical debt. And when you

refactor, the unit tests are on your side to prevent accidentally introducing bugs.

6.1.1.1: Test-Driven Development (TDD)

Test-driven development (TDD) is a software development process in which software requirements are formulated as (unit) test cases before the software is implemented. This is as opposed to the practice where software is implemented first, and test cases are written only after that.

The benefits of TDD are many:

- Lesser likelihood of forgetting to implement some failure scenarios or edge cases
- No big upfront design, but emergent (and usually better) design because of constant refactoring
- Existing tests make the developer feel confident when refactoring
- Less cognitive load for the developer because only one scenario is implemented at a time
- You automatically write unit-testable code
- You don't write unnecessary code because you only write code that makes the tests pass (you can follow the YAGNI principle)
- Less need for debugging

I have been in the industry for almost 30 years, and when I began coding, there were no automated tests or test-driven development. Only since around 2010 have I been writing automated unit tests. Due to this background, TDD has been quite difficult for me because there is something I have grown accustomed to: Implement the software first and then do the testing. I assume many of you have also learned it like that, which can make switching to TDD rather difficult. Little material exists that teaches topics using TDD. The internet is full of books, courses, videos, blogs, and other posts that don't teach you the proper way of development: TDD. The same

applies to this book, also. I present code samples in the book but don't always use TDD because it would make everything complicated and more verbose. We are constantly taught to program test-last! When asking people who have conducted TDD, they say the main benefit is that it reduces stress because you don't have to achieve multiple goals simultaneously, like designing the function and thinking of and implementing (and remembering to implement) all possible execution paths.

TDD can feel unnatural at the beginning. It can slow you down at first. You just have to keep practicing it systematically. In that way, you can start gradually seeing the benefits and build a habit of always using TDD. Later, you will notice that it begins to feel natural, and it does not slow you down anymore but brings you the many benefits depicted above.

The complete TDD cycle, as instructed by *Kent Beck* in his book *Test-Driven Development by Example* consists of the following steps:

1. Add a test for a specified functionality
2. Run all the tests (The just added test should fail if the functionality it is testing is not implemented yet)
3. Write the simplest possible code that makes the tests pass
4. Run all the tests. (They should pass now)
5. Refactor as needed (Existing tests should ensure that anything won't break)
6. Start again from the first step until all functionality is implemented, refactored, and tested

Instead of TDD, you should use low-level behavior-driven development (BDD). We discuss BDD later in this chapter when addressing integration testing. BDD extends TDD by more formally defining the low-level (function-level) behavior. To use BDD, ensure the following in your unit tests:

- Test method name tells the tested function name and the scenario that is tested in that particular test method
- The test method body is organized into three sections: Given, When, and Then steps. This is basically the same thing as Arrange-Act-Assert, which is the suggested way to structure unit tests. In the Given/Arrange phase, you prepare everything for the test. In the Act/When phase, you perform an operation; in the Assert/Then phase, you make assertions about the operation result. Try to keep the assertion phase as simple as possible. The best is to have only a single assertion, but that is not always possible. You can also extract multiple assertions into a well-named private method and call that method from the test method in the assertion phase.

Some TDD practitioners suggest you name a test method after the feature it is testing, including a description of the scenario and expected outcome. That approach can easily make the test method names too long and hard to read. You can always see the expected result by looking at the end of the test method, which should preferably contain only a single assertion. I also like to put test methods in a test class in the same order I have in the tested class. Also, the scenarios can be ordered from a more specialized scenario to a more generalized scenario (or vice versa). All this makes navigating between test methods a breeze. Also, there is not much difference between a class's features and its public methods. Each public method should implement a single feature only (remember the *single responsibility principle*). A single feature (method) consists of one or more scenarios. For example, a `Stack` has four features: You can `push` an item to the stack and `pop` an item from the stack. You can also ask if the stack `is_empty` or ask its `size`. These features have multiple scenarios: for example, how they behave if the stack is empty versus the stack with items in it. When you add features to a class, you should put them into new methods or, preferably, use the *open-closed principle* and put them in a totally new class.

Below are two examples of test method names in a `StackTests` class. The first one contains the method name that

is tested, and the second one is named after the feature:

Figure 6.4

```
class StackTests(unittest.TestCase):
    def test_pop__when_stack_is_empty(self):
        # ...

        # WHEN + THEN
        # self.assertRaises(...)

    # You can also use method name prefixed with 'it':
    # it_should_raise_error_when_popping_item_from_empty_stack
    def should_raise_error_when_popping_item_from_empty_stack
        # ...

        # WHEN + THEN
        # self.assertRaises(...)
```

You can use the feature-based naming instead of the method-based naming convention if you want. I use the method-based naming convention in all unit tests in this book.

Let's continue with an example. Suppose there is the following user story in the backlog waiting to be implemented:

> *Parse configuration properties from a configuration string to a configuration object. Configuration properties can be accessed from the configuration object. If parsing the configuration fails, an error should be produced.*

Let's first write a test for the simplest 'happy path' scenario of the specified functionality: parsing a single property only.

Figure 6.5

```
import unittest

from ConfigParserImpl import ConfigParserImpl


class ConfigParserTests(unittest.TestCase):
    config_parser = ConfigParserImpl()

    def test_parse__when_successful(self):
        # GIVEN
        config_str = 'propName1=value1'

        # WHEN
        config = self.config_parser.parse(config_str)

        # THEN
        self.assertEqual(config.get_property_value('propName1'),
'value1')
```

If we run the test, we get a compilation error, meaning the test case we wrote won't pass. Next, we shall write the simplest possible code to make the test case both compile and pass. We can make shortcuts like using a fixed value (constant) instead of a more generalized solution.

Figure 6.6

```
from typing import Protocol, Final


class Configuration(Protocol):
    def get_property_value(self, property_name: str) -> str:
        pass


class ConfigurationImpl(Configuration):
    def get_property_value(self, property_name: str) -> str |
None:
        return 'value1'


class ConfigParser(Protocol):
    def parse(self, config_str: str) -> Configuration:
        pass


class ConfigParserImpl(ConfigParser):
    def parse(self, config_str: str) -> Configuration:
        return ConfigurationImpl()
```

Let's write a test for a 'happy path' scenario where we have two properties. This forces us to make the implementation more generalized. We cannot use a constant anymore, and we should not use two constants with an if/else-statement because if we want to parse more than two properties, the approach using constants does not scale.

Figure 6.7

```
import unittest

from ConfigParserImpl import ConfigParserImpl


class ConfigParserTests(unittest.TestCase):
    config_parser = ConfigParserImpl()

    def test_parse__when_successful(self):
        # GIVEN
        config_str = 'propName1=value1\npropName2=value2'

        # WHEN
        config = self.config_parser.parse(config_str)

        # THEN
```

```
        self.assertEqual(config.get_property_value('propName1'),
'value1')
        self.assertEqual(config.get_property_value('propName2'),
'value2')
```

If we run all the tests, the new test will fail in the second assertion. Next, we shall write code to make the test cases pass:

Figure 6.8

```
from typing import  Final

class ConfigurationImpl(Configuration):
    def __init__(self, prop_name_to_value: dict[str, str]):
        self.__prop_name_to_value: Final = prop_name_to_value

    def get_property_value(self, property_name: str) -> str |
None:
        return self.__prop_name_to_value.get(property_name);


class ConfigParserImpl(ConfigParser):
    def parse(self, config_str: str) -> Configuration:
        # Parse config_str and assign properties to
        # 'prop_name_to_value' variable

        return ConfigurationImpl(prop_name_to_value)
```

Now, the tests pass, and we can add new functionality. Let's add a test for the case when parsing fails. We can now repeat the TDD cycle from the beginning by creating a failing test first:

Figure 6.9

```
import unittest

from ConfigParser import ConfigParser


class ConfigParserTests(unittest.TestCase):
    # ...

    def test_try_parse__when_parsing_fails(self):
        # GIVEN
        config_str = 'invalid'

        try:
            # WHEN
            self.config_parser.try_parse(config_str)

            # THEN
            self.fail('ConfigParser.ParseError should have been
raised')
        except ConfigParser.ParseError:
            # THEN error was successfully raised
```

Next, we should refactor the implementation to make the second test pass:

Figure 6.10

```
from typing import Protocol

from Configuration import Configuration
from DataExporterError import DataExporterError


class ConfigParser(Protocol):
    class ParseError(DataExporterError):
        pass

    def try_parse(self, config_str: str) -> Configuration:
        pass


class ConfigParserImpl(ConfigParser):
    def try_parse(self, config_str: str) -> Configuration:
        # Try parse config_str and if successful
        # assign config properties to 'prop_name_to_value'
        # variable

        if prop_name_to_value is None:
            raise self.ParseError()
        else:
            return ConfigurationImpl(prop_name_to_value)
```

We must also refactor the other unit tests to call `try_parse` instead of `parse`. We could continue the TDD cycle by adding new test cases for additional functionality if such existed.

Before starting the TDD process, list all the requirements (scenarios) with bullet points so you don't forget any. The scenarios should cover all happy paths, edge cases, and failure/security scenarios. For example:

- Edge case scenarios
    - Scenario A
    - Scenario B
- Failure scenarios
    - Scenario C
    - Scenario D
- Security scenarios
    - Scenario E

- Success scenarios
    - Scenario F

Listing all scenarios is an important step in order not to forget to test something because if you don't write a test for something, it's highly likely you won't implement it either. During the TDD process, you should add any missing scenarios to that list. Order the list so that the most specialized scenarios are listed first. Then, start the TDD process by following the ordered list. The most specialized scenarios are typically the easiest to implement, and this is why you should start with them. Specialized scenarios include edge cases and failure scenarios.

In the simplest case, a specialized scenario can be implemented, for example, by returning a constant from a function. An example of a specialized scenario with a `List` class's `is_empty` method is when the list is empty after creating a new `List` object. Test that first, and only after that test a scenario where something is added to the list, and it is no longer empty.

At the end of the list are the generalized scenarios that should make the function work with any input. To summarize, during the TDD process, you work from a specialized implementation towards a more generalized implementation.

The drawback of TDD is that you cannot always be 100% sure if you have a generalized enough implementation. You can only be 100% sure that your implementation works with the input used in the tests, but you cannot be sure if any input works correctly. To ensure that, you would have to test all possible input values (like all integer values), which is typically unreasonable.

Let's have another TDD example with a function that has edge cases. We should implement a `contains` method for a string class (we assume here that the `in` operator does not exists). The method should do the following:

> *The method takes a string argument, and if that string is found in the string the string object represents, then* `True`

is returned. Otherwise, `False` *is returned.*

There are several scenarios we might want to test to make sure that the function works correctly in every case:

- Edge cases
    1. Strings are equal
    2. Both strings are empty
    3. Argument string is empty
    4. The string under test is empty
    5. Argument string is found at the beginning of the other string
    6. Argument string is found at the end of the other string
    7. Argument string is longer than the other string
- Happy paths 8) Argument string is found in the middle of the other string 9) Argument string is not found in the other string

Let's start with the first scenario:

Figure 6.11

```
import unittest


class StringTests(unittest.TestCase):
    def test_contains__strings_are_equal(self):
        # GIVEN
        string = MyString('String')
        another_string = 'String'

        # WHEN
        string_contains_another_string =
string.contains(another_string)

        # THEN
        self.assertTrue(string_contains_another_string)
```

Next, we do as much implementation as is needed to make the above test pass. The simplest thing to do is to return a constant:

Figure 6.12

```
from typing import Final


class MyString:
    def __init__(self, value: str):
        self.__value: Final = value

    def contains(self, another_string: str) -> bool:
        return True
```

Let's add a failing test for the second scenario:

Figure 6.13

```
class StringTests(unittest.TestCase):
    # ...

    def test_contains__when_both_strings_are_empty(self):
        # GIVEN
        string = MyString('')
        another_string = ''

        # WHEN
        string_contains_another_string =
string.contains(another_string)

        # THEN
        self.assertTrue(string_contains_another_string)
```

We don't have to modify the implementation to make the above test pass. Let's add a test for the third scenario:

Figure 6.14

```
class StringTests(unittest.TestCase):
    # ...

    def test_contains__when_argument_string_is_empty(self):
        # GIVEN
        string = MyString('String')
        another_string = ''

        # WHEN
        string_contains_another_string =
string.contains(another_string)

        # THEN
        self.assertTrue(string_contains_another_string)
```

We don't have to modify the implementation to make the above test pass. Let's add a test for the fourth scenario:

Figure 6.15

```
class StringTests(unittest.TestCase):
    # ...

    def test_contains__when_string_is_empty(self):
```

```
        # GIVEN
        string = MyString('')
        another_string = 'String'

        # WHEN
        string_contains_another_string =
string.contains(another_string)

        # THEN
        self.assertFalse(string_contains_another_string)
```

Let's modify the implementation to make the above (and earlier tests) to pass:

Figure 6.16

```
from typing import Final


class MyString:
    def __init__(self, value: str):
        self.__value: Final = value

    def contains(self, another_string: str) -> bool:
        if not self.__value and another_string:
            return False
        return True
```

Let's add a test for the fifth scenario:

Figure 6.17

```
class StringTests(unittest.TestCase):
    # ...

    def test_contains__when_arg_string_is_found_at_begin(self):
        # GIVEN
        string = MyString("String")
        another_string = "Str"

        # WHEN
        string_contains_another_string =
string.contains(another_string)

        # THEN
        self.assertTrue(string_contains_another_string)
```

We don't have to modify the implementation to make the above test pass. Let's add a test for the sixth scenario:

Figure 6.18

```
class StringTests(unittest.TestCase):
    # ...

    def test_contains__when_arg_string_is_found_at_end(self):
        # GIVEN
        string = MyString("String")
```

```
        another_string = "ng"

        # WHEN
        string_contains_another_string =
string.contains(another_string)

        # THEN
        self.assertTrue(string_contains_another_string)
```

Let’s add a test for the seventh scenario:

Figure 6.19

```
class StringTests(unittest.TestCase):
    # ...

    def
test_contains__when_arg_string_is_longer_than_other_string(self)
:
        # GIVEN
        string = MyString('String')
        another_string = 'String111'

        # WHEN
        string_contains_another_string =
string.contains(another_string)

        # THEN
        self.assertFalse(string_contains_another_string)
```

Let’s modify the implementation:

Figure 6.20

```
from typing import Final


class MyString:
    def __init__(self, value: str):
        self.__value: Final = value

    def contains(self, another_string: str) -> bool:
        if (
            not self.__value
            and another_string
            or len(another_string) > len(self.__value)
        ):
            return False
        return True
```

Let’s add a test for the eigth scenario:

Figure 6.21

```
class StringTests(unittest.TestCase):
    # ...

    def test_contains__when_arg_string_is_found_in_middle(self):
        # GIVEN
        string = MyString('String')
```

```
        another_string = 'ri'

        # WHEN
        string_contains_another_string =
string.contains(another_string)

        # THEN
        self.assertTrue(string_contains_another_string)
```

We don't have to modify the implementation to make the above test pass. Let's add the final test:

Figure 6.22

```
class StringTests(unittest.TestCase):
    # ...

    def test_contains__when_arg_string_is_not_found(self):
        # GIVEN
        string = MyString('String')
        another_string = 'aa'

        # WHEN
        string_contains_another_string =
string.contains(another_string)

        # THEN
        self.assertFalse(string_contains_another_string)
```

Let's modify the implementation to make the above (and earlier tests) to pass:

Figure 6.23

```
from typing import Final


class MyString:
    def __init__(self, value: str):
        self.__value: Final = value

    def contains(self, another_string: str) -> bool:
        if (
            not self.__value
            and another_string
            or len(another_string) > len(self.__value)
            or self.__value.find(another_string) == -1
        ):
            return False
        return True
```

Let's add a test for the eigth scenario:

Figure 6.24

```
class StringTests(unittest.TestCase):
    # ...
```

```
    def test_contains__when_arg_string_is_found_at_middle(self):
        # GIVEN
        string = MyString('String')
        another_string = 'ri'

        # WHEN
        string_contains_another_string =
string.contains(another_string)

        # THEN
        self.assertTrue(string_contains_another_string)
```

We don't have to modify the implementation to make the above test pass.

Next, we should refactor. We can notice that the if-statement condition can be simplified to the following:

Figure 6.25

```
from typing import Final


class MyString:
    def __init__(self, value: str):
        self.__value: Final = value

    def contains(self, another_string: str) -> bool:
        if self.__value.find(another_string) == -1:
            return False
        return True
```

You may have noticed that some refactoring was needed until we came up with the final solution. This is what happens with TDD. You only consider implementation one scenario at a time, which can result in writing code that will be removed/replaced when making a test for the next scenario pass. This is called *emergent design*.

The above example is a bit contrived because we finally used the `find` method, which we could have done already in the first test case, but we didn't because we were supposed to write the simplest implementation to make the test pass. Consider the same example when no `find` method is available, and we must implement the `find` method functionality (looping through the characters, etc.) in the `contains` method ourselves. Then, all the tests start to make sense. Many of them are testing edge/corner cases that are important to test. If you are still in doubt, think about implementing the `contains` method in a language that does not have a for-in loop, but you

must use character indexes. Then, you would finally understand the importance of testing the edge cases: reveal possible off-by-one errors, for example.

When you encounter a bug, it is usually due to a missing scenario: An edge case is not considered, or implementation for a failure scenario or happy path is missing. To remedy the bug, you should practice TDD by adding a failing test for the missing scenario and then make the bug correction in the code to make the added test (and other tests) pass.

There will be one more TDD example later in this chapter when we have an example using BDD, DDD, OOD, and TDD. If you are not yet fully convinced about TDD, the following section presents an alternative to TDD that is still better than doing tests last.

6.1.1.2: Unit Specification-Driven Development (USDD)

For some of you, the above-described TDD cycle may sound cumbersome, or you may not be fully convinced about the benefits it brings. For this reason, I am presenting an alternative to TDD that you can use until you feel ready to try TDD out. The approach I am presenting is inferior to TDD but superior to the traditional "test-last" approach because it reduces the number of bugs by concentrating on the unit (function) specification before the implementation. There are clear benefits in specifying the function behavior beforehand. I call this approach *unit specification-driven development* (USDD). When function behavior is defined first, one is usually less likely to forget to test or implement something. The USDD approach forces you to consider the function specification: happy path(s), possible security issues, edge, and failure cases.

If you don't practice USDD and always do the implementation first, it is more likely you will forget an edge case or a particular failure/security scenario. When you don't practice USDD, you go straight to the implementation, and you tend to think about the happy path(s) only and strive to get them

working. When focusing 100% on getting the happy path(s) working, you don't consider the edge cases and failure/security scenarios. You might forget to implement them or at least some of them. If you forget to implement an edge case or failure scenario, you don't also test it. You can have 100% unit test coverage for a function, but a particular edge case or failure/security scenario is left unimplemented and untested. This is what has happened to me, also. And it has happened more than once. After realizing that the USDD approach could save me from those bugs, I started to take it seriously.

You can conduct USDD as an alternative to TDD/BDD. In USDD, you first specify the unit (i.e., function). You extract all the needed tests from the function specification, including the "happy path" or "happy paths", edge cases, and failure/security scenarios. Then, you put a `fail` call in all the tests so as not to forget to implement them later. Additionally, you can add a comment on the expected result of a test. For example, in failure scenarios, you can write a comment that tells what kind of error is expected to be raised, and in an edge case, you can put a comment that tells with the input of *x*, the output of *y* is expected. (Later, when the tests are implemented, the comments can be removed.)

Let's say that we have the following function specification:

> *Configuration parser's* `parse` *method parses configuration in JSON format into a configuration object. The method should produce an error if the configuration JSON cannot be parsed. Configuration JSON consists of optional and mandatory properties (name and value of specific type). A missing mandatory property should produce an error, and a missing optional property should use a default value. Extra properties should be discarded. A property with an invalid type of value should produce an error. Two property types are supported: integer and string. Integers must have value in a specified range, and strings have a maximum length. The mandatory configuration properties are the following:* `name` *(*`type`*)*

... The optional configuration properties are the following: `name` (`type`) ...

Let's first write a failing test case for the "happy path" scenario:

Figure 6.26

```
import unittest


class ConfigParserTests(unittest.TestCase):
    def test_try_parse__when_successful(self):
        # Happy path scenario, returns a 'Config' object
        self.fail()
```

Next, let's write a failing test case for the other scenarios extracted from the above function specification:

Figure 6.27

```
import unittest


class ConfigParserTests(unittest.TestCase):
    # ...

    def test_try_parse__when_json_parsing_fails(self):
        # Failure scenario, should produce an error
        self.fail()

    def test_try_parse__when_mandatory_prop_is_missing(self):
        # Failure scenario, should produce an error
        self.fail()

    def test_try_parse__when_optional_prop_is_missing(self):
        # Should use default value
        self.fail()

    def test_try_parse__with_extra_props(self):
        # Extra props should be discarded
        self.fail()

    def test_try_parse__when_prop_has_invalid_type(self):
        # Failure scenario, should produce an error
        self.fail()

    def test_try_parse__when_integer_prop_out_of_range(self):
        # Input validation security scenario, should produce an
error
        self.fail()

    def test_try_parse__when_string_prop_too_long(self):
        # Input validation security scenario, should produce an
error
        self.fail()
```

Now, you have a high-level specification of the function in the form of scenarios. Next, you can continue with the function implementation. After you have completed the function implementation, implement the tests one by one and remove the `fail` calls from them.

Compared to TDD, the benefit of this approach is that you don't have to switch continuously between the implementation source code file and the test source code file. In each phase, you can focus on one thing:

1. Function specification
    - What does the function do? (The happy path scenario(s))
    - What failures are possible? (The failure scenario(s))
        - For example, if a function makes a REST API call, all scenarios related to the failure of the call should be considered: connection failure, timeout, response status code not being 2xx, any response data parsing failures
    - Are there security issues? (The security scenarios)
        - For example, if the function gets input from the user, it must be validated, and in case of invalid input, a proper action is taken , like raising an error. Input from the user can be obtained via environment variables, reading files, reading a network socket , and reading standard input.
    - Are there edge cases? (The edge case scenario(s))
    - When you specify the function, it is not mandatory to write the specification down. You can do it in your head if the function is simple. With a more complex function, you might benefit from writing the specification down to fully understand what the function should do
2. Define different scenarios as failing unit tests
3. Function implementation

4. Implementation of unit tests

In real life, the initial function specification is not always 100% correct or complete. During the function implementation, you might discover, e.g., a new failure scenario that was not in the initial function specification. You should immediately add a new failing unit test for that new scenario so you don't forget to implement it later. Once you think your function implementation is complete, go through the function code line-by-line and check if any line can produce an error that has not yet been considered. Having this habit will reduce the possibility of accidentally leaving some error unhandled in the function code.

Sometimes, you need to modify an existing function because you are not always able to follow the *open-closed principle* for various reasons, such as not possible or feasible. When you need to modify an existing function, follow the below steps:

1. Specification of changes to the function
    - What changes in function happy path scenarios?
    - What changes in failure scenarios?
    - What changes in security scenarios?
    - What changes in edge cases?
2. Add/Remove/Modify tests
    - Add new scenarios as failing tests
    - Remove tests for removed scenarios
    - Modify existing tests
3. Implementation changes to the function
4. Implement added unit tests

Let's have an example where we change the configuration parser so that it should produce an error if the configuration contains extra properties. Now we have the specification of the change defined. Next, we need to modify the tests. We need to modify the `test_try_parse__with_extra_props` method as follows:

Figure 6.28

```
import unittest


class ConfigParserTests(unittest.TestCase):
    # ...

    def test_try_parse__with_extra_props(self):
        # Change this scenario so that an error
        # is expected
```

Next, we implement the wanted change and check that all tests pass.

Let's have another example where we change the configuration parser so that the configuration can be given in YAML in addition to JSON. We need to add the following failing unit tests:

Figure 6.29

```
import unittest


class ConfigParserTests(unittest.TestCase):
    # ...

    def test_try_parse__when_yaml_parsing_succeeds(self):
        self.fail()

    def test_try_parse__when_yaml_parsing_fails(self):
        # Should produce an error
        self.fail()
```

We should also rename the following test methods: `test_try_parse__when_successful` and `test_try_parse__when_parsing_fails` to `test_try_parse__when_json_parsing_succeeds` and `test_try_parse__when_json_parsing_fails`. Next, we implement the changes to the function, and lastly, we implement the two new tests. (Depending on the actual test implementation, you may or may not need to make small changes to JSON parsing-related tests to make them pass.)

As the final example, let's make the following change: Configuration has no optional properties, but all properties are mandatory. This means that we can remove the following test: `test_try_parse__when_optional_prop_is_missing`. We

also need to change the `test_try_parse__when_mandatory_prop_is_missing` test:

Figure 6.30

```
import unittest


class ConfigParserTests(unittest.TestCase):
    # ...

    def test_try_parse__when_mandatory_prop_is_missing(self):
        # Change this scenario so that an error
        # is expected
```

Once we have implemented the change, we can run all the tests and ensure they pass.

I have presented two alternative methods for writing unit tests: TDD and USDD. As a professional developer, you should use either of them. TDD brings more benefits, but BDD is much better than *test-last*, i.e., writing unit tests only after implementation is ready. If you think you are not ready for TDD yet, try USDD first and reconsider TDD at some point in the future.

6.1.1.3: Naming Conventions

When functions to be tested are in a class, a respective class for unit tests should be created. For example, if there is a `ConfigParser` class, the respective class for unit tests should be `ConfigParserTests`. This makes locating the file containing unit tests for a particular implementation class easy.

name of the tested method should come. For example, if the tested method is `try_parse`, the test method name should be `test_try_parse`. There are usually several tests for a single function. All test method names should begin with *test_<function-name>*, but the test method name should also contain a description of the specific scenario the test method tests, for example: `test_try_parse__when_parsing_fails`. The name of the tested scenario is separated from the tested function name by two underscores.

6.1.1.4: Mocking

Python has a `unittest.mock` library for mocking in unit tests. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used. Mocks are one form of *test doubles*. Test doubles are any kind of pretend objects used in place of real objects for testing purposes. The following kinds of test doubles can be identified:

- *Fakes* are objects that have working implementations but are usually simplified versions (suitable for testing) of the real implementations.
- *Stubs* are objects providing fixed responses to calls made to them.
- *Spies* are stubs that record information based on how they were called.
- *Mocks* are the most versatile test doubles. They are objects pre-programmed with expectations (e.g., what a method should return when it is called), and like spies, they record information based on how they were called, and those calls can be verified in the test. Mocks are probably the ones you use on a daily basis.

The Python mocking library provides the following ways to mock:

- Patch class/object/attribute/method using `@patch`
- Patch object attribute/method using `@patch.object`
- Create a mock function with `Mock` constructor
- Patch a dictionary

Let's have examples that cover all four different ways of mocking. First, we will have a Kafka client that allows the creation of a topic on a Kafka broker. We want the topic creation to be idempotent, i.e., it does not do anything if the topic already exists. We will use the *unit specification driven development* (USDD) in this exercise by first specifying the functionality of the Kafka client as failing unit tests:

Figure 6.31

```
from unittest import TestCase


class KafkaClientTests(TestCase):
    def test_try_create_topic__when_create_succeeds(self):
        self.fail()

    def test_try_create_topic__when_create_fails(self):
        # Raise an error
        self.fail()

    def test_try_create_topic__when_topic_exists(self):
        self.fail()
```

Next, we will write the implementation for the `KafkaClient` class:

Figure 6.32

```
from confluent_kafka import KafkaError, KafkaException
from confluent_kafka.admin import AdminClient
from confluent_kafka.cimpl import NewTopic

from DataExporterError import DataExporterError


class KafkaClient:
    def __init__(self, kafka_host: str):
        self.__admin_client = AdminClient(
            {'bootstrap.servers': kafka_host}
        )

    class CreateTopicError(DataExporterError):
        pass

    def try_create_topic(
        self,
        name: str,
        num_partitions: int,
        replication_factor: int,
        retention_in_secs: int,
        retention_in_gb: int
    ):
        topic = NewTopic(
            name,
            num_partitions,
            replication_factor,
            config={
                'retention.ms': str(retention_in_secs * 1000),
                'retention.bytes': str(retention_in_gb * pow(10,
9))
            }
        )

        try:
            topic_name_to_creation_future = (
                self.__admin_client.create_topics([topic])
            )

            topic_name_to_creation_future[name].result()
```

```
        except KafkaException as error:
            if error.args[0].code() !=
KafkaError.TOPIC_ALREADY_EXISTS:
                raise self.CreateTopicError(error)
```

Let's implement the first test method to test the successful execution of the `try_create_topic` method:

Figure 6.33

```
from unittest import TestCase
from unittest.mock import Mock, patch

from KafkaClient import KafkaClient


class KafkaClientTests(TestCase):
    @patch('asyncio.Future')
    @patch('KafkaClient.NewTopic')
    @patch('KafkaClient.AdminClient')
    def test_try_create_topic__when_create_succeeds(
        self,
        admin_client_class_mock: Mock,
        new_topic_class_mock: Mock,
        future_class_mock: Mock,
    ):
        # GIVEN
        admin_client_mock = admin_client_class_mock.return_value
        topic_creation_future_mock =
future_class_mock.return_value

        admin_client_mock.create_topics.return_value = {
            'test': topic_creation_future_mock
        }

        kafka_client = KafkaClient('localhost:9092')

        # WHEN
        kafka_client.try_create_topic(
            'test',
            num_partitions=3,
            replication_factor=2,
            retention_in_secs=5 * 60,
            retention_in_gb=100,
        )

        # THEN
        admin_client_class_mock.assert_called_once_with(
            {'bootstrap.servers': 'localhost:9092'}
        )

        new_topic_class_mock.assert_called_once_with(
            'test',
            3,
            2,
            config={
                'retention.ms': str(5 * 60 * 1000),
                'retention.bytes': str(100 * pow(10, 9)),
            },
        )
```

```
        admin_client_mock.create_topics.assert_called_once_with(
            [new_topic_class_mock.return_value]
        )

        topic_creation_future_mock.result.assert_called_once()
```

In the above example, we use two classes, `AdminClient` and `NewTopic` from the Confluent Kafka library. We should not access these dependencies directly in our unit tests. So, we must mock them. This means we patch both `NewTopic` and `AdminClient` classes imported from the `KafkaClient`, which imports them from `confluent_kafka.cimpl` and `confluent_kafka.admin` respectively. The mocks are created using `@patch` decorators. We also mock the `asyncio.Future` class because `AdminClient.create_topics` returns a dict containing a `Future` instance. The mocked versions of the classes are supplied as arguments to the `test_try_create_topic__when_create_succeeds` method. We can access the mocked `AdminClient` and `Future` instances from the mocked classes using the `return_value` property. After executing the test, we need to verify calls to the mocks.

Let's add another test for the case when the topic creation fails:

Figure 6.34

```
from unittest import TestCase
from unittest.mock import Mock, patch

from confluent_kafka import KafkaError, KafkaException
from KafkaClient import KafkaClient


class KafkaClientTests(TestCase):
    @patch('asyncio.Future')
    @patch('KafkaClient.NewTopic')
    @patch('KafkaClient.AdminClient')
    def test_try_create_topic__when_create_fails(
        self,
        admin_client_class_mock: Mock,
        new_topic_class_mock: Mock,
        future_class_mock: Mock,
    ):
        # GIVEN
        kafka_client = KafkaClient('localhost:9092')
        admin_client_mock = admin_client_class_mock.return_value
        topic_creation_future_mock =
future_class_mock.return_value

        admin_client_mock.create_topics.return_value = {
            'test': topic_creation_future_mock
```

```
        }

        topic_creation_future_mock.result.side_effect =
KafkaException(KafkaError(1))

        # WHEN
        try:
            kafka_client.try_create_topic(
                'test',
                num_partitions=3,
                replication_factor=2,
                retention_in_secs=5 * 60,
                retention_in_gb=100,
            )

            self.fail('KafkaClient.CreateTopicError should have
been raised')
        except KafkaClient.CreateTopicError:
            pass

        # THEN
        admin_client_class_mock.assert_called_once_with(
            {'bootstrap.servers': 'localhost:9092'}
        )

        new_topic_class_mock.assert_called_once_with(
            'test',
            3,
            2,
            config={
                'retention.ms': str(5 * 60 * 1000),
                'retention.bytes': str(100 * pow(10, 9)),
            },
        )

        admin_client_mock.create_topics.assert_called_once_with(
            [new_topic_class_mock.return_value]
        )
```

The key in the above test is to make the `Future` mock instance's `result` method raise a `KafkaException` as a side effect. Then, in the test code, we ensure that a `KafkaClient.CreateTopicError` is thrown. If not, we fail the test with a message telling us that a `KafkaClient.CreateTopicError` should have been raised.

The above two test methods contain duplicate code. We should also keep the test code clean. Let's refactor the test case to remove duplicated code. We introduce a `__set_up` method to set up the mocks and create the `KafkaClient` instance. We refactor the common mock call assertions into a separate private method used by both tests. The `@patch` decorators are

moved for the whole test case class, meaning the unit test framework will patch each method starting with a *test* prefix.

Figure 6.35

```
from unittest import TestCase
from unittest.mock import Mock, patch

from confluent_kafka import KafkaError, KafkaException
from KafkaClient import KafkaClient


@patch('asyncio.Future')
@patch('KafkaClient.NewTopic')
@patch('KafkaClient.AdminClient')
class KafkaClientTests(TestCase):
    def test_try_create_topic__when_create_succeeds(
        self,
        admin_client_class_mock: Mock,
        new_topic_class_mock: Mock,
        future_class_mock: Mock,
    ):
        # GIVEN
        self.__set_up(admin_client_class_mock,
future_class_mock)

        # WHEN
        self.kafka_client.try_create_topic('test',
**self.topic_params)

        # THEN
        self.__assert_mock_calls(
            admin_client_class_mock, new_topic_class_mock
        )


self.topic_creation_future_mock.result.assert_called_once()

    def test_try_create_topic__when_create_fails(
        self,
        admin_client_class_mock: Mock,
        new_topic_class_mock: Mock,
        future_class_mock: Mock,
    ):
        # GIVEN
        self.__set_up(admin_client_class_mock,
future_class_mock)
        self.topic_creation_future_mock.result.side_effect =
KafkaException(KafkaError(1))

        # WHEN
        try:
            self.kafka_client.try_create_topic('test',
**self.topic_params)
            self.fail('KafkaClient.CreateTopicError should have
been raised')
        except KafkaClient.CreateTopicError:
            pass

        # THEN
        self.__assert_mock_calls(
```

```
            admin_client_class_mock, new_topic_class_mock
        )

     def __set_up(
        self,
        admin_client_class_mock: Mock,
        future_class_mock: Mock,
    ) -> None:
        # GIVEN
        self.admin_client_mock =
admin_client_class_mock.return_value
        self.topic_creation_future_mock =
future_class_mock.return_value

        self.admin_client_mock.create_topics.return_value = {
            'test': self.topic_creation_future_mock
        }

        self.topic_params = {
            'num_partitions': 3,
            'replication_factor': 2,
            'retention_in_secs': 5 * 60,
            'retention_in_gb': 100,
        }

        self.kafka_client = KafkaClient('localhost:9092')

    def __assert_mock_calls(
        self, admin_client_class_mock: Mock,
new_topic_class_mock: Mock
    ):
        admin_client_class_mock.assert_called_once_with(
            {'bootstrap.servers': 'localhost:9092'}
        )

        new_topic_class_mock.assert_called_once_with(
            'test',
            3,
            2,
            config={
                'retention.ms': str(5 * 60 * 1000),
                'retention.bytes': str(100 * pow(10, 9)),
            },
        )


self.admin_client_mock.create_topics.assert_called_once_with(
            [new_topic_class_mock.return_value]
        )
```

Let’s add implementation for the final test method:

Figure 6.36

```
@patch('asyncio.Future')
@patch('KafkaClient.NewTopic')
@patch('KafkaClient.AdminClient')
class KafkaClientTests(TestCase):
    # ...

    def test_try_create_topic__when_topic_exists(
```

```
        self,
        admin_client_class_mock: Mock,
        new_topic_class_mock: Mock,
        future_class_mock: Mock,
    ):
        # GIVEN
        self.__set_up(admin_client_class_mock,
future_class_mock)
        self.topic_creation_future_mock.result.side_effect = (

KafkaException(KafkaError(KafkaError.TOPIC_ALREADY_EXISTS))
        )

        # WHEN
        self.kafka_client.try_create_topic('test',
**self.topic_params)

        # THEN
        self.__assert_mock_calls(
            admin_client_class_mock, new_topic_class_mock
        )
```

In the above examples, we used `patch` to create mocks for classes. Let's have another example where we patch library methods directly. We should implement an HTTP client that fetches JSON data parsed to a dict from a URL. Let's utilize the USDD and list all possible scenarios for the HTTP client:

- Successfully fetch and parse JSON data from the URL
- Successfully fetch data from the URL, but parsing the data fails. An error should be raised.
- Fetching data from the URL fails with an HTTP status code >=400. An error should be raised.
- Not being able to connect to URL successfully (e.g., malformed URL, connection refused, connection timeout, etc.). An error should be raised.

Let's write a test case with the following failing test methods:

Figure 6.37

```
from unittest import TestCase


class HttpClientTests(TestCase):
    def test_try_fetch_resource__when_fetch_succeeds(self):
        self.fail()

    def test_try_fetch_resource__when_json_parse_fails(self):
        # Should raise an error
        self.fail()
```

```
    def test_try_fetch_resource__when_response_has_error(self):
        # Should raise an error
        self.fail()

    def
test_try_fetch_resource__when_remote_connection_fails(self):
        # Should raise an error
        self.fail()
```

Now we can implement the `HttpClient` class so that it provides the functionality specified by the above test methods.

Figure 6.38

```
from typing import Any

import requests


class HttpClient:
    # Replace the 'Exception' below with the base error
    # class of the software component
    class Error(Exception):
        pass

    def try_fetch_resource(self, url: str) -> dict[str, Any]:
        try:
            response = requests.get(url, timeout=60)
            response.raise_for_status()
            return response.json()
        except requests.RequestException as error:
            raise self.Error(error)
```

If we didn't use BDD, we could have easily ended up with the following implementation, focusing solely on the happy path:

Figure 6.39

```
from typing import Any

import requests


class HttpClient:
    def fetch_resource(self, url: str) -> dict[str, Any]:
        response = requests.get(url, timeout=60)
        response.raise_for_status()
        return response.json()
```

The problem is that it is easy to forget to handle the errors possibly thrown from the `requests.get` and `Response.json` methods if we directly dive into the implementation. Using BDD forces us to stop before implementing anything and think

about the possible error scenarios and edge cases in addition to the happy path scenario.

Let's implement the first test method:

Figure 6.40

```
from unittest import TestCase
from unittest.mock import Mock, patch

from HttpClient import HttpClient

URL = 'https://localhost:8080/'
DICT = {'test': 'test'}


class HttpClientTests(TestCase):
    @patch('requests.Response.__new__')
    @patch('requests.get')
    def test_try_fetch_resource__when_fetch_succeeds(
        self, requests_get_mock: Mock, response_mock: Mock
    ):
        # GIVEN
        requests_get_mock.return_value = response_mock
        response_mock.json.return_value = DICT

        # WHEN
        response_dict = HttpClient().try_fetch_resource(URL)

        # THEN
        requests_get_mock.assert_called_once_with(URL,
timeout=60)
        self.assertDictEqual(response_dict, DICT)
```

Let's implement the second test method:

Figure 6.41

```
import json
from unittest import TestCase
from unittest.mock import Mock, patch

import requests
from HttpClient import HttpClient

URL = 'https://localhost:8080/'
DICT = {'test': 'test'}


class HttpClientTests(TestCase):
    @patch('requests.Response.__new__')
    @patch('requests.get')
    def test_try_fetch_resource__when_json_parse_fails(
        self, requests_get_mock: Mock, response_mock: Mock
    ):
        # GIVEN
        requests_get_mock.return_value = response_mock
        response_mock.json.side_effect =
requests.JSONDecodeError(
            'JSON decode error', json.dumps(DICT), 1
```

```
        )

        # WHEN
        try:
            HttpClient().try_fetch_resource(URL)
            self.fail('HttpClient.Error should have been
raised')
        except HttpClient.Error as error:
            # THEN
            self.assertIn('JSON decode error', str(error))

        # THEN
        requests_get_mock.assert_called_once_with(URL,
timeout=60)
```

We once again have duplicated test code, and we must refactor the tests:

Figure 6.42

```
import json
from unittest import TestCase
from unittest.mock import Mock, patch

import requests
from HttpClient import HttpClient

URL = 'https://localhost:8080/'
DICT = {'test': 'test'}


@patch('requests.Response.__new__')
@patch('requests.get')
class HttpClientTests(TestCase):
    def test_try_fetch_resource__when_fetch_succeeds(
        self, requests_get_mock: Mock, response_mock: Mock
    ):
        # GIVEN
        requests_get_mock.return_value = response_mock
        response_mock.json.return_value = DICT

        # WHEN
        dict_ = HttpClient().try_fetch_resource(URL)

        # THEN
        requests_get_mock.assert_called_once_with(URL,
timeout=60)
        self.assertDictEqual(dict_, DICT)

    def test_try_fetch_resource__when_json_parse_fails(
        self, requests_get_mock: Mock, response_mock: Mock
    ):
        # GIVEN
        requests_get_mock.return_value = response_mock
        response_mock.json.side_effect =
requests.JSONDecodeError(
            'JSON decode error', json.dumps(DICT), 1
        )

        # WHEN
```

```
        self.assertRaises(
            HttpClient.Error, HttpClient().try_fetch_resource,
URL
        )

        # THEN
        requests_get_mock.assert_called_once_with(URL,
timeout=60)
```

Let's add the final two test methods to complete the test case. I also changed to try-except blocks to use `assertRaises` method to showcase an alternative way to verify that a function call raises an error. Use the `assertRaises` method if possible because it shortens and simplifies the test code.

Figure 6.43

```
import json
from unittest import TestCase
from unittest.mock import Mock, patch

import requests
from HttpClient import HttpClient

URL = 'https://localhost:8080/'
DICT = {'test': 'test'}


@patch('requests.Response.__new__')
@patch('requests.get')
class HttpClientTests(TestCase):
    # ...

    def test_try_fetch_resource__when_response_has_error(
        self, requests_get_mock: Mock, response_mock: Mock
    ):
        # GIVEN
        requests_get_mock.return_value = response_mock
        response_mock.raise_for_status.side_effect =
requests.HTTPError()

        # WHEN
        self.assertRaises(
            HttpClient.Error, HttpClient().try_fetch_resource,
URL
        )

        # THEN
        requests_get_mock.assert_called_once_with(URL,
timeout=60)

    def test_try_fetch_resource__when_remote_connection_fails(
        self, requests_get_mock: Mock, response_mock: Mock
    ):
        # GIVEN
        requests_get_mock.side_effect =
requests.ConnectionError()

        # WHEN
```

```
        self.assertRaises(
            HttpClient.Error, HttpClient().try_fetch_resource,
URL
        )

        # THEN
        requests_get_mock.assert_called_once_with(URL,
timeout=60)
```

Let's have an example where we use `@patch.dict`. Let's assume that we have the following code without unit tests:

Figure 6.44

```
import os
import sys

from KafkaClient import KafkaClient


def get_environ_var(name: str) -> str:
    return (
        os.environ.get(name)
        or f'Environment variable {name} is not defined'
    )


def main():
    kafka_client = KafkaClient(get_environ_var('KAFKA_HOST'))

    try:
        kafka_client.try_create_topic(
            get_environ_var('KAFKA_TOPIC'),
            num_partitions=3,
            replication_factor=2,
            retention_in_secs=5 * 60,
            retention_in_gb=100,
        )
    except KafkaClient.CreateTopicError:
        sys.exit(1)


if __name__ == '__main__':
    main()
```

In the unit test case, we use `@patch.dict` to patch the `os.environ` dict. In the second test method, we also use the `@patch.object` decorator instead of the plain `@patch` decorator. The `@patch.object` method patches a method/attribute with a mock in `KafkaClient` type objects.

Figure 6.45

```
import os
from unittest import TestCase
from unittest.mock import Mock, patch

from KafkaClient import KafkaClient
```

```
from main import main

KAFKA_HOST = 'localhost:9092'
KAFKA_TOPIC = 'test'


@patch.dict(os.environ, {'KAFKA_HOST': KAFKA_HOST})
@patch.dict(os.environ, {'KAFKA_TOPIC': KAFKA_TOPIC})
class MainTests(TestCase):
    @patch('main.KafkaClient')
    def test_main__when_exec_succeeds(self,
kafka_client_class_mock: Mock):
        # GIVEN
        kafka_client_mock = kafka_client_class_mock.return_value

        # WHEN
        main()

        # THEN

kafka_client_class_mock.assert_called_once_with(KAFKA_HOST)

kafka_client_mock.try_create_topic.assert_called_once_with(
            KAFKA_TOPIC,
            num_partitions=3,
            replication_factor=2,
            retention_in_secs=5 * 60,
            retention_in_gb=100,
        )

    @patch.object(KafkaClient, '__init__')
    @patch.object(KafkaClient, 'try_create_topic')
    @patch('sys.exit')
    def test_main__when_exec_failed(
        self,
        sys_exit_mock: Mock,
        try_create_topic_mock: Mock,
        kafka_client_init_mock: Mock,
    ):
        # GIVEN
        kafka_client_init_mock.return_value = None
        try_create_topic_mock.side_effect =
KafkaClient.CreateTopicError()

        # WHEN
        main()

        # THEN

kafka_client_init_mock.assert_called_once_with(KAFKA_HOST)
        sys_exit_mock.assert_called_once_with(1)
```

Let's create a unit test case for source code that uses dependency injection. We have the following code from an earlier chapter, and we would like to create a unit test for the `Application` class `run` method. In the below example, we assume that each class is in its own module named according

to the class name, and the `di_container = DiContainer()` definition is in a module named *di_container.py*.

Figure 6.46

```
from enum import Enum
from typing import Protocol

from dependency_injector import containers, providers
from dependency_injector.wiring import Provide, inject

class LogLevel(Enum):
    ERROR = 1
    WARN = 2
    INFO = 3
    # ...


class Logger(Protocol):
    def log(self, log_level: LogLevel, message: str):
        pass


class StdOutLogger(Logger):
    def log(self, log_level: LogLevel, message: str):
        # Log to standard output


class DiContainer(containers.DeclarativeContainer):
    wiring_config = containers.WiringConfiguration(
        modules=['Application']
    )

    logger = providers.Singleton(StdOutLogger)


di_container = DiContainer()


class Application:
    @inject
    def __init__(self, logger: Logger = Provide['logger']):
        self.__logger = logger

    def run(self):
        self.__logger.log(LogLevel.INFO, 'Starting application')
        # ...
```

In the below unit test, we first create a mock instance of the `Logger` class and then override the `logger` provider in the DI container to use the mock. We use the `override` context manager to define the scope of the override.

Figure 6.47

```
from unittest import TestCase
from unittest.mock import Mock

from Application import Application
```

```
from di_container import di_container
from Logger import Logger, LogLevel


class ApplicationTests(TestCase):
    def test_run__when_execution_succeeds(self):
        logger_mock = Mock(Logger)

        with di_container.logger.override(logger_mock):
            # GIVEN
            application = Application()

            # WHEN
            application.run()

            # THEN
            logger_mock.log.assert_called_once_with(
                LogLevel.INFO, 'Starting application'
            )
```

6.1.1.5: Web UI Component Unit Testing

UI component unit testing differs from regular unit testing because you cannot necessarily test the functions of a UI component in isolation if you have, for example, a React functional component. You must conduct UI component unit testing by mounting the component to the DOM and then perform tests by triggering events, for example. This way, you can test the event handler functions of a UI component. The rendering part should also be tested. It can be tested by producing a snapshot of the rendered component and storing that in version control. Further rendering tests should compare the rendered result to the snapshot stored in the version control.

Below is an example of testing the rendering of a React component, `NumberInput`:

Figure 6.48. NumberInput.test.jsx

```
import renderer from 'react-test-renderer';
// ...

describe('NumberInput') () => {
  // ...

  describe('render', () => {
    it('renders with buttons on left and right"', () => {
      const numberInputAsJson =
        renderer
          .create(<NumberInput buttonPlacement="leftAndRight"/>)
          .toJSON();
```

```
      expect(numberInputAsJson).toMatchSnapshot();
    });

    it('renders with buttons on right', () => {
      const numberInputAsJson =
        renderer
          .create(<NumberInput buttonPlacement="right"/>)
          .toJSON();

      expect(numberInputAsJson).toMatchSnapshot();
    });
  });
});
```

Below is an example unit test for the number input's decrement button's click event handler function, `decrementValue`:

Figure 6.49. NumberInput.test.jsx

```
import { render, fireEvent, screen } from '@testing-
library/react'
// ...

describe('NumberInput') () => {
  // ...

  describe('decrementValue', () => {
    it('should decrement value by given step amount', () => {
      render(<NumberInput value="3" stepAmount={2} />);
      fireEvent.click(screen.getByText('-'));
      const numberInputElement = screen.getByDisplayValue('1');
      expect(numberInputElement).toBeTruthy();
    });
  });
});
```

In the above example, we used the testing-library, which has implementations for all the common UI frameworks: React, Vue, and Angular. It means you can use mostly the same testing API regardless of your UI framework. There are tiny differences, basically only in the syntax of the `render` method. If you had implemented some UI components and unit tests for them with React, and you would like to reimplement them with Vue, you don't need to reimplement all the unit tests. You only need to modify them slightly (e.g., make changes to the `render` function calls). Otherwise, the existing unit tests should work because the behavior of the UI component did not change, only its internal implementation technology from React to Vue.

6.1.2: Software Component Integration Testing Principle

> ***Integration testing aims to test that a software component works against actual dependencies and that its public methods correctly understand the purpose and signature of other public methods they are using.***

The target of software component integration testing is that all public functions of the software component should be touched by at least one integration test. Not all functionality of the public functions should be tested because that has already been done in the unit testing phase. This is why there are fewer integration tests than unit tests. The term *integration testing* sometimes refers to integrating a complete software system or a product. However, it should be used to describe software component integration only. When testing a product or a software system, the term *end-to-end testing* should be used to avoid confusion and misunderstandings.

The best way to do the integration testing is using black-box testing. The software component is treated as a black box with inputs and outputs. Test automation developers can use any programming language and testing framework to develop the tests. Integration tests do not depend on the source code. It can be changed or completely rewritten in a different programming language without the need to modify the integration tests. Test automation engineers can also start writing integration tests immediately and don't have to wait for implementation to be ready.

The best way to define integration tests is by using behavior-driven development (BDD). and acceptance test-driven development ATDD. BDD and ATDD encourage teams to use domain-driven design and concrete examples to formalize a shared understanding of how a software component should behave. In BDD and ATDD, behavioral specifications are the root of the integration tests. A development team should create behavioral specifications for each backlog feature. The specifications are the basis for integration tests that also serve

as acceptance tests for the feature. When the team demonstrates a complete feature in a system demo, they should also demonstrate the passing acceptance tests. This practice will shift the integration testing to the left, meaning writing the integration tests can start early and proceed in parallel with the actual implementation. BDD and ATDD also ensure that it is less likely to forget to test some functionality because the functionality is first formally specified as tests before any implementation begins.

When writing behavioral specifications, happy-path scenarios and the main error scenarios should be covered. The idea is not to test every possible error that can occur.

One widely used way to write behavioral specifications is the Gherkin language. However, it is not the only way. So, we cannot say that BDD equals Gherkin. You can even write integration tests using a unit testing framework if you prefer. An example of that approach is available in *Steve Freeman*'s and *Nat Pryce*'s book *Growing Object-Oriented Software, Guided by Tests*.

When using the Gherkin language, the behavior of a software component is described as features. There should be a separate file for each feature. These files have the *.feature* extension. Each feature file describes one feature and one or more scenarios for that feature. The first scenario should be the so-called "happy path" scenario, and other possible scenarios should handle additional happy paths, failures, and edge cases that need to be tested. Remember that you don't have to test every failure and edge case because those were already tested in the unit testing phase.

When the integration tests are black-box tests, the Gherkin features should be end-to-end testable (from software component input to output); otherwise, writing integration tests would be challenging. For example, suppose you have a backlog feature for the data exporter microservice for consuming Avro binary messages from Kafka. In that case, you cannot write an integration test because it is not end-to-end testable. You can't verify that an Avro binary message was successfully read from Kafka because there is no output in the

feature to compare the input with. If you cannot write integration tests for a backlog feature, then you cannot prove and demonstrate to relevant stakeholders that the feature is completed by executing the integration (i.e., acceptance) tests, e.g., in a SAFe system demo. For this reason, it is recommended to make all backlog features such that they can be demonstrated with an end-to-end integration (=acceptance) test case.

Let's consider the data exporter microservice. If we start implementing it from scratch, we should define features in such an order that we first build capability to test end-to-end, for example:

- Export a message from Apache Kafka to Apache Pulsar
- Export single field Avro binary message from Apache Kafka to Apache Pulsar with copy transformation and field type *x*
    - Repeat the above feature for all possible Avro field types (primitive and complex)
- Message filtering
- Type conversion transformations from type x to type y
- Expression transformations
- Output field filtering
- Pulsar export with TLS,
- Kafka export,
- Kafka export with TLS,
- CSV export
- JSON export

The first feature in the above list builds the capability for black-box/E2E integration tests (from software component input to output). This process is also called creating a *walking skeleton* of the software component first. After you have a walking skeleton, you can start adding some "flesh" (other features) around the bones.

Below is a simplified example of one feature in a *data-visualization-configuration-service*. We assume that the service is a REST API. The feature is for creating a new chart. (In a real-life scenario, a chart contains more properties like the chart's data source and what measure(s) and dimension(s) are shown in the chart, for example). In our simplified example, a chart contains the following properties: layout id, type, number of x-axis categories shown, and how many rows of chart data should be fetched from the chart's data source.

Figure 6.50

```
Feature: Create chart
  Creates a new chart

  Scenario: Creates a new chart successfully
    Given chart layout id is 1
    And chart type is "line"
    And X-axis categories shown count is 10
    And fetched row count is 1000

    When I create a new chart

    Then I should get the chart given above
         with response code 201 "Created"
```

The above example shows how the feature's name is given after the `Feature` keyword. You can add free-form text below the feature's name to describe the feature in more detail. Next, a scenario is defined after the `Scenario` keyword. First, the name of the scenario is given. Then comes the steps of the scenario. Each step is defined using one of the following keywords: `Given`, `When`, `Then`, `And`, and `But`. A scenario should follow this pattern:

- Step(s) to describe initial context/setup (Given/And steps)
- Step to describe an event (When step)
- Step(s) to describe the expected outcome for the event (Then/And steps)

We can add another scenario to the above example:

Figure 6.51

```
Feature: Create chart
  Creates a new chart

  Scenario: A new chart is created successfully
```

```
    Given chart layout id is 1
    And chart type is "line"
    And X-axis categories shown count is 10
    And fetched row count is 1000

    When I create a new chart

    Then I should get the chart given above
         with status code 201 "Created"

  Scenario: Chart creation fails due to missing mandatory
parameter
    When I create a new chart

    Then I should get a response with status code 400 "Bad
Request"
    And response body should contain an errorMessage property
with
        "is mandatory field" entry for following fields
      | layoutId                |
      | fetchedRowCount         |
      | xAxisCategShownCount    |
      | type                    |
```

Now, we have one feature with two scenarios specified. Next, we shall implement the scenarios. We want to implement the integration tests in Python, so we will use Behave BDD tool that supports the Gherkin language.

We place integration test code into the source code repository's *integrationtests* directory. The feature files are put in the *integrationtests/features* directory. Feature directories should be organized into subdirectories like source code: creating subdirectories for subdomains. We can put the above *create_chart.feature* file to the *integrationtests/features/chart* directory.

Let's first create an *environment.py* file in the *integration-tests/features* to store things common to all step implementations:

Figure 6.52

```
BASE_URL = 'http://localhost:8080/data-visualization-
configuration-service/'
```

Next, we need to provide an implementation for each step in the scenarios. Let's start with the first scenario. We shall create a *create_chart_steps.py* file in the *src/integrationtests/features/chart/steps* directory for the implementation of the steps:

Figure 6.53

```
import requests
from behave import given, then, when
from behave.runner import Context
from environment import BASE_URL

input_chart = {}


@given('chart layout id is {layout_id:d}')
def step_impl(context: Context, layout_id: int):
    input_chart['layoutId'] = layout_id


@given('chart type is "{type}"')
def step_impl2(context: Context, type: str):
    input_chart['type'] = type


@given('X-axis categories shown count is
{x_axis_categ_shown_count:d}')
def step_impl3(context: Context, x_axis_categ_shown_count: int):
    input_chart['xAxisCategShownCount'] =
x_axis_categ_shown_count


@given('fetched row count is {fetched_row_count:d}')
def step_impl4(context: Context, fetched_row_count: int):
    input_chart['fetchedRowCount'] = fetched_row_count


@when('I create a new chart')
def step_impl5(context: Context):
    context.response = requests.post(BASE_URL + 'charts',
data=input_chart)
    context.response_dict = context.response.json()


@then(
    'I should get the chart given above with status code
{status_code:d} "{reason}"'
)
def step_impl6(context: Context, status_code: int, reason: str):
    assert context.response.status_code == status_code
    assert context.response.reason == reason
    created_chart = context.response_dict
    assert int(created_chart['id']) > 0
    assert created_chart['layoutId'] == input_chart['layoutId']
    assert created_chart['type'] == input_chart['type']

    assert created_chart['xAxisCategShownCount'] == (
        input_chart['xAxisCategShownCount']
    )

    assert created_chart['fetchedRowCount'] == (
        input_chart['fetchedRowCount']
    )
```

The above implementation contains a function for each step. Each function is decorated with a decorator for a specific Gherkin keyword: `@given`, `@when`, and `@then`. Note that a step in a scenario can be templated. For example, the step `Given chart layout id is 1` is templated and implemented in the function `@Given("chart layout id is {layout_id:d}") def step_impl(context: Context, layout_id: int)` where the actual layout id is given as a parameter to the function. You can use this templated step in different scenarios that can give a different value for the layout id, for example: `Given chart layout id is 8`. The `:d` modifier after the `layout_id` tells Behave that this variable should be converted to an integer.

The `@when('I create a new chart')` step implementation uses the *requests* library for submitting an HTTP POST request to the *data-visualization-configuration-service*. The `@then('I should get the chart given above with status code {status_code:d} "{reason}")` step implementation takes the HTTP POST response stored in the `context` and validates the status code and the properties in the response body.

The second scenario is a common failure scenario where you create something with missing parameters. Because this scenario is common (i.e., we can reuse the same steps in other features), we put the step definitions in a file named *common_steps.py* in the *common* subdirectory of the *integrationtests/features/steps* directory.

Here are the step implementations:

Figure 6.54

```
from behave import then
from behave.runner import Context


@then(
    'I should get a response with status code {status_code:d} "
{reason}"'
)
def step_impl1(context: Context, status_code: int, reason: str):
    assert context.response.status_code == status_code
    assert context.response.reason == reason


@then(
```

```
    'response body should contain error object with {error}
entry for following fields'
)
def step_impl2(context: Context, error: str):
    error_message = context.response_dict['errorMessage']
    for field_name in context.table:
        assert f'{field_name} {error}' in error_message
```

To execute the integration tests with Behave, run the `behave` command in the *integration-tests* directory. You can add tags to features and scenarios. You can add a command line parameter for the `behave` command to run features/scenarios with a specific tag or tags only.

Some frameworks offer their way of creating integration tests. For example, the Django web framework offers its own way of doing integration tests. There are two reasons why I don't recommend using framework-specific testing tools. The first reason is that your integration tests are coupled to the framework, and if you decide to reimplement your microservice using a different language or framework, you also need to reimplement the integration tests. When you use a generic BDD tool like Behave, your integration tests are not coupled to any microservice implementation programming language or framework. The second reason is that there is less learning and information burden for QA/test engineers when they don't have to master multiple framework-specific integration testing tools. If you use a single tool like Behave in all the microservices in a software system, it will be easier for QA/test engineers to work with different microservices.

For API microservices, one more alternative to implement integration tests is an API development platform like Postman. Postman can be used to write integration tests using JavaScript.

Suppose we have an API microservice called *sales-item-service* that offers CRUD operations on sales items. Below is an example API request for creating a new sales item. You can define this in Postman as a new request:

Figure 6.55

```
POST http://localhost:3000/sales-item-service/sales-items
HTTP/1.1
Content-Type: application/json
```

```
{
  "name": "Test sales item",
  "price": 10,
}
```

Here is a Postman test case to validate the response to the above request:

Figure 6.56

```
pm.test("Status code is 201 Created", function () {
  pm.response.to.have.status(201);
});

const salesItem = pm.response.json();
pm.collectionVariables.set("salesItemId", salesItem.id)

pm.test("Sales item name", function () {
  return pm.expect(salesItem.name).to.eql("Test sales item");
})

pm.test("Sales item price", function () {
  return pm.expect(salesItem.price).to.eql(10);
})
```

In the above test case, the response status code is verified first, and then the `salesItem` object is parsed from the response body. Value for the variable `salesItemId` is set. This variable will be used in subsequent test cases. Finally, the values of the `name` and `price` properties are checked.

Next, a new API request could be created in Postman to retrieve the just created sales item:

Figure 6.57

```
GET http://localhost:3000/sales-item-service/sales-
items/{{salesItemId}} HTTP/1.1
```

We used the value stored earlier in the `salesItemId` variable in the request URL. Variables can be used in the URL and request body using the following notation: `{{<variable-name>}}`. Let's create a test case for the above request:

Figure 6.58

```
pm.test("Status code is 200 OK", function () {
  pm.response.to.have.status(200);
});

const salesItem = pm.response.json();

pm.test("Sales item name", function () {
  return pm.expect(salesItem.name).to.eql("Test sales item");
})
```

```
pm.test("Sales item price", function () {
  return pm.expect(salesItem.price).to.eql(10);
})
```

API integration tests written in Postman can be utilized in a CI pipeline. An easy way to do that is to export a Postman collection to a file that contains all the API requests and related tests. A Postman collection file is a JSON file. Postman offers a Node.js command-line utility called Newman. It can run API requests and related tests from an exported Postman collection file.

YYou can run integration tests stored in an exported Postman collection file with the below command in a CI pipeline:

Figure 6.59

```
newman run
integrationtests/integrationTestsPostmanCollection.json
```

In the above example, we assume that a file named *integrationTestsPostmanCollection.json* has been exported from the Postman to the *integrationtests* directory in the source code repository.

6.1.2.1: Web UI Integration Testing

You can also use the Gherkin language when specifying UI features. For example, the TestCafe UI testing tool can be used with the gherkin-testcafe tool to make TestCafe support the Gherkin syntax. Let's create a simple UI feature:

Figure 6.60

```
Feature: Greet user
  Entering user name and clicking submit button
  displays a greeting for the user

  Scenario: Greet user successfully
    Given there is "John Doe" entered in the input field
    When I press the submit button
    Then I am greeted with text "Hello, John Doe"
```

Next, we can implement the above steps in JavaScript using the TestCafe testing API:

Figure 6.61

```
// Imports...

// 'Before' hook runs before the first step of each scenario.
// 't' is the TestCafe test controller object
Before('Navigate to application URL', async (t) => {
  // Navigate browser to application URL
  await t.navigateTo('...');
});

Given('there is {string} entered in the input field',
      async (t, [userName]) => {
  // Finds an HTML element with CSS id selector and
  // enters text to it
  await t.typeText('#user-name', userName);
});

When('I press the submit button', async (t) => {
  // Finds an HTML element with CSS id selector and clicks it
  await t.click('#submit-button');
});

When('I am greeted with text {string}', async (t, [greeting]) =>
{
  // Finds an HTML element with CSS id selector
  // and compares its inner text
  await t.expect(Selector('#greeting').innerText).eql(greeting);
});
```

Another tool similar to TestCafe is Cypress. You can also use Gherkin with Cypress with the cypress-cucumber-preprocessor library. Then, you can write your UI integration tests like this:

Figure 6.62

```
Feature: Visit duckduckgo.com website

  Scenario: Visit duckduckgo.com website successfully
    When I visit duckduckgo.com
    Then I should see the search bar
```

Figure 6.63

```
import { When, Then } from
  '@badeball/cypress-cucumber-preprocessor';

When("I visit duckduckgo.com", () => {
  cy.visit("https://www.duckduckgo.com");
});

Then("I should see the search bar", () => {
  cy.get("input").should(
    "have.attr",
    "placeholder",
    "Search the web without being tracked"
  );
});
```

6.1.2.2: Setting Up Integration Testing Environment

Before integration tests can be run, an integration testing environment must be set up. An integration testing environment is where the tested microservice and all its dependencies are running. The easiest way to set up an integration testing environment for a containerized microservice is to use Docker Compose, a simple container orchestration tool for a single host.

Let's create a *docker-compose.yml* file for the *sales-item-service* microservice, which has a MySQL database as a dependency. The microservice uses the database to store sales items.

Figure 6.64. docker-compose.yaml

```
version: "3.8"

services:
  wait-for-services-ready:
    image: dokku/wait
  sales-item-service:
    restart: always
    build:
      context: .
    env_file: .env.ci
    ports:
      - "3000:3000"
    depends_on:
      - mysql
  mysql:
    image: mysql:8.0.22
    command: --default-authentication-
plugin=mysql_native_password
    restart: always
    cap_add:
      - SYS_NICE
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_PASSWORD}
    ports:
      - "3306:3306"
```

In the above example, we first define a service *wait-for-services-ready*, which we will use later. Next, we define our microservice, *sales-item-service*. We ask Docker Compose to build a container image for the *sales-item-service* using the *Dockerfile* in the current directory. Then, we define the environment for the microservice to be read from an *.env.ci*

file. We expose port 3000 and tell that our microservice depends on the *mysql* service.

Next, we define the `mysql` service. We tell what image to use, give a command-line parameter, and define the environment and expose a port.

Before we can run the integration tests, we must spin the integration testing environment up using the `docker-compose up` command:

Figure 6.65

```
docker-compose up --env-file .env.ci --build -d
```

We tell the `docker-compose` command to read environment variables from an *.env.ci* file that should contain an environment variable named `MYSQL_PASSWORD`. We ask Docker Compose to always build the *sales-item-service* by specifying the `--build` flag. The `-d` flag tells the `docker-compose` command to run in the background.

Before we can run the integration tests, we must wait until all services defined in the *docker-compose.yml* are up and running. We use the *wait-for-services-ready* service provided by the dokku/wait image. We can wait for the services to be ready by issuing the following command:

Figure 6.66

```
docker-compose
  --env-file .env.ci
  run wait-for-services-ready
  -c mysql:3306,sales-item-service:3000
  -t 600
```

The above command will finish after *mysql* service's port 3306 and *sales-item-service's* port 3000 can be connected (as specified with the `-c` flag, the `-t` flag specifies a timeout for waiting). After the above command is finished, you can run the integration tests. In the below example, we run the integration tests using the *newman* CLI tool:

Figure 6.67

```
newman run
integrationtests/integrationTestsPostmanCollection.json
```

If your integration tests are implemented using Behave, you can run them by going to the *integrationtests* directory and run the `behave` command there. Instead of using the *dokku/wait* image for waiting services to be ready, you can do the waiting in Behave's `before_all` function. Just make a loop that tries to make a TCP connection to `mysql:3306` and `sales-item-service:3000`. When both connections succeed, break the loop to start the tests.

After integration tests are completed, you can shut down the integration testing environment:

Figure 6.68

```
docker-compose down
```

If you need other dependencies in your integration testing environment, you can add them to the *docker-compose.yml* file. If you need to add other microservices with dependencies, you must also add transitive dependencies. For example, if you needed to add another microservice that uses a PostgreSQL database, you would have to add the other microservice and PostgreSQL database to the *docker-compose.yml* file as new services.

Let's say the *sales-item-service* depends on Apache Kafka 2.x that depends on a Zookeeper service. The *sales-item-service's docker-compose.yml* looks like the below after adding Kafka and Zookeeper:

Figure 6.69. docker-compose.yaml

```
version: "3.8"

services:
  wait-for-services-ready:
    image: dokku/wait
  sales-item-service:
    restart: always
    build:
      context: .
    env_file: .env.ci
    ports:
      - 3000:3000
    depends_on:
      - mysql
      - kafka
  mysql:
    image: mysql:8.0.22
    command: --default-authentication-
plugin=mysql_native_password
```

```
    restart: always
    cap_add:
      - SYS_NICE
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_PASSWORD}
    ports:
      - "3306:3306"
  zookeeper:
    image: bitnami/zookeeper:3.7
    volumes:
      - "zookeeper_data:/bitnami"
    ports:
      - 2181:2181"
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
  kafka:
    image: bitnami/kafka:2.8.1
    volumes:
      - "kafka_data:/bitnami"
    ports:
      - "9092:9092"
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
    depends_on:
      - zookeeper

volumes:
  zookeeper_data:
    driver: local
  kafka_data:
    driver: local
```

6.1.3: Complete Example with BDD, ATDD, DDD, OOD and TDD

Let's have a complete example using the following design principles: BDD, ATDD, DDD, OOD, and TDD. We will implement a *gossiping bus drivers* application, which some of you might be familiar with. Product management gives us the following user story:

> *Each bus driver drives a bus along a specified circular route. A route consists of one or more bus stops. Bus drivers drive the route and stop at each bus stop. Bus drivers have a set of rumors. At the bus stop, drivers gossip (share rumors) with other drivers stopped at the same bus stop. The application stops when all rumors are shared, or bus drivers have driven for a maximum number of bus stops. Upon exit, the application should*

inform the user whether all rumors were successfully shared.

We start with BDD and ATDD and write a formal behavioral specification for the above informal description:

Figure 6.70

```
Feature: Gossiping bus drivers

  Scenario: Bus drivers successfully share all rumors
    Given maximum number of bus stops driven is 100
    Given bus drivers with the following routes and rumors
      | Route                                  | Rumors
|
      | stop-a, stop-b, stop-c                 | rumor1, rumor2
|
      | stop-d, stop-b, stop-e                 | rumor1,
rumor3, rumor4 |
      | stop-f, stop-g, stop-h, stop-i, stop-e  | rumor1,
rumor5, rumor6 |

    When bus drivers have completed driving
    Then all rumors are successfully shared

  Scenario: Bus drivers fails to share all rumors due to driving
maximum number of stops
    Given maximum number of bus stops driven is 5
    Given bus drivers with the following routes and rumors
      | Route                                  | Rumors
|
      | stop-a, stop-b, stop-c                 | rumor1, rumor2
|
      | stop-d, stop-b, stop-e                 | rumor1,
rumor3, rumor4 |
      | stop-f, stop-g, stop-h, stop-i, stop-e  | rumor1,
rumor5, rumor6 |

      When bus drivers have completed driving
      Then all rumors are not shared

  Scenario: Bus drivers fail to share all rumors because bus
routes never cross
     Given maximum number of bus stops driven is 100
     Given bus drivers with the following bus routes and rumors
       | Route                   | Rumors                 |
       | stop-a, stop-b, stop-c   | rumor1, rumor2         |
       | stop-d, stop-e, stop-f   | rumor1, rumor3, rumor4 |

     When bus drivers have completed driving
     Then all rumors are not shared
```

Next, we add a task to the team's backlog for write integration (acceptance) tests for the above scenarios. The implementation of the integration tests can start parallel to the actual implementation of the user story. The user story description

provided by product management does not specify how below things should be exactly implemented:

- Upon application exit, inform the user whether all rumors were successfully shared or not
- How bus drivers are supplied to the application

The team should discuss the above two topics and consult the product manager for specific requirements. If there is no feedback from the product manager, the team can decide how to implement the above things. For example, the team could specify the following:

- If all rumors were successfully shared, the application should exit with an exit code zero; otherwise, exit with a non-zero exit code.
- Application gets the maximum number of driven bus stops as the first command line parameter
- Application gets drivers as subsequent command line parameters
- Each driver is specified with a string in the following format: , e.g., `bus-stop-a,bus-stop-b;rumor-1,rumor-2`

Below are the step implementations for the above Gherkin feature specification:

Figure 6.71

```
import subprocess

from behave import given, then, when
from behave.runner import Context


@given(
    'maximum number of bus stops driven is
{max_driven_bus_stop_count:d}'
)
def step_impl(context: Context, max_driven_bus_stop_count: int):
    context.max_driven_bus_stop_count =
max_driven_bus_stop_count


@given('bus drivers with the following routes and rumors')
def step_impl2(context: Context):
    context.drivers = []
```

```
    for driver in context.table:
        bus_route = ','.join(
            bus_stop.strip() for bus_stop in
driver['Route'].split(',')
        )

        rumors = ','.join(
            rumor.strip() for rumor in
driver['Rumors'].split(',')
        )

        context.drivers.append(f'{bus_route};{rumors}')


@when('bus drivers have completed driving')
def step_impl3(context: Context):
    context.exit_code = subprocess.run(
        [
            'python main.py',
            str(context.max_driven_bus_stop_count),
            ' '.join(context.drivers),
        ]
    ).returncode


@then('all rumors are successfully shared')
def step_impl4(context: Context):
    assert context.exit_code == 0


@then('all rumors are not shared')
def step_impl5(context: Context):
    assert context.exit_code != 0
```

In the `@given` steps, we store information in the `context`. We store driver definitions as strings to the `drivers` attribute of the `context`. In the `@when` step, the application is launched with command line arguments, and in the `@then` steps, the exit code of the sub-process is examined to be either 0 (successful sharing of all rumors) or non-zero (sharing of all rumors failed).

Before starting the implementation using TDD, we must first design our application using DDD and then OOD. Let's continue with the DDD phase. We can start with event storming and define the *domain events* first:

- The maximum number of driven bus stops is parsed from the command line
- Bus drivers are parsed from the command line

- Bus drivers have driven until all rumors have been shared
- The bus driver has driven to the next bus stop according to the bus route
- Rumors are shared with the drivers at the bus stop

Let's introduce the *actors*, *commands* and *entities* related to the above domain events:

- The maximum number of driven bus stops is parsed from the command line
    - Actor: MaxDrivenStopCountParser, Command: parse
- Bus drivers are parsed from the command line
    - Actor: BusDriversParser, Command: parse, Entities: BusDriver
- Bus drivers have driven until all rumors have been shared
    - Actor: GossipingBusDrivers, Command: drive_until_all_rumors_shared, Entities: Rumor, BusDriver
- The bus driver has driven to the next bus stop according to the bus route
    - Actor: BusDriver, Command: drive_to_next_bus_stop, Entities: BusStop
    - Actor: BusRoute, Command: get_next_bus_stop, Entities: BusStop
- Rumors are shared with the drivers at the bus stop
    - Actor: BusStop, Command: share_rumors_with_drivers, Entities BusDriver, Rumor

Based on the above output of the DDD phase, we can design our classes using OOD. We design actor classes and put public behavior to them and design the `Rumor` entity class with no behavior. Below is the class diagram:

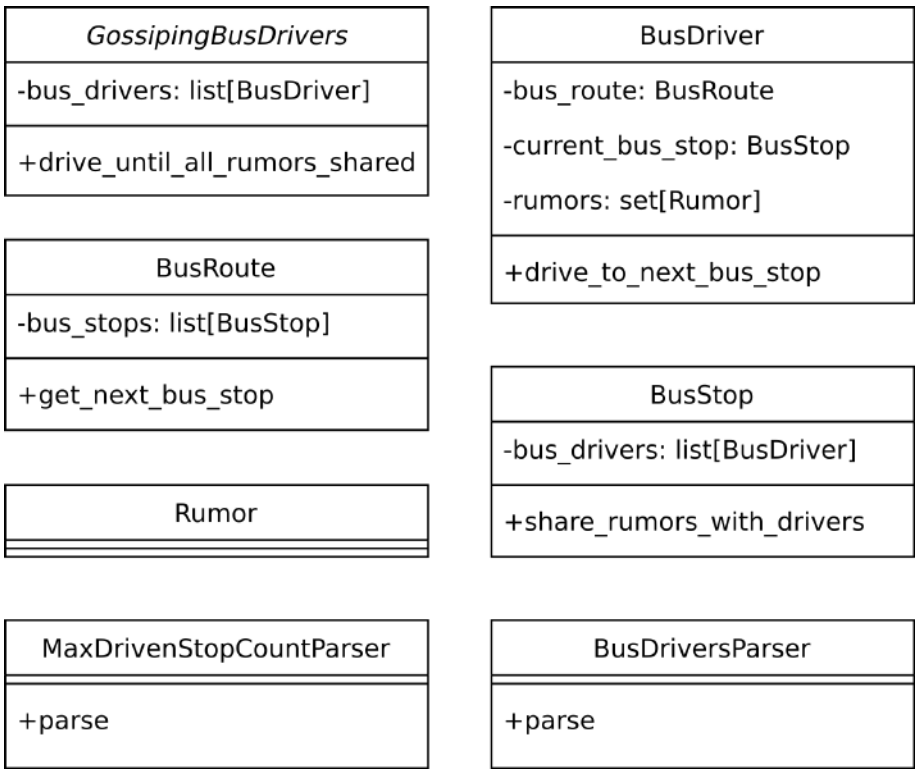

Figure 6.72. Class diagram

We should not forget the *program against interfaces principle* (dependency inversion principle). So, let's add interfaces to the class diagram:

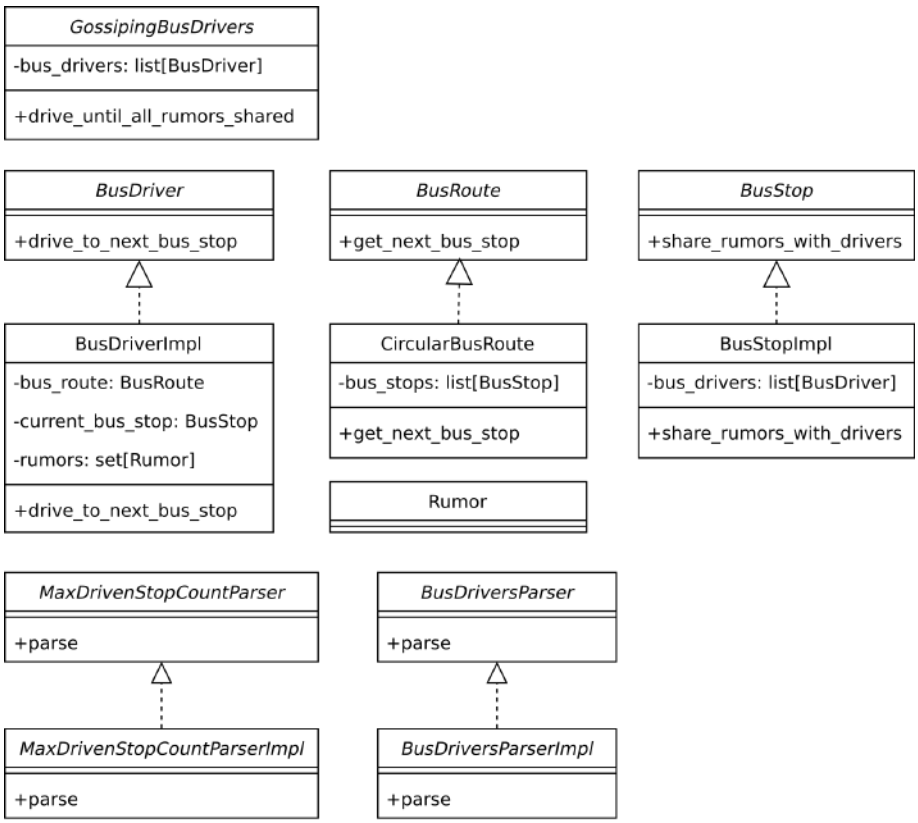

Figure 6.73. Class diagram with interfaces

Now that we have our design done, we can add the following tasks to the team's backlog:

- GossipingBusDrivers class implementation
- BusDriverImpl class implementation
- CircularBusRoute class implementation
- BusStopImpl class implementation
- Rumor class implementation
- MaxDrivenStopCountParser class implementation
- BusDriversParser class implementation

We already had the integration tests implementation task added to the backlog earlier. There are eight tasks in the backlog, and each can be implemented (either entirely or at least partially) in parallel. If the team has eight team members, each of them can pick a task for themselves and proceed in parallel to complete the user story as fast as possible.

We can start implementing classes one public method at a time. Let's start with the most straightforward class, `Rumor`, which does not have behavior at all:

Figure 6.74

```
class Rumor:
    pass
```

Next, we can implement the `CircularBusRoute` class using TDD. Let's start with the simplest case, which is also a failure scenario: If the bus route has no bus stops, a `ValueError` with an informative message should be raised. Let's implement a unit test for that scenario:

Figure 6.75

```
class CircularBusRouteTests(unittest.TestCase):
    def test_constructor__when_no_bus_stops(self):
        try:
            # WHEN
            CircularBusRoute([])

            self.fail('ValueError should have been raised')
        except ValueError as error:
            # THEN
            self.assertEqual(
                str(error), 'Bus route must have at least one
bus stop'
            )
```

Let's implement the constructor of the `CircularBusRoute` class to make the above test pass:

Figure 6.76

```
class BusStop(Protocol):
    pass


class BusRoute(Protocol):
    pass


class CircularBusRoute(BusRoute):
    def __init__(self, bus_stops: list[BusStop]):
        if not bus_stops:
            raise ValueError('Bus route must have at least one
bus stop')

        self.__bus_stops: Final = bus_stops.copy()
```

The unit test for the next scenario is the following: if there is only one stop in the bus route, the `get_next_bus_stop` method

should always return that bus stop (because the bus route is circular).

Figure 6.77

```
class CircularBusRouteTests(unittest.TestCase):
    def test_get_next_bus_stop__when_one_bus_stop(self):
        # GIVEN
        bus_stop = BusStopImpl()
        bus_route = CircularBusRoute([bus_stop])

        # WHEN
        next_bus_stop = bus_route.get_next_bus_stop(bus_stop)

        # THEN
        self.assertEqual(next_bus_stop, bus_stop)
```

Let's implement the `get_next_bus_stop` method to make the above test to pass:

Figure 6.78

```
class BusRoute(Protocol):
    def get_next_bus_stop(self, current_bus_stop: BusStop):
        pass


class CircularBusRoute(BusRoute):
    # ...

    def get_next_bus_stop(self, current_bus_stop: BusStop) ->
BusStop:
        return self.__bus_stop[0]
```

Let's implement a unit test for the following scenario: If the `get_next_bus_stop` method's argument is a bus stop not belonging to the bus route, a `ValueError` with an informative message should be raised.

Figure 6.79

```
class CircularBusRouteTests(unittest.TestCase):
    # GIVEN
    bus_stop_a = BusStopImpl()
    bus_route = CircularBusRoute([bus_stop_a])
    bus_stop_b = BusStopImpl()

    try:
        # WHEN
        bus_route.get_next_bus_stop(bus_stop_b)

        self.fail('ValueError should have been raised')
    except ValueError as error:
        # THEN
        self.assertEqual(
            str(error), 'Bus stop does not belong to bus route'
        )
```

Let's modify the `get_next_bus_stop` implementation:

Figure 6.80

```
class CircularBusRoute(BusRoute):
    # ...

    def get_next_bus_stop(self, current_bus_stop: BusStop) ->
BusStop:
        if current_bus_stop not in self.__bus_stops:
            raise ValueError('Bus stop does not belong to bus
route')

        return self.__bus_stop[0]
```

Next, we specify two scenarios of how the `get_next_bus_stop` method should behave when there is more than one stop in the bus route:

> *When a current bus stop is given, the `get_next_bus_stop` method should return the next bus stop in a list of bus stops for the route.*

Let's write a failing unit test for the above scenario:

Figure 6.81

```
class CircularBusRouteTests(unittest.TestCase:
    def
test_get_next_bus_stop__when_next_bus_stop_in_list_exists(self):
        # GIVEN
        bus_stop_a = BusStopImpl()
        bus_stop_b = BusStopImpl()
        bus_route = CircularBusRoute([bus_stop_a, bus_stop_b])

        # WHEN
        next_bus_stop = bus_route.get_next_bus_stop(bus_stop_a)

        # THEN
        self.assertEqual(next_bus_stop, bus_stop_b)
```

Let's modify the source code to make the above test pass:

Figure 6.82

```
class CircularBusRoute(BusRoute):
    # ...

    def get_next_bus_stop(self, current_bus_stop: BusStop) ->
BusStop:
        if current_bus_stop not in self.__bus_stops:
            raise ValueError('Bus stop does not belong to bus
route')

        if len(self.__bus_stops) == 1:
```

```
            return self.__bus_stops[0]

        curr_bus_stop_index =
self.__bus_stops.index(current_bus_stop)
        return self.__bus_stops[curr_bus_stop_index + 1]
```

Let's add a test for the following scenario:

> *If there is no next bus stop in the list of the route's bus stops, the `get_next_bus_stop` method should return the first bus stop (due to the route being circular).*

Figure 6.83

```
class CircularBusRouteTests(unittest.TestCase):
    def
test_get_next_bus_stop__when_no_next_bus_stop_in_list(self):
        # GIVEN
        bus_stop_a = BusStopImpl()
        bus_stop_b = BusStopImpl()
        bus_route = CircularBusRoute([bus_stop_a, bus_stop_b])

        # WHEN
        next_bus_stop = bus_route.get_next_bus_stop(bus_stop_b)

        # THEN
        self.assertEqual(next_bus_stop, bus_stop_a)
```

Let's make the above test pass:

Figure 6.84

```
class CircularBusRoute(BusRoute):
    # ...

    def get_next_bus_stop(self, current_bus_stop: BusStop) ->
BusStop:
        if current_bus_stop not in self.__bus_stops:
            raise ValueError('Bus stop does not belong to bus
route')

        bus_stop_count = len(self.__bus_stops)

        if bus_stop_count == 1:
            return self.__bus_stops[0]

        curr_bus_stop_index =
self.__bus_stops.index(current_bus_stop)
        next_bus_stop_index = curr_bus_stop_index + 1

        if (next_bus_stop_index <= bus_stop_count - 1):
            return self.__bus_stops[next_bus_stop_index]
        else:
            return self.__bus_stops[0]
```

Our code could benefit from refactoring. With the help of existing unit tests, we can safely refactor the code to the following:

Figure 6.85

```
class CircularBusRoute(BusRoute):
    def __init__(self, bus_stops: list[BusStop]):
        if not bus_stops:
            raise ValueError('Bus route must have at least one
bus stop')

        self.__bus_stops: Final = bus_stops.copy()
        self.__bus_stop_count: Final = len(bus_stops)

    def get_next_bus_stop(self, current_bus_stop: BusStop) ->
BusStop:
        try:
            curr_bus_stop_index =
self.__bus_stops.index(current_bus_stop)
        except ValueError:
            raise ValueError('Bus stop does not belong to bus
route')

        next_bus_stop_index = (
            curr_bus_stop_index + 1
        ) % self.__bus_stop_count

        return self.__bus_stops[next_bus_stop_index]
```

Next, we shall implement the `BusStopImpl` class and the `share_rumors_with_drivers` method. Let's start by specifying what that method should do: After the execution of the method is completed, all the drivers at the bus stop should have the same set of rumors that is a union of all the rumors that the drivers at the bus stop have.

Let's create a test for the above specification. We will have three bus drivers at a bus stop. Because bus drivers are implemented in a separate class, we will use mocks for the drivers.

Figure 6.86

```
class BusStopImplTests(unittest.TestCase):
    @patch('BusDriverImpl.__new__')
    @patch('BusDriverImpl.__new__')
    @patch('BusDriverImpl.__new__')
    def test_share_rumors_with_drivers(
        self,
        bus_driver_mock3: Mock,
        bus_driver_mock2: Mock,
        bus_driver_mock1: Mock,
    ):
        # GIVEN
        bus_drivers = [bus_driver_mock1, bus_driver_mock2,
```

```
bus_driver_mock3]

        rumor1 = Rumor()
        rumor2 = Rumor()
        rumor3 = Rumor()
        all_rumors = {rumor1, rumor2, rumor3}

        bus_driver_mock1.get_rumors.return_value = {rumor1,
rumor2}
        bus_driver_mock2.get_rumors.return_value = {rumor2}
        bus_driver_mock3.get_rumors.return_value = {rumor2,
rumor3}

        bus_stop = BusStopImpl()
        bus_stop.add(bus_driver_mock1)
        bus_stop.add(bus_driver_mock2)
        bus_stop.add(bus_driver_mock3)

        # WHEN
        bus_stop.share_rumors_with_drivers()

        # THEN
        self.__assert_rumors_are_set(all_rumors, bus_drivers)

    def __assert_rumors_are_set(self, all_rumors,
bus_driver_mocks):
        for bus_driver_mock in bus_driver_mocks:

bus_driver_mock.set_rumors.assert_called_with(all_rumors)
```

Let's implement the `BusStopImpl` class to make the above test pass:

Figure 6.87

```
class BusDriver(Protocol):
    def get_rumors(self) -> set[Rumor]:
        pass

    def set_rumors(self, rumors: set[Rumor]) -> None:
       pass


class BusStop(Protocol):
    def share_rumors_with_drivers(self) -> None:
        pass

    def add_bus_driver(self, bus_driver: BusDriver) -> None:
        pass


class BusStopImpl(BusStop):
    def __init__(self):
        self.__bus_drivers: Final = set()

    def share_rumors_with_drivers(self) -> None:
        all_rumors = {
            rumor
            for bus_driver in self.__bus_drivers
            for rumor in bus_driver.get_rumors()
```

```
        }

        for bus_driver in self.__bus_drivers:
            bus_driver.set_rumors(all_rumors)

    def add(self, bus_driver: BusDriver) -> None:
        self.__bus_drivers.add(bus_driver)
```

Let's implement the `BusDriverImpl` class next. As shown above, we needed to add two methods, `get_rumors` and `set_rumors`, to the `BusDriver` interface. Let's make a unit test for the `get_rumors` method, which returns the rumors given for the driver in the constructor:

Figure 6.88

```
class BusDriveImplTests(unittest.TestCase):
    rumor1 = Rumor()
    rumor2 = Rumor()

    @patch('CircularBusRoute.__new__')
    def test_get_rumors(self, bus_route_mock: Mock):
        # GIVEN
        bus_driver = BusDriverImpl(
            bus_route_mock, {self.rumor1, self.rumor2}
        )

        # WHEN
        rumors = bus_driver.get_rumors()

        # THEN
        self.assertEqual(rumors, {self.rumor1, self.rumor2})
```

Now we can implement the `get_rumors` method to make the above test pass:

Figure 6.89

```
class BusDriverImpl(BusDriver):
    def __init__(self, rumors: set[Rumor]):
        self.__rumors = rumors.copy()

    def get_rumors(self) -> set[Rumor]:
        return self.__rumors
```

Let's make a unit test for the `set_rumors` method, which should override the rumors given in the constructor:

Figure 6.90

```
class BusDriveImplTests(unittest.TestCase):
    @patch('CircularBusRoute.__new__')
    def test_set_rumors(self, bus_route_mock: Mock):
        # GIVEN
        rumor3 = Rumor()
        rumor4 = Rumor()
```

```
        bus_driver = BusDriverImpl(
            bus_route_mock, {self.rumor1, self.rumor2}
        )

        # WHEN
        bus_driver.set_rumors({rumor3, rumor4})

        # THEN
        self.assertEqual(bus_driver.get_rumors(), {rumor3,
rumor4})
```

Now we can implement the `get_rumors` method to make the above test pass:

Figure 6.91

```
class BusDriverImpl(BusDriver):
    # ...

    def set_rumors(self, rumors: set[Rumor]) -> None:
        self.__rumors = rumors.copy()
```

We need to implement the `drive_to_next_bus_stop` method in the `BusDriverImpl` class: A bus driver has a current bus stop that is initially the first bus stop of the route. The driver drives to the next bus stop according to its route. When the driver arrives at the next bus stop, the driver is added to the bus stop. The driver is removed from the current bus stop, which is changed to the next bus stop.

Figure 6.92

```
class BusDriverImplTests(unittest.TestCase):
    @patch('BusStopImpl.__new__')
    @patch('BusStopImpl.__new__')
    @patch('CircularBusRoute.__new__')
    def test_drive_to_next_bus_stop(
        self,
        bus_route_mock: Mock,
        bus_stop_a_mock: Mock,
        bus_stop_b_mock: Mock,
    ):
        # GIVEN
        bus_route_mock.get_first_bus_stop.return_value =
bus_stop_a_mock
        bus_route_mock.get_next_bus_stop.return_value =
bus_stop_b_mock
        bus_driver = BusDriverImpl(bus_route_mock, set())

        # WHEN
        bus_driver.drive_to_next_bus_stop()

        # THEN
        bus_stop_a_mock.remove.assert_called_with(bus_driver)
        bus_stop_b_mock.add.assert_called_with(bus_driver)
```

```
        self.assertEqual(
            bus_driver.get_current_bus_stop(), bus_stop_b_mock
        )
```

Let's implement the `BusDriverImpl` class to make the above test pass. Before that, we must add a test for the new `get_first_bus_stop` method in the `BusRouteImpl` class.

Figure 6.93

```
class CircularBusRouteTests(unittest.TestCase):
    def test_get_first_bus_stop(self):
        # GIVEN
        bus_stop_a = BusStopImpl()
        bus_stop_b = BusStopImpl()
        bus_route = CircularBusRoute([bus_stop_a, bus_stop_b])

        # WHEN
        first_bus_stop = bus_route.get_first_bus_stop()

        # THEN
        self.assertEqual(first_bus_stop, bus_stop_a)
```

Let's modify the `CircularBusRoute` class to make the above test pass:

Figure 6.94

```
class BusRoute(Protocol):
    # ...

    def get_first_bus_stop(self) -> BusStop:
        pass


class CircularBusRoute(BusRoute):
    # ...

    def get_first_bus_stop(self) -> BusStop:
        return self.__bus_stops[0]
```

Here is the implementation of the `drive_to_next_bus_stop` method in the `BusDriverImpl` class:

Figure 6.95

```
class BusDriverImpl(BusDriver):
    def __init__(self, bus_route: BusRoute, rumors: set[Rumor]):
        self.__bus_route = bus_route
        self.__current_bus_stop = bus_route.get_first_bus_stop()
        self.__current_bus_stop.add(self)
        self.__rumors = rumors.copy()

    def drive_to_next_bus_stop(self) -> BusStop:
       self.__current_bus_stop.remove(self)
```

```
        self.__current_bus_stop =
self.__bus_route.get_next_bus_stop(
            self.__current_bus_stop
        )

        self.__current_bus_stop.add(self)
        return self.__current_bus_stop

    def get_current_bus_stop(self) -> BusStop:
        return self.__current_bus_stop

    # ...
```

Finally, we should implement the `GossipingBusDrivers` class and its `drive_until_all_rumors_shared`. Let's write a unit test for the first scenario when all rumors are shared after driving from one bus stop to the next. The `drive_until_all_rumors_shared` method makes drivers drive to the next bus stop (the same for both drivers) and share rumors there.

Figure 6.96

```
class GossipingBusDrivers(unittest.TestCase):
    rumor1 = Rumor()
    rumor2 = Rumor()
    all_rumors = {rumor1, rumor2}

    @patch('BusStopImpl.__new__')
    @patch('BusDriverImpl.__new__')
    @patch('BusDriverImpl.__new__')
    def test_drive_until_all_rumors_shared__after_one_stop(
        self,
        bus_driver_mock1: Mock,
        bus_driver_mock2: Mock,
        bus_stop_mock: Mock,
    ):
        # GIVEN
        bus_driver_mock1.drive_to_next_bus_stop.return_value = (
            bus_stop_mock
        )

        bus_driver_mock2.drive_to_next_bus_stop.return_value = (
            bus_stop_mock
        )

        bus_driver_mock1.get_rumors.return_value =
self.all_rumors
        bus_driver_mock2.get_rumors.return_value =
self.all_rumors

        gossiping_bus_drivers = GossipingBusDrivers(
            [bus_driver_mock1, bus_driver_mock2]
        )

        # WHEN
        all_rumors_were_shared = (
```

```
gossiping_bus_drivers.drive_until_all_rumors_shared(100)
        )

        # THEN
        self.assertTrue(all_rumors_were_shared)

bus_stop_mock.share_rumors_with_drivers.assert_called_once()
```

Let's make the above test pass with the below code.

Figure 6.97

```
class GossipingBusDrivers:
    def __init__(self, bus_drivers: list[BusDriver]):
        self.__bus_drivers = bus_drivers.copy()
        self.__all_rumors = self.__get_all_rumors()

    def drive_until_all_rumors_shared(self) -> bool:
        while True:
            for bus_driver in self.__bus_drivers:
                bus_stop = bus_driver.drive_to_next_bus_stop()
                bus_stop.share_rumors_with_drivers()

            if self.__all_rumors_are_shared():
                return True

    def __get_all_rumors():
        return {
            rumor
            for bus_driver in self.__bus_drivers
            for rumor in bus_driver.get_rumors()
        }

    def __all_rumors_are_shared(self) -> bool:
        return all(
            [
                bus_driver.get_rumors() == self.__all_rumors
                for bus_driver in self.__bus_drivers
            ]
        )
```

Let's add a test for a scenario where two bus drivers drive from their starting bus stops to two different bus stops and must continue driving because all rumors were not shared at the first bus stop. Rumors are shared when drivers continue driving to their next bus stop, which is the same for both drivers.

Figure 6.98

```
class GossipingBusDriversTests(unittest.TestCase):
    # ...

    @patch('BusStopImpl.__new__')
    @patch('BusStopImpl.__new__')
    @patch('BusStopImpl.__new__')
    @patch('BusDriverImpl.__new__')
    @patch('BusDriverImpl.__new__')
```

```
    def test_drive_until_all_rumors_shared__after_two_stops(
        self,
        bus_driver_mock1: Mock,
        bus_driver_mock2: Mock,
        bus_stop_mock1: Mock,
        bus_stop_mock2: Mock,
        bus_stop_mock3: Mock,
    ):
        # GIVEN
        bus_stop_mocks = [bus_stop_mock1, bus_stop_mock2,
bus_stop_mock3]

        bus_driver_mock1.drive_to_next_bus_stop.side_effect = [
            bus_stop_mock1,
            bus_stop_mock3,
        ]

        bus_driver_mock2.drive_to_next_bus_stop.side_effect = [
            bus_stop_mock2,
            bus_stop_mock3,
        ]

        bus_driver_mock1.get_rumors.side_effect = [
            {self.rumor1},
            {self.rumor1},
            self.all_rumors,
        ]

        bus_driver_mock2.get_rumors.side_effect = [
            {self.rumor2},
            {self.rumor2},
            self.all_rumors,
        ]

        gossiping_bus_drivers = GossipingBusDrivers(
            [bus_driver_mock1, bus_driver_mock2]
        )

        # WHEN
        all_rumors_were_shared = (

gossiping_bus_drivers.drive_until_all_rumors_shared(100)
        )

        # THEN
        self.assertTrue(all_rumors_were_shared)

        for bus_stop_mock in bus_stop_mocks:

bus_stop_mock.share_rumors_with_drivers.assert_called_once()
```

Let's modify the implementation:

Figure 6.99

```
class GossipingBusDrivers:
    # ...

    def drive_until_all_rumors_shared(self) -> bool:
        while True:
            bus_stops = {
```

```
                bus_driver.drive_to_next_bus_stop()
                for bus_driver in self.__bus_drivers
            }

            for bus_stop in bus_stops:
                bus_stop.share_rumors_with_drivers()

            if self.__all_rumors_are_shared():
                return True
    # ...
```

Next, we should implement a test where drivers don't have common bus stops and they have driven the maximum number of bus stops:

Figure 6.100

```
class GossipingBusDriversTests(unittest.TestCase):
    # ...

    @patch('BusStopImpl.__new__')
    @patch('BusStopImpl.__new__')
    @patch('BusDriverImpl.__new__')
    @patch('BusDriverImpl.__new__')
    def
test_drive_until_all_rumors_shared__when_rumors_are_not_shared(
        self,
        bus_driver_mock1: Mock,
        bus_driver_mock2: Mock,
        bus_stop_mock1: Mock,
        bus_stop_mock2: Mock,
    ):
        # GIVEN
        bus_stop_mocks = [bus_stop_mock1, bus_stop_mock2]

        bus_driver_mock1.drive_to_next_bus_stop.return_value = (
            bus_stop_mock1
        )

        bus_driver_mock2.drive_to_next_bus_stop.return_value = (
            bus_stop_mock2
        )

        bus_driver_mock1.get_rumors.return_value = {self.rumor1}
        bus_driver_mock2.get_rumors.return_value = {self.rumor2}

        gossiping_bus_drivers = GossipingBusDrivers(
            [bus_driver_mock1, bus_driver_mock2]
        )

        max_driven_stop_count = 2

        # WHEN
        all_rumors_were_shared = (
            gossiping_bus_drivers.drive_until_all_rumors_shared(
                max_driven_stop_count
            )
        )

        # THEN
```

```
        self.assertFalse(all_rumors_were_shared)

        for bus_stop_mock in bus_stop_mocks:
            self.assertEqual(

bus_stop_mock.share_rumors_with_drivers.call_count, 2
            )
```

Let' modify the implementation to make the above test to pass:

Figure 6.101

```
class GossipingBusDrivers:
    def __init__(self, bus_drivers: list[BusDriver]):
        self.__bus_drivers = bus_drivers.copy()
        self.__nbr_of_stops_driven = 0

    def drive_until_all_rumors_shared(
        self, max_driven_stop_count: int
    ) -> bool:
        while True:
            bus_stops = {
                bus_driver.drive_to_next_bus_stop()
                for bus_driver in self.__bus_drivers
            }

            self.__nbr_of_stops_driven += 1

            for bus_stop in bus_stops:
                bus_stop.share_rumors_with_drivers()

            if self.__all_rumors_are_shared():
                return True
            elif self.__nbr_of_stops_driven ==
max_driven_stop_count:
                return False
    # ...
```

Let's refactor the method slightly so that it is not too long (max 5-9 statements):

Figure 6.102

```
class GossipingBusDrivers:
    # ...

    def drive_until_all_rumors_shared(
        self, max_driven_stop_count: int
    ) -> bool:
        while True:
            bus_stops = {
                bus_driver.drive_to_next_bus_stop()
                for bus_driver in self.__bus_drivers
            }

            self.__driven_stop_count += 1
            self.__share_rumors(bus_stops)
```

```
            if self.__all_rumors_are_shared():
                return True
            elif self.__driven_stop_count ==
max_driven_stop_count:
                return False

    @staticmethod
    def __share_rumors(bus_stops: set[BusStop]):
        for bus_stop in bus_stops:
            bus_stop.share_rumors_with_drivers()

    # ...
```

Next, we shall implement the `MaxDrivenStopCountParser` and its 'parse' method. Let's create a failing test:

Figure 6.103

```
class MaxDrivenStopCountParserTests(unittest.TestCase):
    def test_parse__when_it_succeeds(self):
        # GIVEN
        max_driven_stop_count_as_str = '2'

        # WHEN
        max_driven_stop_count =
MaxDrivenStopCountParserImpl().parse(
            max_driven_stop_count_as_str
        )

        # THEN
        self.assertEqual(max_driven_stop_count, 2)
```

Now we can implement the class to make the above test to pass:

Figure 6.104

```
class MaxDrivenStopCountParser(Protocol):
    def parse(self, max_driven_stop_count_as_str: str) -> int:
        pass


class MaxDrivenStopCountParserImpl(MaxDrivenStopCountParser):
    def parse(self, max_driven_stop_count_as_str: str) -> int:
        return int(max_driven_stop_count_as_str)
```

Next, we specify that the `parse` method should throw a `ValueError` if the parsing fails:

Figure 6.105

```
class MaxDrivenStopCountParserTests(unittest.TestCase):
     def test_parse__when_it_fails(self):
         # GIVEN
         max_driven_stop_count_as_str = 'invalid'

         # WHEN + THEN
         self.assertRaises(
```

```
            ValueError,
            MaxDrivenStopCountParserImpl().parse,
            max_driven_stop_count_as_str,
        )
```

The above test will pass without modification to the implementation.

Next, we shall implement the `BusDriversParser` and its 'parse' method. We will skip the failure scenarios, like when a bus driver specification does not contain at least one bus stop and one rumor. Let's first create a failing test for a scenario where we have only one driver with one bus stop and one rumor:

Figure 6.106

```
class BusDriversParserImplTests(unittest.TestCase):
    def
test_parse__with_one_driver_that_has_one_bus_stop_and_one_rumor(
self):
        # GIVEN
        bus_driver_spec = 'bus-stop-a;rumor1'

        # WHEN
        bus_drivers =
BusDriversParserImpl().parse([bus_driver_spec])

        # THEN

self.__assert_has_circular_bus_route_with_one_stop(bus_drivers)
        self.assertEqual(len(bus_drivers[0].get_rumors()), 1)

    def __assert_has_circular_bus_route_with_one_stop(self,
bus_drivers):
        self.assertEquals(len(bus_drivers), 1)
        bus_stop = bus_drivers[0].get_current_bus_stop()
        next_bus_stop = bus_drivers[0].drive_to_next_bus_stop()
        self.assertEqual(bus_stop, next_bus_stop)
```

Let's implement the `parse` method to make the above test pass:

Figure 6.107

```
class BusDriversParser(Protocol):
    def parse(self, bus_driver_specs: list[str]) ->
list[BusDriver]:
        pass


class BusDriversParserImpl(Protocol):
    def parse(self, bus_driver_specs: list[str]) ->
list[BusDriver]:
        return [
            self.__get_bus_driver(bus_driver_spec)
```

```
            for bus_driver_spec in bus_driver_specs
        ]

    def __get_bus_driver(self, bus_driver_spec: str) ->
BusDriver:
        return BusDriverImpl(CircularBusRoute([BusStopImpl()]),
{Rumor()})
```

Let's create a test for a scenario where we have multiple drivers with one bus stop and one rumor each; both the bus stops and rumors for drivers are different:

Figure 6.108

```
class BusDriversParserImplTests(unittest.TestCase):
    def
test_parse__with_multiple_drivers_with_different_bus_stop_and_ru
mor(self):
        # GIVEN
        bus_driver_spec = ['bus-stop-a;rumor1', 'bus-stop-
b;rumor2']

        # WHEN
        bus_drivers =
BusDriversParserImpl().parse(bus_driver_spec)

        # THEN
        self.__assert_bus_stops_are_not_same(bus_drivers)

        self.assertNotEqual(
            bus_drivers[0].get_rumors(),
bus_drivers[1].get_rumors()
        )

    def __assert_bus_stops_are_not_same(self, bus_drivers):
        self.assertEqual(len(bus_drivers), 2)
        driver1_stop1 = bus_drivers[0].get_current_bus_stop()
        driver2_stop1 = bus_drivers[1].get_current_bus_stop()
        self.assertNotEqual(driver1_stop1, driver2_stop1)
```

The above test will pass without modifications to the implementation.

Let's create a test for a scenario where we have multiple drivers with one bus stop and one rumor each; the bus stops are the same, but rumors are different:

Figure 6.109

```
class BusDriversParserImplTests(unittest.TestCase):
    def
test_parse__with_multiple_drivers_with_a_common_bus_stop(self):
        # GIVEN
        bus_driver_spec = ["bus-stop-a;rumor1", "bus-stop-
a;rumor2"]

        # WHEN
```

```
        bus_drivers =
BusDriversParserImpl().parse([bus_driver_spec])

        # THEN
        self.__assert_bus_stops_are_same(bus_drivers)

    def assert_bus_stop_are_same(self, bus_drivers):
        driver1_stop = bus_driver[0].get_current_bus_stop()
        driver2_stop = bus_driver[1].get_current_bus_stop()
        self.assertEqual(driver1_stop, driver2_stop)
```

Let's modify the implementation to make the above test pass:

Figure 6.110

```
class BusDriversParserImpl(Protocol):
    def __init__(self):
        self.__name_to_bus_stop = {}

    def parse(self, bus_driver_specs: list[str]) ->
list[BusDriver]:
        return [
            self.__get_bus_driver(bus_driver_spec)
            for bus_driver_spec in bus_driver_specs
        ]

    def __get_bus_driver(self, bus_driver_spec: str) ->
BusDriver:
        bus_stop_name, _ = bus_driver_spec.split(';')

        if self.__name_to_bus_stop.get(bus_stop_name) is None:
            self.__name_to_bus_stop[bus_stop_name] =
BusStopImpl()

        return BusDriverImpl(

CircularBusRoute([self.__name_to_bus_stop[bus_stop_name]]),
            {Rumor()},
        )
```

Let's create a test for a scenario where we have multiple drivers with one rumor each. The rumors are the same:

Figure 6.111

```
class BusDriversParserImplTests(unittest.TestCase):
    def test_parse__with_multiple_drivers_and_a_common_rumor(
        self,
    ):
        # GIVEN
        bus_driver_spec = ['bus-stop-a;rumor1', 'bus-stop-
b;rumor1']

        # WHEN
        bus_drivers =
BusDriversParserImpl().parse(bus_driver_spec)

        # THEN
        self.assertEqual(
            bus_drivers[0].get_rumors(),
```

```
bus_drivers[1].get_rumors()
        )
```

Let's modify the implementation to make the test pass:

Figure 6.112

```
class BusDriversParserImpl(Protocol):
    def __init__(self):
        self.__name_to_bus_stop = {}
        self.__name_to_rumor = {}

    def parse(self, bus_driver_specs: list[str]) ->
list[BusDriver]:
        return [
            self.__get_bus_driver(bus_driver_spec)
            for bus_driver_spec in bus_driver_specs
        ]

    def __get_bus_driver(self, bus_driver_spec: str) ->
BusDriver:
        bus_stop_name, rumor_name = bus_driver_spec.split(';')

        if self.__name_to_bus_stop.get(bus_stop_name) is None:
            self.__name_to_bus_stop[bus_stop_name] =
BusStopImpl()

        if self.__name_to_rumor.get(rumor_name) is None:
            self.__name_to_rumor[rumor_name] = Rumor()

        return BusDriverImpl(

CircularBusRoute([self.__name_to_bus_stop[bus_stop_name]]),
            {self.__name_to_rumor[rumor_name]},
        )
```

Let's create a test for a scenario where we have multiple drivers with multiple bus stops (the first bus stop is the same):

Figure 6.113

```
class BusDriversParserImplTests(unittest.TestCase):
    def
test_parse__with_multiple_drivers_and_multiple_bus_stops_where_f
irst_is_common(se\
lf):
        # GIVEN
        bus_driver_spec = [
            'bus-stop-a,bus-stop-b;rumor1',
            'bus-stop-a,bus-stop-c;rumor2',
        ]

        # WHEN
        bus_drivers =
BusDriversParserImpl().parse(bus_driver_spec)

        # THEN
        self.__assert_only_first_bus_stops_are_same(bus_drivers)
```

```
    def assert_only_first_bus_stops_are_same(self, bus_drivers):
        driver1_stop1 = bus_drivers[0].get_current_bus_stop()
        driver2_stop1 = bus_drivers[1].get_current_bus_stop()
        self.assertEqual(driver1_stop1, driver2_stop1)

        driver1_stop2 = bus_drivers[0].drive_to_next_bus_stop()
        driver2_stop2 = bus_drivers[1].drive_to_next_bus_stop()
        self.assertNotEqual(driver1_stop2, driver2_stop2)
```

Let's make the above test pass:

Figure 6.114

```
class BusDriversParserImpl(Protocol):
    # ...

    def parse(self, bus_driver_specs: list[str]) ->
list[BusDriver]:
        return [
            self.__get_bus_driver(bus_driver_spec)
            for bus_driver_spec in bus_driver_specs
        ]

    def __get_bus_driver(self, bus_driver_spec: str) ->
BusDriver:
        bus_route_spec, rumor_name = bus_driver_spec.split(';')
        bus_stop_names = bus_route_spec.split(',')

        for bus_stop_name in bus_stop_names:
            if self.__name_to_bus_stop.get(bus_stop_name) is
None:
                self.__name_to_bus_stop[bus_stop_name] =
BusStopImpl()

        bus_stops = [
            self.__name_to_bus_stop[bus_stop_name]
            for bus_stop_name in bus_stop_names
        ]

        if self.__name_to_rumor.get(rumor_name) is None:
            self.__name_to_rumor[rumor_name] = Rumor()

        return BusDriverImpl(
            CircularBusRoute(bus_stops),
{self.__name_to_rumor[rumor_name]}
        )
```

Let's create a test for a scenario where we have multiple drivers with multiple rumors (one of which is the same):

Figure 6.115

```
class BusDriversParserImplTests(unittest.TestCase):
    def
test_parse__with_multiple_drivers_and_multiple_rumors(self):
        # GIVEN
        bus_driver_specs = [
            'bus-stop-a;rumor1,rumor2,rumor3',
            'bus-stop-b;rumor1,rumor3',
        ]
```

```
        # WHEN
        bus_drivers =
BusDriversParserImpl().parse(bus_driver_specs)

        # THEN
        self.__assert_rumors_differ_by_one(bus_drivers)

    def __assert_rumors_differ_by_one(self, bus_drivers):
        self.assertEqual(len(bus_drivers[0].get_rumors()), 3)
        self.assertEqual(len(bus_drivers[1].get_rumors()), 2)

        rumor_diff = (
            bus_drivers[0]
            .get_rumors()
            .difference(bus_drivers[1].get_rumors())
        )

        self.assertEqual(len(rumor_diff), 1)
```

Let's modify the implementation to make the above test pass:

Figure 6.116

```
class BusDriversParserImpl(Protocol):
    # ...

    def parse(self, bus_driver_specs: list[str]) ->
list[BusDriver]:
        return [
            self.__get_bus_driver(bus_driver_spec)
            for bus_driver_spec in bus_driver_specs
        ]

    def __get_bus_driver(self, bus_driver_spec: str) ->
BusDriver:
        bus_route_spec, rumors_spec = bus_driver_spec.split(';')
        bus_stop_names = bus_route_spec.split(',')

        for bus_stop_name in bus_stop_names:
            if self.__name_to_bus_stop.get(bus_stop_name) is
None:
                self.__name_to_bus_stop[bus_stop_name] =
BusStopImpl()

        bus_stops = [
            self.__name_to_bus_stop[bus_stop_name]
            for bus_stop_name in bus_stop_names
        ]

        rumor_names = rumors_spec.split(',')

        for rumor_name in rumor_names:
            if self.__name_to_rumor.get(rumor_name) is None:
                self.__name_to_rumor[rumor_name] = Rumor()

        rumors = {
            self.name_to_rumor[rumor_name] for rumor_name in
rumor_names
        }
```

```
        return BusDriverImpl(CircularBusRoute(bus_stops),
rumors)
```

Now that we have implemented all tests, we can refactor the `parse` to the following:

Figure 6.117

```
class BusDriversParserImpl(Protocol):
    def __init__(self):
        self.__name_to_bus_stop = {}
        self.__name_to_rumor = {}

    def parse(self, bus_driver_specs: list[str]) ->
list[BusDriver]:
        return [
            self.__get_bus_driver(bus_driver_spec)
            for bus_driver_spec in bus_driver_specs
        ]

    def __get_bus_driver(self, bus_driver_spec: str) ->
BusDriver:
        bus_route_spec, rumors_spec = bus_driver_spec.split(';')
        bus_stop_names = bus_route_spec.split(',')
        bus_stops = self.__get_bus_stops(bus_stop_names)
        rumor_names = rumors_spec.split(',')
        rumors = self.__get_rumors(rumor_names)
        return BusDriverImpl(CircularBusRoute(bus_stops),
rumors)

    def __get_bus_stops(self, bus_stop_names: list[str]) ->
list[BusStop]:
        for name in bus_stop_names:
            if self.__name_to_bus_stop.get(name) is None:
                self.__name_to_bus_stop[name] = BusStopImpl()

        return [self.__name_to_bus_stop[name] for name in
bus_stop_names]

    def __get_rumors(self, rumor_names: list[str]) ->
set[Rumor]:
        for name in rumor_names:
            if self.__name_to_rumor.get(name) is None:
                self.__name_to_rumor[name] = Rumor()

        return {self.__name_to_rumor[name] for name in
rumor_names}
```

Finally we need to implement the *main.py*:

Figure 6.118

```
class Main:
    @staticmethod
    def run():
        max_driven_stop_count =
MaxDrivenStopCountParserImpl().parse(
            sys.argv[1]
        )
```

```
        bus_drivers = BusDriversParserImpl().parse(sys.argv[2:])

        all_rumors_were_shared = GossipingBusDrivers(
            bus_drivers
        ).drive_until_all_rumors_shared(max_driven_stop_count)

        code = 0 if all_rumors_were_shared else 1
        sys.exit(code)


if __name__ == '__main__':
    Main().run()
```

The program can be run from the command line, for example:

Figure 6.119

```
python main.py 100 "stop-a;rumor-1" "stop-a;rumor2"
```

Let's say that product management wants a new feature and puts the following user story on the backlog:

- Support a back-and-forth bus route in addition to the circular bus route.

Because we used the *program against interfaces principle* earlier, we can implement this new feature using the *open-closed principle* by implementing a new `BackNForthBusRoute` class that implements the `BusRoute` interface. Similarly, we could later introduce other new features using the open-closed principle:

- Quick bus stops where drivers don't have time to share rumors could be implemented in a new `QuickBusStop` class implementing the `BusStop` interface
- Forgetful bus drivers that remember others' rumors only, e.g., for a certain number of bus stops, could be implemented with a new `ForgetfulBusDriver` class that implements the `BusDriver` interface

Let's consider implementing a simple API containing CRUD operations. Product management has defined the following feature (or epic) in the backlog:

> *Sales item API creates, reads, updates, and deletes sales items. Sales items are stored in a persistent storage. Each sales item consists of the following attributes: unique ID, name, and price. Sales items can be created, updated, and deleted only by administrators.*

First, the architecture team should provide technical guidance on implementing the backlog feature (or epic). The guidance could be the following: API should be a REST API, MySQL database should be used as persistent storage, and Keycloak should be used as the IAM system. Next, the development team should perform threat modeling (facilitated by the product security lead if needed). Threat modelling should result in additional security-related user stories that should be added to the backlog feature (or epic) and implemented by the team.

> Consider implementing the above-specified sales item API in a real-life scenario as two separate APIs for improved security: One public internet-facing API for reading sales items and another private API for administrator-related operations. The private admin API should not be accessible from the public internet directly, but access from the internet should require a VPN connection (and proper authorization, of course), for example.

The development team should split the backlog feature into user stories in the PI planning (or before it in the IP iteration). The team will come up with the following user stories:

1. As an admin user, I want to create a new sales item in a persistent storage. A sales item contains the following attributes: id, name, and price
2. As a user, I want to read sales items from the persistent storage
3. As an admin user, I want to update a sales item in the persistent storage
4. As an admin user, I want to delete a sales item from the persistent storage

Next, the team should continue by applying BDD to each user story:

User story 1:

Figure 6.120. create_sales_item.feature

```
Feature: Create sales item

  Background: Database is available and a clean table exists
    Given database is available
    And table is empty
    And auto increment is reset


  Scenario: Successfully create sales item as admin user
    When the following sales items are created as admin user
       | name         | price |
       | Sales item 1 | 10    |
       | Sales item 2 | 20    |
       | Sales item 3 | 30    |

    Then a response with status code 201 is received
    And the following sales items are received as response
      | name         | price | Id |
      | Sales item 1 | 10    | 1  |
      | Sales item 2 | 20    | 2  |
      | Sales item 3 | 30    | 3  |


   Scenario: Try to create a sales item with invalid data
      When the following sales items are created as admin user
        | name         | price |
        | Sales item 1 | aa    |

      Then a response with status code 400 is received


    Scenario: Try to create new sales when database is
unavailable
      Given database is unavailable

      When the following sales items are created as admin user
        | name         | price |
        | Sales item 1 | 10    |

      Then a response with status code 503 is received


    Scenario: Try to create a sales item as normal user
      When the following sales items are created as normal user
        | name         | price |
        | Sales item 1 | 10    |

      Then a response with status code 403 is received


    Scenario: Try to create a sales item unauthenticated
      When the following sales items are created unauthenticated
        | aame         | price |
        | Sales item 1 | aa    |
```

```
      Then a response with status code 401 is received
```

User story 2:

Figure 6.121. read_sales_item.feature

```
Feature: Read sales items

  Background: Database is available and a clean table exists
    Given database is available
    And table is empty
    And auto increment is reset


  Scenario: Successfully read sales items
    Given the following sales items are created
      | name         | price |
      | Sales item 1 | 10    |
      | Sales item 2 | 20    |
      | Sales item 3 | 30    |

    When sales items are read

    Then a response with status code 200 is received
    And the following sales items are received as response
      | name         | price | Id |
      | Sales item 1 | 10    | 1  |
      | Sales item 2 | 20    | 2  |
      | Sales item 3 | 30    | 3  |


  Scenario: Try to read sales items when database is unavailable
    Given database is unavailable
    When sales items are read
    Then a response with status code 503 is received
```

User story 3:

Figure 6.122. update_sales_item.feature

```
Feature: Update sales item

  Background: Database is available and a clean table exists
    Given database is available
    And table is empty
    And auto increment is reset


  Scenario: Successfully update sales item as admin user
    Given the following sales items are created as admin user
      | name         | price |
      | Sales item 1 | 10    |

    When the created sales item is updated to the following as
admin user
       | name         | price |
       | Sales item X | 100   |

    Then reading the sales item should provide the following
```

```
response
      | name         | price |
      | Sales item X | 100   |


  Scenario: Try to update sales item with invalid data
    When sales item with id 1 is updated to the following
      | name         | price |
      | Sales item X | aa    |

    Then a response with status code 400 is received


  Scenario: Sales item update fails because sales item is not
found
    When sales item with id 999 is updated to the following
      | name         | price |
      | Sales item X | 100   |

    Then a response with status code 404 is received


  Scenario: Try to update sales item when database is
unavailable
    Given database is unavailable

    When sales item with id 1 is updated to the following
      | name         | price  |
      | Sales item X | 100    |

    Then a response with status code 503 is received


  Scenario: Try to update sales as normal user
    When sales item with id 1 is updated to the following as
normal user
      | name         | price |
      | Sales item X | aa    |

    Then a response with status code 403 is received


  Scenario: Try to update sales unauthenticated
    When sales item with id 1 is updated to the following
unauthenticated
      | name         | price |
      | Sales item X | aa    |

    Then a response with status code 401 is received
```

User story 4:

Figure 6.123. delete_sales_item.feature

```
Feature: Delete sales item

  Background: Database is available and clean table exists
    Given database is available
    And table is empty
    And auto increment is reset
```

```
  Scenario: Successfully delete sales item as admin user
    Given the following sales items are created as admin user
      | name         | price |
      | Sales item 1 | 10    |

    When the created sales item is deleted as admin user

    Then a response with status code 204 is received
    And reading sales items should provide an empty array as
response


  Scenario: Try to delete a non-existent sales with
    When sales item with id 9999 is deleted as admin user
    Then a response with status code 204 is received


  Scenario: Try to delete a sales item when database is
unavailable
    Given database is unavailable
    When sales item with id 1 is deleted as admin user
    Then a response with status code 503 is received


  Scenario: Try to delete a sales as normal user
    When sales item with id 1 is deleted as normal user
    Then a response with status code 403 is received


  Scenario: Try to delete a sales unauthenticated
    When sales item with id 1 is deleted unauthenticated
    Then a response with status code 401 is received
```

In the above features, we have considered the main failure scenarios in addition to the happy path scenario. Remember that you should also test the most common failure scenarios as part of integration testing.

All of the above features include a `Background` section defining steps executed before each scenario. In the `Background` section, we first ensure the database is available. This is needed because some scenarios make the database unavailable on purpose. Then, we clean the sales items table in the database. This can be done by connecting to the database and executing SQL statements like `DELETE FROM sales_items` and `ALTER TABLE sales_items AUTO_INCREMENT=1`. Here, I am assuming a MySQL database is used. The database availability can be toggled by issuing `docker pause` and `docker unpause` commands for the database server container using `subprocess.run` or `subprocess.Popen`. You should put the step implementations for the `Background` section into a

common `background_steps.py` file because the same `Background` section is used in all features. Before being able to execute integration tests, a *docker-compose.yml* file must be created. The file should define the microservice itself and its dependencies, the database (MySQL), and the IAM system (Keycloak).

Additionally, we must configure the IAM system before executing the integration tests. This can be done in the *environment.py* file in the `before_all` function. This function is executed by the `behave` command before executing the integration tests. Let's assume we are using Keycloak as the IAM system. To configure Keycloak, we can use the Keycloak's Admin REST API. We need to perform the following (Change the hardcoded version number in the URLs to the newest):

- Create a client (https://www.keycloak.org/docs-api/23.0.2/rest-api/index.html#_clients)
- Create an admin role for the client (https://www.keycloak.org/docs-api/23.0.2/rest-api/index.html#_roles)
- Create an admin user with the admin role (https://www.keycloak.org/docs-api/23.0.2/rest-api/index.html#_users)
- Create a regular user without the admin role

An alternative method to test authorization is to use a configurable mock OAuth2 server instead of a real IAM system.

Let's implement the steps for the first scenario of the first user story (creating a sales item):

Figure 6.124. create_sales_item_steps.py

```
import requests
from behave import then, when
from behave.runner import Context
from environment import SALES_ITEM_API_URL


@when('the following sales items are created {user_type}')
def step_impl(context: Context, user_type: str):
    # Obtain an 'access token' for an admin or normal user based
```

```
on 'user_type'
    # from the IAM system using OAuth2 Resource Owner Password
Flow
    # https://auth0.com/docs/get-started/authentication-and-
authorization-flow/resource-o\
wner-password-flow
    # Using this flow, you exchange client_id, client_secret,
    # username and password for an access token

    auth_header = (
        None
        if user_type == 'unauthenticated'
        else f'Bearer {access_token}'
    )

    context.received_sales_items = []
    context.status_codes = []

    for row in context.table:
        sales_item = {key: row[key] for key in row.keys()}

        response = requests.post(
            SALES_ITEM_API_URL,
            sales_item,
            headers={'Authorization': auth_header},
        )

        context.status_codes.append(response.status_code)
        received_sales_item = response.json()
        context.received_sales_items.append(received_sales_item)


@then('a response with status code {expected_status_code:d} is
received')
def step_impl(context: Context, expected_status_code: int):
    for received_status_code in context.status_codes:
        assert received_status_code == expected_status_code


@then('the following sales items are received as response')
def step_impl(context: Context):
    recv_and_expected_sales_items = zip(
        context.received_sales_items,
        context.table,
    )

    for (
        recv_sales_item,
        expected_sales_item,
    ) in recv_and_expected_sales_items:
        for key in expected_sales_item.keys():
            assert recv_sales_item[key] ==
expected_sales_item[key]
```

To make the rest of the scenarios for the first feature pass, we need to add the following:

Figure 6.125. common_steps.py

```
@given('database is unavailable')
def step_impl(context: Context):
    # Use subprocess.run or subprocess.Popen to execute
    # 'docker pause' command for the
    # database container
```

We will skip implementing the rest of the steps because they are similar to those we implemented above. The main difference is using different methods of the `request` library: `get` for reading, `put` for updating and `delete` for deleting.

Next, the development team should perform DDD for the user stories. The first user story is comprised of the following domain events:

- user is authorized
- sales item is validated
- sales item is created
- sales item is persisted

From the above domain events, we can infer the following:

- Actor: UserAuthorizer, Command: authorize
- Actor: InputSalesItemValidator, Command: validate, Entity: InputSalesItem
- Actor: CreateSalesItemService: Command: create_sales_item, Entities: InputSalesItem, OutputSalesItem
- Actor: CreateSalesItemRepository, Command: save, Entity: SalesItem

Let's conduct DDD for the second user story. We will have the following domain events:

- sales items are read from the persistent store
- sales items are converted to output format

From the above domain events, we can infer the following:

- Actor: ReadSalesItemsRepository, Command: read_all, Entity: SalesItem
- Actor: ReadSalesItemsService: Command: read_sales_items, Entities: OutputSalesItem

The team decides to use the *clean microservice design principle*. Thus, interfaces and controller classes should be added for both user stories. The source code directory should look like the following:

Figure 6.126

```
sales-item-service
└── src
    └── sales-items
        ├── common
        │   ├── dtos
        │   │   ├── InputSalesItem.py
        │   │   └── OutputSalesItem.py
        │   ├── entities
        │   │   └── SalesItem.py
        │   └── validator
        │       └── InputSalesItemValidator.py
        ├── create
        │   ├── CreateSalesItemRepository.py
        │   ├── CreateSalesItemService.py
        │   ├── CreateSalesItemServiceImpl.py
        │   ├── RestCreateSalesItemController.py
        │   └── SqlCreateSalesItemRepository.py
        └── read
            ├── ReadSalesItemsRepository.py
            ├── ReadSalesItemsService.py
            ├── ReadSalesItemsServiceImpl.py
            ├── RestReadSalesItemsController.py
            └── SqlReadSalesItemsRepository.py
```

When we continue with DDD for the rest of the user stories, we should eventually have subdirectories for `update` and `delete` features like those for `create` and `read` features. What we ended up having is called *vertical slice design or architecture*, which we presented in the second chapter. A different team member can implement each user story, and each team member has their own subdirectory (a vertical slice) to work on to minimize merge conflicts. Things common to all features are put into the `common` subdirectory. The development continues with each team member conducting TDD for the public methods in the classes. This kind of parallel development provides better agility when the whole team (instead of just one team member) can focus on the same

feature (or epic) to complete it as fast as possible, and only after that proceed to the next feature (or epic) on the prioritized backlog. The implementation details of the classes are not shown here, but the coming chapters about API design and databases show details how to create a REST controller, DTOs, a service class, and repositories like ORM, SQL, and MongoDB.

As I mentioned earlier, Gherkin is not the only syntax, and Behave is not the only tool to conduct BDD. I want to give you an example where Robot Framework (RF) is used as an alternative to Gherkin and Behave. You don't have to completely say goodbye to the Gherkin syntax because the Robot framework also supports a Gherkin-style way to define test cases. The below example is for the first user story we defined earlier: creating a sales item. The example shows how to test that user story's first two scenarios (or *test cases* in RF vocabulary). The below *.robot* file resembles the Gherkin *.feature* file. Each test case in the *.robot* file contains a list of steps that are *keywords*, defined in the *.resource* files. Each keyword defines code (a list of functions) to execute. The functions are implemented in the *libraries*.

Figure 6.127. create_sales_item.robot

```
*** Settings ***
Documentation   Create sales item
Resource        database_setup.resource
Resource        create_sales_item.resource
Test Setup      Database is available and a clean table exists

*** Test Cases ***
Successfully create sales item as admin user
    When a sales item is created as admin user
salesitem1  ${10}
    Then a response with status code is received
${201}
    And the following sales item is received as response
salesitem1  ${10}

Try to create a sales item with invalid data
    When a sales item is created as admin user
salesitem   aa
    Then a response with status code is received
${400}
```

Figure 6.128. database_setup.resource

```
*** Settings ***
Documentation
Library         ./DatabaseSetup.py
```

```
*** Keywords ***
Database is available and a clean table exists
    Start database if needed
    Clear table
    Reset auto increment
```

Figure 6.129. create_sales_item.resource

```
*** Settings ***
Documentation
Library         ./CreateSalesItem.py

*** Keywords ***
Database is available and a clean table exists
    Start database if needed
    Clear table
    Reset auto increment

A sales item is created as admin user
    [Arguments]    ${name}    ${price}
    Create sales item as admin user     ${name}     ${price}

A response with status code is received
    [Arguments]    ${status_code}
    Verify response     ${status code}

The following sales item is received as response
    [Arguments]    ${name}    ${price}
    Verify received sales item  ${name}     ${price}
```

Figure 6.130. DatabaseSetup.py

```
class DatabaseSetup:
    def __init__(self):
        pass

    def start_database_if_needed(self):
        # ...

    def clear_table(self):
        # ...

    def reset_auto_increment(self):
        # ...
```

Figure 6.131. CreateSalesItem.py

```
import requests
from environment import SALES_ITEM_API_URL


class CreateSalesItem:
    def __init__(self):
        self.response = None

    def create_sales_item_as_admin_user(self, name: str, price:
str | int):
        # Obtain admin user access token from the IAM system
        access_token = ''
        sales_item = {'name': name, 'price': price}
```

```
        self.response = requests.post(
            SALES_ITEM_API_URL,
            sales_item,
            headers={'Authorization': f'Bearer {access_token}'},
        )

    def verify_response(self, status_code: int):
        assert self.response.status_code == status_code

    def verify_received_sales_item(self, name: str, price: int):
        sales_item = self.response.json()
        assert sales_item['name'] == name
        assert sales_item['price'] == price
```

6.1.4: End-to-End (E2E) Testing Principle

End-to-end (E2E) testing should test a complete software system (i.e., the integration of microservices) so that each test case is end-to-end (from the software system's south-bound interface to the software system's north-bound interface).

As the name says, in E2E testing, test cases should be end-to-end. They should test that each microservice is deployed correctly to the test environment and connected to its dependent services. The idea of E2E test cases is not to test details of microservices' functionality because that has already been tested in unit and software component integration testing. This is why there should be only a handful of E2E test cases.

Let's consider a telecom network analytics software system that consists of the following applications:

- Data ingestion
- Data correlation
- Data aggregation
- Data exporter
- Data visualization

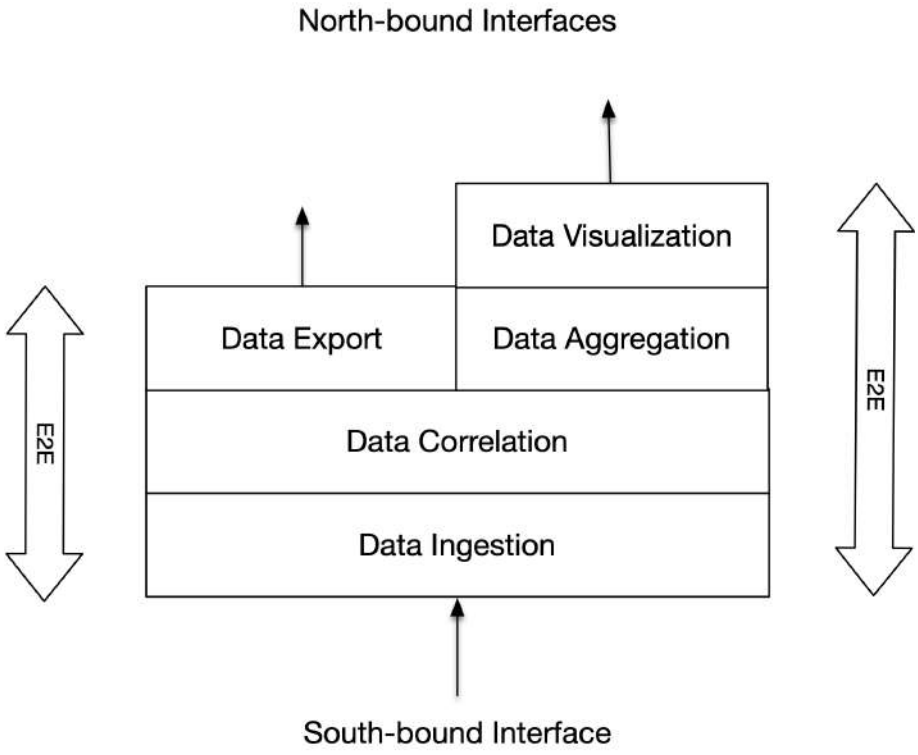

Figure 6.132. Telecom Network Analytics Software System

The southbound interface of the software system is the data ingestion application. The data visualization application provides a web client as a northbound interface. The data exporter application also provides another northbound interface for the software system.

E2E tests are designed and implemented similarly to software component integration tests. We are just integrating different things (microservices instead of functions). E2E testing starts with the specification of E2E features. These features can be specified using, for example, the Gherkin language and put in *.feature* files.

You can start specifying and implementing E2E tests right after the architectural design for the software system is completed. This way, you can shift the implementation of the E2E test to the left and speed up the development phase. You should not start specifying and implementing E2E only after the whole software system is implemented.

Our example software system should have at least two happy-path E2E features. One is for testing the data flow from data

ingestion to data visualization, and another feature is to test the data flow from data ingestion to data export. Below is the specification of the first E2E feature:

Figure 6.133

```
Feature: Visualize ingested, correlated and
         aggregated data in web UI's dashboard's charts

  Scenario: Data ingested, correlated and aggregated is
visualized
            successfully in web UI's dashboard's charts

    Given southbound interface simulator is configured
          to send input messages that contain data...
    And data ingester is configured to read the input messages
        from the southbound interface
    And data correlator is configured to correlate
        the input messages
    And data aggregator is configured to calculate
        the following counters...
    And data visualization is configured with a dashboard
containing
        the following charts viewing the following
counters/KPIs...

    When southbound interface simulator sends the input messages
    And data aggregation period is waited
    And data content of each data visualization web UI's
dasboard's
        chart is exported to a CSV file

    Then CSV export file of the first chart should
         contain the following values...
    And the CSV export file of the second chart should
         contain the following values...
    .
    .
    .
    And CSV export file of the last chart should
         contain the following values...
```

Then, we can create another feature that tests the E2E path from data ingestion to data export:

Figure 6.134

```
Feature: Export ingested, correlated and transformed data
         to Apache Pulsar

  Scenario: Data ingested, correlated and transformed is
            successfully exported to Apache Pulsar
    Given southbound interface simulator is configured to send
          input messages that contain data...
    And data ingester is configured to read the input messages
        from the southbound interface
    And data correlator is configured to correlate
       the input messages
    And data exporter is configured to export messages with
        the following transformations to Apache Pulsar...
```

```
When southbound interface simulator sends the input messages
And messages from Apache Pulsar are consumed

Then first message from Apache Pulsar should have
     the following fields with following values...
And second message from Apache Pulsar should have
    the following fields with following values...
.
.
.
And last message from Apache Pulsar should have
    the following fields with following values...
```

Next, E2E tests can be implemented. Any programming language and tool compatible with the Gherkin syntax, like Behave with Python, can be used. If the QA/test engineers in the development teams already use Behave for integration tests, it would be a natural choice to use Behave also for the E2E tests.

The software system we want to E2E test must reside in a production-like test environment. Usually, E2E testing is done in both the CI and the staging environment(s). Before running the E2E tests, the software needs to be deployed to the test environment.

If we consider the first feature above, implementing the E2E test steps can be done so that the steps in the `Given` part of the scenario are implemented using an externalized configuration. If our software system runs in a Kubernetes cluster, we can configure the microservices by creating the needed ConfigMaps. The southbound interface simulator can be controlled by launching a Kubernetes Job or, if it is a microservice with an API, commanding it via its API. After waiting for all the ingested data to be aggregated and visualized, the E2E test can launch a test tool suited for web UI testing (like TestCafe) to export chart data from the web UI to downloaded files. Then, the E2E test compares the content of those files with expected values.

You can run E2E tests in a CI environment after each commit to the main branch (i.e., after the microservice CI pipeline run has finished) to test that the new commit did not break any E2E tests. Alternatively, if the E2E tests are complex and take

a long time to execute, you can run the E2E tests in the CI environment on a schedule, like hourly, but at least nightly.

You can run E2E tests in a staging environment using a separate pipeline in your CI/CD tool.

6.2: Non-Functional Testing Principle

> ***In addition to multi-level functional testing, non-functional testing, as automated as possible, should be performed for a software system.***

The most important categories of non-functional testing are the following:

- Performance testing
- Data volume testing
- Stability testing
- Reliability testing
- Stress and scalability testing
- Security testing

6.2.1: Performance Testing

The goal of performance testing is to verify the performance of a software system. This verification can be done on different levels and in different ways, for example, by verifying each performance-critical microservice separately.

To measure the performance of a microservice, performance tests can be created to benchmark the busy loop or loops in the microservice. If we take the data exporter microservice as an example, there is a busy loop that performs message decoding, transformation, and encoding. We can create a performance test using a unit testing framework for this busy loop. The performance test should execute the code in the busy loop for a certain number of rounds (like 50,000 times) and verify that the execution duration does not exceed a specified threshold

value obtained on the first run of the performance test. The performance test aims to verify that performance remains at the same level as it has been or is better. If the performance has worsened, the test won't pass. In this way, you cannot accidentally introduce changes that negatively affect the performance without noticing it. This same performance test can also be used to measure the effects of optimizations. First, you write code for the busy loop without optimizations, measure the performance, and use that measure as a reference point. After that, you introduce optimizations individually and see if and how they affect the performance.

The performance test's execution time threshold value must be separately specified for each developer's computer. This can be achieved by having a different threshold value for each computer hostname running the test.

You can also run the performance test in a CI pipeline, but you must first measure the performance in that pipeline and set the threshold value accordingly. Also, the computing instances running CI pipelines must be homogeneous. Otherwise, you will get different results on different CI pipeline runs.

The above-described performance test was for a unit (one public function without mocking), but performance testing can also be done on the software component level. This is useful if the software component has external dependencies whose performance needs to be measured. In the telecom network analytics software system, we could introduce a performance test for the *data-ingester-service* to measure how long it takes to process a certain number of messages, like one million. After executing that test, we have a performance measurement available for reference. When we try to optimize the microservice, we can measure the performance of the optimized microservice and compare it to the reference value. If we make a change known to worsen the performance, we have a reference value to which we can compare the deteriorated performance and see if it is acceptable. And, of course, this reference value will prevent a developer from accidentally making a change that negatively impacts the microservice's performance.

You can also measure end-to-end performance. In the telecom network analytics software system, we could measure the performance from data ingestion to data export, for example.

6.2.2: Data Volume Testing

The goal of data volume testing is to measure the performance of a database by comparing an empty database to a database with a sizeable amount of data stored in it. With data volume testing, we can measure the impact of data volume on a software component's performance. Usually, an empty database has better performance than a database containing a high amount of data. This depends on the database and how it scales with large amounts of data.

6.2.3: Stability Testing

Stability testing aims to verify that a software system remains stable when running for an extended period of time under load. This testing is also called *load*, *endurance*, or *soak* testing. The term "extended period" can be interpreted differently depending on the software system. But this period should be many hours, preferably several days, even up to one week. Stability testing aims to discover problems like sporadic bugs or memory leaks. A sporadic bug is a bug that occurs only in certain conditions or at irregular intervals. A memory leak can be so small that the software component must run for tens of hours after it becomes clearly visible. It is recommended that when running the software system for a longer period, the induced load to the software system follows a natural pattern (mimicking the production load), meaning that there are peaks and lows in the load.

Stability testing can be partly automated. The load to the system can be generated using tools created for that purpose, like Apache JMeter, for example. Each software component can measure crash count, and those statistics can be analyzed automatically or manually after the stability testing is completed. Analyzing memory leaks can be trickier, but crashes due to out-of-memory and situations where a software

component is scaling out due to lack of memory should be registered.

6.2.4: Reliability Testing

Reliability testing aims to verify that a software system runs reliably. The software system is reliable when it is resilient to failures and recovers from failures automatically as fast as possible. Reliability testing is also called availability, *recovery*, or *resilience* testing.

Reliability testing involves chaos engineering to induce various failures in the software system's environment. It should also ensure the software system stays available and can automatically recover from failures.

Suppose you have a software system deployed to a Kubernetes cluster. You can make stateless services highly available by configuring them to run multiple pods. If one node goes down, it will terminate one of the pods (never allow scheduling all the microservice pods on the same node). However, the service remains available and usable because at least one other pod is still running on a different node. Also, after a short while, when Kubernetes notices that one pod is missing, it will create a new pod on a new node, and there will be the original number of pods running, and the recovery from the node down is successful.

Many parts of the reliability testing can be automated. You can use ready-made chaos engineering tools or create and use your tools. Use a tool to induce failures in the environment. Then verify based on the service's business-criticality that the service remains highly available or swiftly recovers from a failure.

Considering the telecom network analytics software system, we could introduce a test case where the message broker (e.g., Kafka) is shut down. Then, we expect alerts to be triggered after a while by the microservices that try to use the unavailable message broker. After the message broker is restarted, the alerts should cancel automatically, and the microservices should continue normal operation.

6.2.5: Stress and Scalability Testing

Stress testing aims to verify that a software system runs under high load. In stress testing, the software system is exposed to a load higher than the system's usual load. The software system should be designed as scalable, which means that the software system should also run under high load. Thus, stress testing should test the scalability of the software system and see that microservices scale out when needed. At the end of stress testing, the load is returned back to the normal level, and scaling in the microservices can also be verified.

You can specify a HorizontalPodAutoscaler (HPA) for a Kubernetes Deployment. In the HPA manifest, you must specify the minimum number of replicas. This should be at least two if you want to make your microservice highly available. You also need to specify the maximum number of replicas so that your microservice does not consume too many computing resources in some weird failure case. You can make the horizontal scaling (in and out) happen by specifying a target utilization rate for CPU and memory. Below is an example Helm chart template for defining a Kubernetes HPA:

Figure 6.135

```
{{- if eq .Values.env "production" }}
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: {{ include "microservice.fullname" . }}
  labels:
    {{- include "microservice.labels" . | nindent 4 }}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ include "microservice.fullname" . }}
  minReplicas: {{ .Values.hpa.minReplicas }}
  maxReplicas: {{ .Values.hpa.maxReplicas }}
  metrics:
    {{- if .Values.hpa.targetCPUUtilizationPercentage }}
    - type: Resource
      resource:
        name: cpu
        targetAverageUtilization: {{
.Values.hpa.targetCPUUtilizationPercentage }}
    {{- end }}
    {{- if .Values.hpa.targetMemoryUtilizationPercentage }}
    - type: Resource
      resource:
        name: memory
```

```
        targetAverageUtilization: {{
.Values.hpa.targetMemoryUtilizationPercentage }}
    {{- end }}
{{- end }}
```

It is also possible to specify the autoscaling to use an external metric. An external metric could be Kafka consumer lag, for instance. If the Kafka consumer lag grows too high, the HPA can scale the microservice out for more processing power for the Kafka consumer group. When the Kafka consumer lag decreases below a defined threshold, HPA can scale in the microservice to reduce the number of pods.

6.2.6: Security Testing

Security testing aims to verify that a software system is secure and does not contain security vulnerabilities. One part of security testing is performing vulnerability scans of the software artifacts. Typically, this means scanning the microservice containers using an automatic vulnerability scanning tool. Another essential part of security testing is penetration testing, which simulates attacks by a malicious party. Penetration testing can be performed using an automated tool like [ZAP] (https://www.zaproxy.org/) or Burp Suite.

Penetration testing tools try to find security vulnerabilities in the following categories:

- Cross-site scripting
- SQL injection
- Path disclosure
- Denial of service
- Code execution
- Memory corruption
- Cross-site request forgery (CSRF)
- Information disclosure
- Local/remote file inclusion

A complete list of possible security vulnerabilities found by the ZAP tool can be found at ZAP Alert Details.

6.2.7: Other Non-Functional Testing

Other non-functional testing is documentation testing and several UI-related non-functional testing, including accessibility (A11Y) testing, visual testing, usability testing, and localization and internationalization (I18N) testing.

6.2.7.1: Visual Testing

I want to bring up visual testing here because it is important. Backstop.js and cypress-plugin-snapshots test web UI's HTML and CSS using snapshot testing. Snapshots are screenshots taken of the web UI. Snapshots are compared to ensure that the visual look of the application stays the same and no bugs are introduced with HTML or CSS changes.

7: Security Principles

This chapter describes principles related to security and addresses the main security features from a software developer's point of view.

7.1: Shift Security to Left Principle

> ***Shift security implementation to the left. Implement security-related features sooner rather than later.***

Security is integral to production-quality software, like the source code itself and all the tests. Suppose that security-related features are implemented only in a very late project phase. In that case, there is a greater possibility of not finding time to implement them or forgetting to implement them. For that reason, security-related features should be implemented first rather than last. The threat modeling process described in the next section should be used to identify the potential threats and provide a list of security features that need to be implemented as threat countermeasures.

7.2: Have a Product Security Lead Principle

> ***Each product team should have a security lead appointed. The role of the security lead is to ensure that the product is secure.***

The security lead works tightly with development teams. They educate teams on security-related processes and security features. The security lead facilitates the teams in the below-described threat modeling process, but following the process is the team's responsibility, as is the actual implementation of security features.

7.3: Use Threat Modelling Process Principle

The threat modeling process enables you to identify, quantify, and address security risks associated with a software component or an application. The threat modeling process is composed of three high-level steps:

- Decompose the application
- Determine and rank threats
- Determine countermeasures and mitigation

7.3.1: Decompose Application

The application decomposition step is to gain knowledge of what parts the application is composed of, the external dependencies, and how they are used. This step can be performed after the application architecture is designed. The results of this step are:

- Identify an attacker's entry points to the application
- Identify assets under threat. These assets are something that an attacker is interested in
- Identify trust levels, e.g., what users with different user roles can do

7.3.2: Determine and Rank Threats

A threat categorization methodology should be used to determine possible threats. The *STRIDE* method categorizes threats to the following categories:

Category	Description
Spoofing	Attacker acting as another user without real authentication or using stolen credentials
Tampering	Attacker maliciously changing data
Repudiation	Attacker being able to perform prohibited operations
Information disclosure	Attacker gaining access to sensitive data
Denial of service	Attacker trying to make the service unusable
Elevation of privilege	Attacker gaining unwanted access rights

7.3.2.1: STRIDE Threat Examples

- Spoofing
  - The attacker can read other users' data using the other user's id when there is proper authorization missing
  - The attacker can steal user credentials on the network because insecure protocol, like HTTP instead of HTTPS, is used
  - The attacker creates a fake website login page to steal user credentials
  - The attacker can intercept network traffic and replay some user's requests as such or modified
- Tampering
  - The attacker gains access to the database using SQL injection and can change existing data
  - The attacker can modify other users' data using the other user's id when there is proper authorization missing
- Repudiation
  - The attacker can perform malicious action without notice when there is audit logging missing

- Information disclosure
    - Sensitive information is accidentally sent in request responses (like error stack traces or business confidential data)
    - Sensitive information is not adequately encrypted
    - Sensitive information is accessible without proper authorization (e.g., role-based)
- Denial of service
    - The attacker can create an unlimited number of requests when proper request rate limiting is missing
    - Attacker can send requests with large amounts of data when data size is not limited at all
    - Attacker can try to make regular expression DoS attacks by sending strings that can cause regular expression evaluation to take a lot of CPU time
    - The attacker can send invalid values in requests to try to crash the service or cause a forever loop if no proper input validation is in place
- Elevation of privilege An attacker who does not have a user account can access the service because of missing authentication/authorization
    - The attacker can act as an administrator because the service does not check that the user has a proper role
    - The attacker can access the operating system with root rights because the process runs with root user rights.

The *Application Security Frame* (ASF) categorizes application security features into the following categories:

Category	Description
Audit & Logging	Logging user actions to detect, e.g., repudiation attacks
Authentication	Prohibit identity spoofing attacks
Authorization	Prohibit elevation of privilege attacks
Configuration Management	Proper storage of secrets and configuring the system with the least privileges
Data Protection in Transit and Rest	Using secure protocols like TLS, encrypting sensitive information like PII in databases
Data Validation	Validate input data from users to prevent, e.g., injection and ReDoS attacks
Exception Management	Do not reveal implementation details in error messages to end-users

When using the above-described threat categorization methodologies, threats in each category should be listed based on the information about the decomposed application: what are the application entry points and assets that need to be secured? After listing potential threats in each category, the threats should be ranked. There are several ways to rank threats. The simplest way to rank threats is to put them in one of the three categories based on the risk: high, medium, and low. As a basis for the ranking, you can use information about the threat's probability and how big an adverse effect (impact) it has. The idea of ranking is to prioritize security features. Security features for high-risk threats should be implemented first.

7.3.3: Determine Countermeasures and Mitigation

The determining countermeasures step should list user stories for the needed security features. These security features should eliminate or at least mitigate the threats. If you have a threat that cannot be eliminated or mitigated, you can accept the risk if the threat is categorized as a low-risk threat. A low-risk threat has a low impact on the application, and the probability of the threat realization is low. Suppose you have found a

threat with a very high risk in your application, and you cannot eliminate or mitigate that threat. In that case, you should eliminate the threat by completely removing the threat-related features from the application.

7.3.4: Threat Modeling Example using STRIDE

Let's have a simple example of threat modeling in practice. We will perform threat modeling for a REST API microservice called *order-service*. The microservice handles orders (CRUD operations on order entities) in an e-commerce software system. Orders are persisted in a database. The microservice communicates with another microservice(s). The first step in the threat modeling process is to decompose the application.

7.3.4.1: Decompose Application

In this phase, we will decompose the *order-service* to see what parts it is composed of and what its dependencies are.

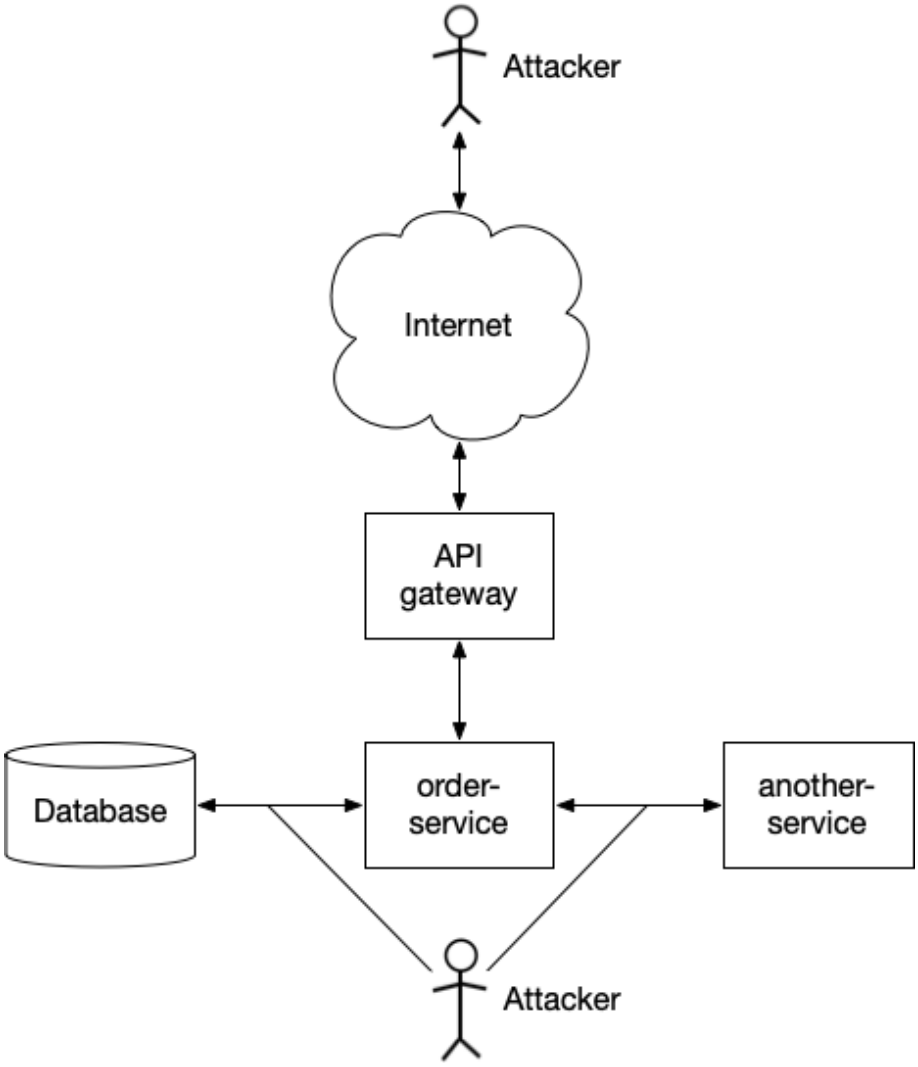

Figure 7.1. Decomposed Order Service

Based on the above view of the *order-service*, we shall next identify the following:

- Identify an attacker's entry points to the application
- Identify assets under threat. These assets are something that an attacker is interested in
- Identify trust levels, e.g., what users with different user roles can do

As drawn in the above picture, the attacker's entry points are from the internet (the *order-service* is exposed to the public internet via API Gateway), and an internal attacker could be able to sniff the network traffic between services.

Assets under threat are the API Gateway, *order-service*, its database, and unencrypted network traffic.

The *order-service* has the following trust levels:

- Users can place orders for themselves (not for other users)
- Users can view their orders (not other users)
- Users can update their order only before it is packaged and shipped
- Administrator can create/read/update/delete any order

7.3.4.2: Determine and Rank Threats

Next, we should list possible threats in each category of the STRIDE method. We also define the risk level for each possible threat.

1. Spoofing
    1. Attacker trying to create an order for someone else (Risk: High)
    2. Attacker trying to read/update someone else's order (Risk: High)
    3. Attacker acting as someone else using stolen credentials (Risk: Medium)
2. Tampering
    1. Attacker trying to tamper with database using SQL injection (Risk: High)
    2. Attacker able to capture and modify unencrypted internet traffic (Risk: High)
    3. Attacker able to capture and modify unencrypted internal network traffic (Risk: Low)
3. Repudiation
    1. Attacker being able to conduct malicious operations without getting caught (Risk: High)

4. Information disclosure
    1. The attacker can access sensitive information because it is not adequately encrypted (Risk: Medium)
    2. The attacker receives sensitive information like detailed stack traces in request responses. (Risk: Medium) The attacker can use that information and exploit possible security holes in the implementation.
    3. Information is disclosed to the attacker because internet traffic is plain text, i.e., not secured (Risk: High)
    4. Information is disclosed to the attacker because internal network traffic is plain text, i.e., not secured (Risk: Low)
5. Denial of service
    1. Attacker trying to make too many requests (Risk: High)
    2. Attacker trying to send requests with large amounts of data when data size is not limited at all (Risk: High)
    3. Attacker trying to make regular expression DoS (ReDos) attacks by sending strings that can cause regular expression evaluation to take a lot of CPU time (Risk: High)
    4. Attacker trying to send invalid values in requests to try to crash the service or cause a forever loop if no proper input validation is in place (Risk: High)
6. Elevation of Privilege
    1. An attacker who does not have a user account can access the service because of missing authentication/authorization (Risk: High)
    2. Attacker can act as an administrator because the service does not check that the user has a proper role (Risk: High)

3. The attacker can access the operating system with root rights because the process is running with root user rights (Risk: Medium)

7.3.5: Determine Countermeasures and Mitigation

Next, we shall define countermeasure user stories for each threat. The threat numbers a countermeasure is for are listed after the countermeasure.

1. Allow only the user that owns a particular resource to access it (1.1, 1.2)
2. Implement audit logging for operations that create/modify/delete orders (1.3, 3.1)
3. Use parameterized SQL statements or ORM and configure the least permissions for the database user (2.1). The normal database user should not be able to do anything that is only administrator-related, like deleting, creating/dropping tables, etc.
4. Only allow secure internet traffic to the API gateway (TLS is terminated at the API gateway) (1.3, 2.2)
5. Implement mTLS between services using a service mesh like Istio (2.3, 4.4)
6. Encrypt all sensitive information like Personally Identifiable Information (PII) and critical business data in the database (4.1)
7. Do not return error stack traces when the microservice is running in production (4.2)
8. Implement request rate-limiting in the API gateway (5.1.)
9. Validate input data to the microservice and define the maximum allowed string, array, and request lengths (5.2). Additionally, consider audit logging input validation failures
10. Do not use regular expressions in validation or use regular expressions that cannot cause ReDoS (5.3.)

11. Validate input data to the microservice, e.g., correct types, min/max of numeric values, and list of allowed values (5.4). Additionally, consider audit logging input validation failures
12. Implement user authentication and authorization using JWTs (1.1, 1.2, 6.1). Consider audit logging authentication/authorization failures to detect possible attacks
13. Verify that the JWT contains an admin role for administrator-only operations before allowing the operation (1.1, 1.2, 6.2). Additionally, configure the system so that admin operations are inaccessible from the internet unless needed.
14. For the containerized microservice, define the following:
    - Container should not be privileged
    - All capabilities are dropped
    - Container filesystem is read-only
    - Only a non-root user is allowed to run inside the container
    - Define the non-root user and group under which the container should run
    - Disallow privilege escalation
    - Use a distroless or the smallest possible container base image

Next, we should prioritize the above user stories according to related threat risk levels. Let's calculate a priority index for each user story using the following values for threat risk levels:

- High = 3
- Medium = 2
- Low = 1

Here are the prioritized user stories from the highest priority index (PI) to the lowest:

1. Implement user authentication and authorization using JWTs (PI: 9)
2. For administrator-only operations, verify that the JWT contains an admin role before allowing the operation (PI: 9). Additionally, configure the system so that admin operations are not accessible from the internet unless needed
3. Only allow secure internet traffic to the API gateway (TLS is terminated at the API gateway) (PI: 6)
4. Allow only the user that owns a specific resource to access it (PI: 6)
5. Implement audit logging for operations that create/modify/delete orders (PI: 5)
6. Implement request rate-limiting, e.g. in the API gateway (PI: 3)
7. Validate input data to the microservice and define maximum allowed string and array lengths (PI: 3)
8. Use parameterized SQL statements or ORM, and configure the least permissions for the database user (PI: 3). The normal database user should not be able to do anything that is only administrator-related, like deleting, creating/dropping tables, etc.
9. Do not use regular expression in validation or use regular expression that cannot cause ReDoS (PI: 3)
10. Validate input data to the microservice, e.g., min/max numeric values, list of allowed values (PI: 3)
11. Encrypt all sensitive information like Personally Identifiable Information (PII) and critical business data in the database (PI: 2)
12. Implement mTLS between services using a service mesh like Istio (PI: 2)

13. Do not return error stack traces when the microservice is running in production (PI: 2)
14. For the containerized microservice, define the following: … (PI: 2)

The team should review the prioritized list of security user stories with the product security lead. Because security is an integral part of a software system, at least all the above user stories having a priority index greater than two should be implemented before delivering the first production version. The user stories with PI <= 2 could be delivered immediately in the first feature package after the initial delivery. This is just an example. Everything depends on what level of security is wanted and required. The relevant stakeholders should be involved in making the decisions about the level of product security.

We did not list threats for missing security-related HTTP response headers in the above example. This is because they are the same for any REST API. These security-related HTTP response headers are discussed in a later section of this chapter. The sending of these headers should be consolidated to the API gateway so that all API microservices don't have to implement sending security headers themselves.

7.3.6: Threat Modeling Example Using ASF

Threat modeling using ASF goes the same way as in the previous example using the STRIDE method. The only difference is that the threats are categorized differently. We should be able to find all the same threats. Let's try to put the earlier found threats into ASF categories:

- Audit & Logging
    - Attacker being able to conduct malicious operations without getting caught (Risk: High)
    - Attacker acting as someone else using stolen credentials (Risk: Medium)

- Authentication An attacker who does not have a user account can access the service because of missing authentication/authorization (Risk: High)
- Authorization
    - The attacker can act as an administrator because the service does not check that the user has a proper role (Risk: High)
    - Attacker trying to create an order for someone else (Risk: High)
    - Attacker trying to read/update someone else's order (Risk: High)
- Configuration Management
    - The attacker can access the operating system with root rights because the process is running with root user rights (Risk: Medium)
- Data Protection in Transit and Rest
    - Attacker trying to tamper with database using SQL injection (Risk: High)
    - Attacker able to capture and modify unencrypted internet traffic (Risk: High)
    - Attacker able to capture and modify unencrypted internal network traffic (Risk: Low)
    - Attacker able to access sensitive information because it is not adequately encrypted (Risk: Medium)
    - Information is disclosed to the attacker because internet traffic is plain text, i.e., not secured (Risk: High)
    - Information is disclosed to the attacker because internal network traffic is plain text, i.e., not secured (Risk: Low)
- Data Validation

    - Attacker trying to make too many requests (Risk: High)
    - Attacker trying to send requests with large amounts of data when data size is not limited at all (Risk: High)
    - Attacker trying to tamper with database using SQL injection (Risk: High)
    - Attacker trying to make regular expression DoS (ReDos) attacks by sending strings that can cause regular expression evaluation
    - Attacker trying to send invalid values in requests to try to crash the service or cause a forever loop if no proper input validation is in place (Risk: High)
- Exception Management
    - The attacker receives sensitive information like detailed stack traces in request responses. (Risk: Medium)

You can even use two different threat categorization methods, like STRIDE and ASF, together because when using multiple methods, it is more likely to discover all the possible threats. Considering the ASF categorization, we can see that the Configuration Management category speaks about the storage of secrets. When we used STRIDE, we did not discover any secrets-related threats. But if we think about it, our *order-service* should have at least three secrets: database user name, database user password, and the encryption key used to encrypt sensitive data in the database. We must store these secrets safely, like using a Secret in a Kubernetes environment. None of these secrets should be hard-coded in the source code.

7.4: Security Features

This section focuses on security features that are relevant for software developers. It lists the most common security features that need to be implemented in typical software systems. It also provides some guidance on how to implement these

security features. For example, you should use a secure algorithm and encryption key when implementing encryption.

7.4.1: Authentication and Authorization

When implementing user authentication and authorization for an application, use a 3rd party authorization service. Don't try to build an authorization service by yourself. You can easily make mistakes. Also, it can be a security risk if your application handles plaint-text user credentials. It is better to use a battle-tested solution that has the most significant bugs corrected and can store user credentials securely. We use Keycloak as an authorization service in the coming examples.

Also, try using established 3rd party libraries as much as possible instead of writing all authorization-related code yourself. It is also helpful to create a single frontend authentication/authorization library and use that same library in multiple projects instead of constantly implementing authentication and authorization-related functionality from scratch in different projects.

7.4.1.1: OpenID Connect Authentication and Authorization in Frontend

Regarding frontend authorization, attention must be paid to the secure storage of authorization-related secrets like *code verifier* and *tokens*. Those must be stored in a secure location in the browser. Below is a list of some insecure storing mechanisms:

- Cookies
  - Sent automatically, a CSRF threat
- Session/Local Storage
  - Easily stolen by malicious code (XSS threat)
- Encrypted session/local storage
  - Easily stolen by malicious code because the encryption key is in plain text

- Global variable
    - Easily stolen by malicious code (XSS threat)

Storing secrets in closure variables is not inherently insecure, but secrets are lost on page refresh or new page.

Below is an example that uses a service worker to store secrets securely. The additional benefit of a service worker is that it does not allow malicious 3rd party code to modify the service worker's `fetch` method so that it can, for example, steal access tokens.

It is easy for malicious code to change the global `fetch` function:

Figure 7.2

```
fetch = () => console.log('hacked');
fetch() // prints 'hacked' to console
```

Below is a more realistic example:

Figure 7.3

```
originalFetch = fetch;
fetch = (url, options) => {
  // Implement malicious attack here
  // For example: change some data in the request body

  // Then call original fetch implementation
  return originalFetch(url, options);
}
```

Of course, one can ask: why is it possible to modify the built-in method on the global object like that? Of course, it should not be possible, but unfortunately, it is.

Let's create a Vue.js application that performs authentication and authorization using the OpenID Connect protocol, an extension of the OAuth2 protocol.

In the main module below, we set up the global `fetch` to always return an error and only allow our `tryMakeHttpRequest` function to use the original global `fetch` method. Then, we register a service worker. If the service worker has already been registered, it is not registered again. Finally, we create the application (`App` component), activate

the router, activate the Pinia middleware for state management, and mount the application to a DOM node:

Figure 7.4

```
import { setupFetch } from "@/tryMakeHttpRequest";
setupFetch();
import { createApp } from "vue";
import { createPinia } from "pinia";
import App from "@/App.vue";
import router from "@/router";

if ("serviceWorker" in navigator) {
  await navigator.serviceWorker.register("/serviceWorker.js");
}

const app = createApp(App);
const pinia = createPinia();
app.use(pinia);
app.use(router);
app.mount("#app");
```

Below is the definition of the `App` component. After mounting, it will check whether the user is already authorized.

If the user is authorized, their authorization information will be fetched from the service worker, and the user's first name will be updated in the authorization information store. The user will be forwarded to the *Home* page. If the user is not authorized, authorization will be performed.

Figure 7.5

```
<template>
  <HeaderView />
  <router-view></router-view>
</template>

<script setup>
import { onMounted } from "vue";
import { useRouter } from "vue-router";
import authorizationService from "@/authService";
import { useAuthInfoStore } from "@/stores/authInfoStore";
import HeaderView from "@/HeaderView.vue";
import tryMakeHttpRequest from "@/tryMakeHttpRequest";

const router = useRouter();
const route = useRoute();

onMounted(async () => {
  const response = await
tryMakeHttpRequest("/authorizedUserInfo");
  const responseBody = await response.text();
  if (responseBody !== "") {
    const authorizedUserInfo = JSON.parse(responseBody);
    const { setFirstName } = useAuthInfoStore();
    setFirstName(authorizedUserInfo.firstName);
    router.push({ name: "home" });
```

```
  } else if (route.path !== '/auth') {
    authorizationService
      .tryAuthorize()
      .catch(() =>  router.push({ name: "auth-error" }));
  }
});
</script>
```

Figure 7.6

```
import { ref } from "vue";
import { defineStore } from "pinia";

export const useAuthInfoStore =
  defineStore("authInfoStore", () => {
    const firstName = ref('');

    function setFirstName(newFirstName: string) {
      firstName.value = newFirstName;
    }

    return { firstName, setFirstName };
  });
```

The header of the application displays the first name of the logged-in user and a button for logging the user out:

Figure 7.7

```
<template>
  <span>{{authInfoStore.firstName}}</span>
   
  <button @click="logout">Logout</button>
</template>

<script setup>
import { useRouter } from "vue-router";
import authorizationService from "@/authService";
import { useAuthInfoStore } from "@/stores/authInfoStore";

const authInfoStore = useAuthInfoStore();
const router = useRouter();

function logout() {
  authorizationService
    .tryLogout()
    .catch(() => router.push({ name: "auth-error" }));
}
</script>
```

The `tryMakeHttpRequest` function is a wrapper around the browser's global `fetch` method. It will start an authorization procedure if an HTTP request returns the HTTP status code 403 *Forbidden*.

Figure 7.8

```
import authorizationService from "@/authService";

let originalFetch: typeof fetch;

export default function tryMakeHttpRequest(
  url: RequestInfo,
  options?: RequestInit
): Promise<Response> {
  return originalFetch(url, options).then(async (response) => {
    if (response.status === 403) {
      try {
        await authorizationService.tryAuthorize();
      } catch {
        // Handle auth error, return response with status 403
      }
    }

    return response;
  });
}

export function setupFetch() {
  originalFetch = fetch;
  // @ts-ignore
  // eslint-disable-next-line no-global-assign
  fetch = () =>
    Promise.reject(new Error('Global fetch not implemented'));
}
```

Below is the implementation of the service worker:

Figure 7.9

```
const allowedOrigins = [
  "http://localhost:8080", // IAM in dev environment
  "http://localhost:3000", // API in dev environment
  "https://software-system-x.domain.com" // prod environment
];

const apiEndpointRegex = /\/api\//;
const tokenEndpointRegex = /\/openid-connect\/token$/;
const data = {};

// Listen to messages that contain data
// to be stored inside the service worker
addEventListener("message", (event) => {
  if (event.data) {
    data[event.data.key] = event.data.value;
  }
});

function respondWithUserInfo(event) {
  const response =
    new Response(data.authorizedUserInfo
                   ? JSON.stringify(data.authorizedUserInfo)
                   : '');
  event.respondWith(response);
}

function respondWithIdToken(event) {
```

```
  const response = new Response(data.idToken
                                ? data.idToken
                                : '');
  event.respondWith(response);
}

function respondWithTokenRequest(event) {
  let body = "grant_type=authorization_code";
  body += `&code=${data.code}`;
  body += `&client_id=app-x`;
  body += `&redirect_uri=${data.redirectUri}`;
  body += `&code_verifier=${data.codeVerifier}`;
  const tokenRequest = new Request(event.request, { body });

  // Verify that state received from the authorization
  // server is same as sent by this app earlier
  if (data.state === data.receivedState) {
    event.respondWith(fetch(tokenRequest));
  } else {
    // Handle error
  }
}

function respondWithApiRequest(event) {
  const headers = new Headers(event.request.headers);

  // Add Authorization header that contains the access token
  if (data.accessToken) {
    headers.append("Authorization",
                   `Bearer ${data.accessToken}`);
  }

  const authorizedRequest = new Request(event.request, {
    headers
  });

  event.respondWith(fetch(authorizedRequest));
}

function fetchHandler(event) {
  const requestUrl = new URL(event.request.url);

  if (event.request.url.endsWith('/authorizedUserInfo') &&
      !apiEndpointRegex.test(requestUrl.pathname)) {
    respondWithUserInfo(event);
  } else if (event.request.url.endsWith('/idToken') &&
             !apiEndpointRegex.test(requestUrl.pathname)) {
    respondWithIdToken(event);
  } else if (allowedOrigins.includes(requestUrl.origin)) {
    if (tokenEndpointRegex.test(requestUrl.pathname)) {
      respondWithTokenRequest(event);
    } else if (apiEndpointRegex.test(requestUrl.pathname)) {
      respondWithApiRequest(event);
    }
  } else {
    event.respondWith(fetch(event.request));
  }
}

// Intercept all fetch requests and handle
// them with 'fetchHandler'
addEventListener("fetch", fetchHandler);
```

Authorization using the OAuth2 Authorization Code Flow is started with a browser redirect to a URL of the following kind:

Figure 7.10

```
https://authorization-server.com/auth?
response_type=code&client_id=CLIENT_ID&redirect_uri\
=https://example-
app.com/cb&scope=photos&state=1234zyx...ghvx3&code_challenge=COD
E_CHALLE
NGE&code_challenge_method=SHA256
```

The query parameters in the above URL are the following:

- *response_type=code* - Indicates that you expect to receive an authorization code
- *client_id* - The client id you used when you created the client on the authorization server (We'll create the client a bit later)
- *redirect_uri* - Indicates the URI to redirect the browser after authorization is completed. You also need to define this URI in the authorization server. (We'll see this a bit later)
- *scope* - One or more scope values indicating which parts of the user's account you wish to access. Scopes should be separated by URL-encoded space characters
- *state* - A random string generated by your application, which you'll verify later
- *code_challenge* - PKCE extension: URL-safe base64-encoded SHA256 hash of the code verifier. A code verifier is a random string secret you generate
- *code_challenge_method=S256* - PKCE extension: indicates which hashing method is used (S256 means SHA256)

We should use the PKCE extension as an additional security measure. PKCE extends the Authorization Code Flow to prevent CSRF and authorization code injection attacks.

If authorization is successful, the authorization server will redirect the browser to the above-given *redirect_uri* with *code* and *state* given as URL query parameters, for example:

Figure 7.11

```
https://example-app.com/cb?
code=AUTH_CODE_HERE&state=1234zyx...ghvx3
```

- *code* - The authorization code returned from the authorization server
- *state* - The same state value that you passed earlier

After the application is successfully authorized, tokens can be requested with the following kind of HTTP POST request:

Figure 7.12

```
POST https://authorization-server.com/token HTTP/1.1
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=AUTH_CODE_HERE&
redirect_uri=REDIRECT_URI&
client_id=CLIENT_ID&
code_verifier=CODE_VERIFIER
```

- *grant_type=authorization_code* - The grant type for this flow is _authorization*code*
- *code=AUTH_CODE_HERE* - This is the code you received when the browser was redirected back to your application from the authorization server.
- *redirect_uri=REDIRECT_URI* - Must be identical to the redirect URI provided earlier during authorization
- *client_id=CLIENT_ID* - The client id you used when you created the client on the authorization server
- *code_verifier=CODE_VERIFIER* - The random string secret you generated earlier

Below is the implementation of the `AuthorizationService` class. It provides methods for authorization, getting tokens and logout.

Figure 7.13

```
import pkceChallenge from "pkce-challenge";
import jwt_decode from "jwt-decode";
import tryMakeHttpRequest from "@/tryMakeHttpRequest";
import type { useAuthInfoStore } from "@/stores/authInfoStore";

interface AuthorizedUserInfo {
  readonly userName: string;
  readonly firstName: string;
  readonly lastName: string;
  readonly email: string;
}

export default class AuthorizationService {
  constructor(
    private readonly oidcConfigurationEndpoint: string,
    private readonly clientId: string,
    private readonly authRedirectUrl: string,
    private readonly loginPageUrl: string
  ) {}

  // Try to authorize the user using the OpenID Connect
  // Authorization Code Flow
  async tryAuthorize(): Promise<void> {
    // Store the redirect URI in service worker
    navigator.serviceWorker?.controller?.postMessage({
      key: "redirectUri",
      value: this.authRedirectUrl
    });

    // Store the state secret in service worker
    const state = crypto.randomUUID();
    navigator.serviceWorker?.controller?.postMessage({
      key: "state",
      value: state,
    });

    // Generate a PKCE challenge and store
    // the code verifier in service worker
    const challenge = pkceChallenge(128);
    navigator.serviceWorker?.controller?.postMessage({
      key: "codeVerifier",
      value: challenge.code_verifier,
    });

    const authUrl = await this.tryCreateAuthUrl(state,
challenge);

    // Redirect the browser to authorization server's
    // authorization URL
    location.href = authUrl;
  }

  // Try get access, refresh and ID token from
  // the authorization server's token endpoint
  async tryGetTokens(
    authInfoStore: ReturnType<typeof useAuthInfoStore>
  ): Promise<void> {
    const oidcConfiguration = await this.getOidcConfiguration();

    const response =
      await tryMakeHttpRequest(oidcConfiguration.token_endpoint,
```

```
{
        method: "post",
        mode: "cors",
        headers: {
          "Content-Type": "application/x-www-form-urlencoded",
        },
    });

    const tokens = await response.json();
    this.storeTokens(tokens);
    this.storeAuthorizedUserInfo(tokens.id_token,
authInfoStore);
  }

  // Logout and redirect to login page
  async tryLogout(): Promise<void> {
    const oidcConfiguration = await this.getOidcConfiguration();

    // Clear authorized user info in service worker
    navigator.serviceWorker?.controller?.postMessage({
      key: "authorizedUserInfo",
      value: undefined
    });

    // Get ID token from service worker
    const response = await tryMakeHttpRequest("/idToken");
    const idToken = await response.text();

    // Redirect browser to authorization server's
    // logout endpoint
    if (idToken !== "") {
      location.href =
        oidcConfiguration.end_session_endpoint +
        `?post_logout_redirect_uri=${this.loginPageUrl}` +
        `&id_token_hint=${idToken}`;
    } else {
      location.href = oidcConfiguration.end_session_endpoint;
    }
  }

  private async getOidcConfiguration(): Promise<any> {
    const response =
      await tryMakeHttpRequest(this.oidcConfigurationEndpoint);

    return response.json();
  }

  private async tryCreateAuthUrl(
    state: string,
    challenge: ReturnType<typeof pkceChallenge>
  ) {
    const oidcConfiguration = await this.getOidcConfiguration();
    let authUrl = oidcConfiguration.authorization_endpoint;

    authUrl += "?response_type=code";
    authUrl += "&scope=openid+profile+email";
    authUrl += `&client_id=${this.clientId}`;
    authUrl += `&redirect_uri=${this.authRedirectUrl}`;
    authUrl += `&state=${state}`;
    authUrl += `&code_challenge=${challenge.code_challenge}`;
    authUrl += "&code_challenge_method=S256";
```

```
    return authUrl;
  }

  private storeTokens(tokens: any) {
    navigator.serviceWorker?.controller?.postMessage({
      key: "accessToken",
      value: tokens.access_token,
    });

    navigator.serviceWorker?.controller?.postMessage({
      key: "refreshToken",
      value: tokens.refresh_token,
    });

    navigator.serviceWorker?.controller?.postMessage({
      key: "idToken",
      value: tokens.id_token,
    });
  }

  private storeAuthorizedUserInfo(
    idToken: any,
    authInfoStore: ReturnType<typeof useAuthInfoStore>
    ) {
    const idTokenClaims: any = jwt_decode(idToken);

    const authorizedUserInfo = {
      userName: idTokenClaims.preferred_username,
      firstName: idTokenClaims.given_name,
      lastName: idTokenClaims.family_name,
      email: idTokenClaims.email,
    };

    navigator.serviceWorker?.controller?.postMessage({
      key: "authorizedUserInfo",
      value: authorizedUserInfo
    });

    authInfoStore.setFirstName(idTokenClaims.given_name);
  }
}
```

Below is an example response you get when you execute the `tryMakeHttpRequest` function in the `tryGetTokens` method:

Figure 7.14

```
{
  "access_token": "eyJz93a...k4laUWw",
  "id_token": "UFn43f...c5vvfGF",
  "refresh_token": "GEbRxBN...edjnXbL",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

The `AuthorizationCallback` component is the component that will be rendered when the authorization server redirects the browser back to the application after successful

authorization. This component stores the authorization code and the received state in the service worker and initiates a token request. After receiving tokens, it will route the application to the home page. As an additional security measure, the token request will only be performed if the original *state* and *received state* are equal. This check is done in the service worker code.

Figure 7.15

```
<template>
  <div></div>
</template>

<script setup>
import { onMounted } from "vue";
import { useRouter, useRoute } from "vue-router";
import authorizationService from "@/authService";
import { useAuthInfoStore } from "@/stores/authInfoStore";

const { query } = useRoute();
const router = useRouter();
const authInfoStore = useAuthInfoStore();

onMounted(async () => {
  // Store authorization code in service worker
  navigator.serviceWorker?.controller?.postMessage({
    key: "code",
    value: query.code,
  });

  // Store received state in service worker
  navigator.serviceWorker?.controller?.postMessage({
    key: "receivedState",
    value: query.state,
  });

  // Try fetch tokens
  try {
    await authorizationService.tryGetTokens(authInfoStore);
    router.push({ name: "home" });
  } catch (error) {
    router.push({ name: "auth-error" });
  }
});
</script>
```

Other UI components the application uses are defined below:

Figure 7.16

```
<template>
  <div>Error</div>
</template>
```

Figure 7.17

```
<template>
  <div>Login</div>
</template>
```

Figure 7.18

```
<template>
  <div>Home</div>
</template>
```

The application's router is the following:

Figure 7.19

```
import { createRouter, createWebHistory } from "vue-router";
import AuthorizationCallback from "@/AuthorizationCallback.vue";
import AuthorizationError from "@/AuthorizationError.vue";
import HomeView from "@/HomeView.vue";
import LoginView from "@/LoginView.vue";

const routes = [
  {
    path: "/",
    name: "login",
    component: LoginView,
  },
  {
    path: "/auth",
    name: "auth",
    component: AuthorizationCallback,
  },
  {
    path: "/auth-error",
    name: "auth-error",
    component: AuthorizationError,
  },
  {
    path: "/home",
    name: "home",
    component: HomeView,
  },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});

export default router;
```

The below *authService* module contains definitions of needed constants and creates an instance of the `AuthorizationService` class. The below code contains hard-coded values for a local development environment. In real life, these values should be taken from environment variables. The below values work if you have a Keycloak service running at

localhost:8080 and the Vue app running at *localhost:5173*. You must create a client in the Keycloak named ‘app-x’. Additionally, you must define a valid redirect URI and add an allowed web origin. Lastly, you must configure a valid post-logout redirect URI (see the below image). The default access token lifetime in Keycloak is just one minute. You can increase that for testing purposes in the realm settings (the token tab)

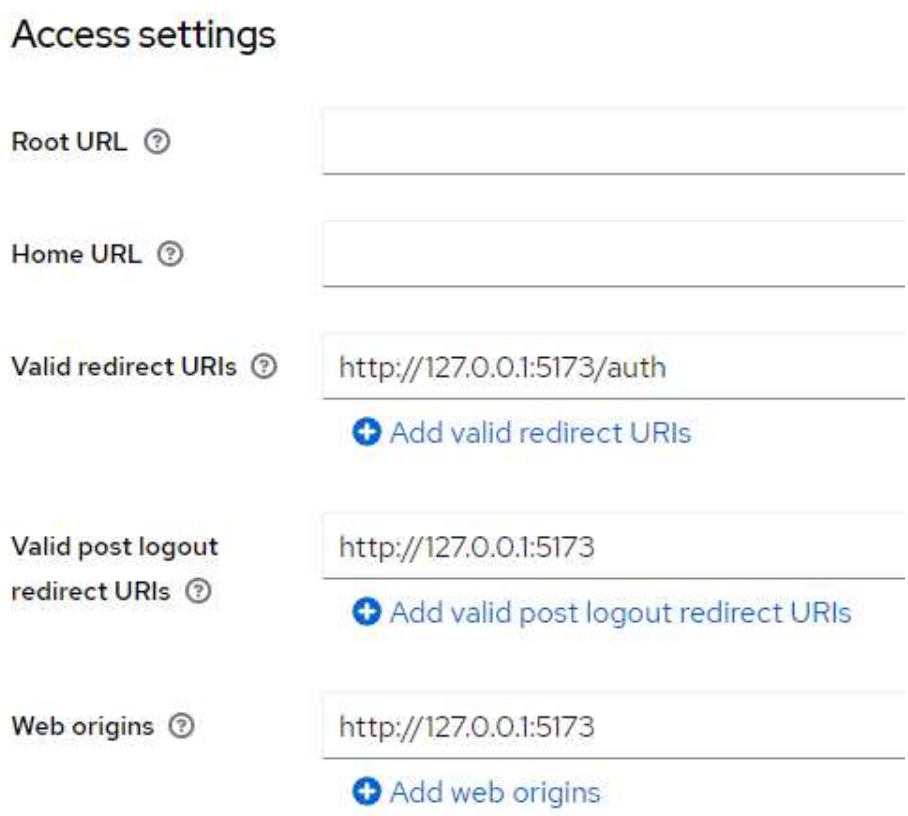

Figure 7.20. Keycloak Settings for the Client

Figure 7.21

```
import AuthorizationService from "@/AuthorizationService";

const oidcConfigurationEndpoint =
  "http://localhost:8080/realms/master/.well-known/openid-
configuration";

const clientId = "app-x";
const redirectUrl = "http://127.0.0.1:5173/auth";
const loginPageUrl = "http://127.0.0.1:5173";

const authorizationService = new AuthorizationService(
  oidcConfigurationEndpoint,
  clientId,
  redirectUrl,
```

```
  loginPageUrl
);

export default authorizationService;
```

7.4.1.2: OAuth2 Authorization in Backend

Only let authorized users access resources. The best way not to forget to implement authorization is to deny access to resources by default. For example, you can require that an authorization decorator is specified for all controller methods. If an API endpoint does not require authorization, a special decorator like `@allow_any_user` could be used. An exception should be thrown if a controller method lacks an authorization decorator. This way, you can never forget to add an authorization annotation to a controller method.

Broken access control is number one in the OWASP Top 10 for 2021. Remember to disallow users to create resources for other users. Also, you must disallow users to view, edit, or delete resources belonging to someone else (also known as Insecure Direct Object Reference (IDOR) prevention). It is not enough to use universally unique ids (UUIDs) as ids for resources instead of basic integers. This is because if an attacker can obtain a URL for an object with a UUID, he can access the object behind the URL because there is no access control in place.

Below is a JWT-based authorizer class that can be used in a FastAPI API service. We use the following additional libraries in the example: python-benedict and pyjwt. The below example utilizes role-based access control (RBAC), but there are more modern alternatives, including attribute-based access control (ABAC) and relationship-based access control (ReBAC). More information about those is available in OWASP Authorization Cheat Sheet

Figure 7.22

```
from typing import Protocol

class Authorizer(Protocol):
    # ...
```

Figure 7.23. jwt_authorizer.py

```
import os
from typing import Any, Final
from collections.abc import Callable

import requests
from Authorizer import Authorizer
from benedict import benedict
from fastapi import HTTPException, Request
from jwt import PyJWKClient, PyJWKClientError, decode
from jwt.exceptions import InvalidTokenError


class __JwtAuthorizer(Authorizer):
    IAM_ERROR: Final = 'IAM error'

    def __init__(self):
        # OpenId Connect configuration endpoint in the IAM
system
        self.__oidc_config_url = os.environ['OIDC_CONFIG_URL']
        self.__jwks_client = None

        # With Keycloak you can use e.g., realm_access.roles
        self.__roles_claim_path =
os.environ['JWT_ROLES_CLAIM_PATH']

        # This is the URL where you can fetch the user id for a
        # specific 'sub' claim value in the access token
        # For example: http://localhost:8082/user-service/users
        self.__get_users_url = os.environ['GET_USERS_URL']

    def authorize(self, request: Request) -> None:

self.__decode_jwt_claims(request.headers.get('Authorization'))

    # Authorize a user to create a resource for self
    # Checks that the supplied user_id is the same as the
user_id
    # of the user owning the JWT
    # Note! For some IAM systems other than Keycloak,
    # you might need to use 'uid'
    # claim instead of 'sub' to get unique user id
    def authorize_for_self(
        self, user_id: int, request: Request
    ) -> None:
        jwt_user_id = self.__get_jwt_user_id(request)
        user_is_authorized = user_id == jwt_user_id
        if not user_is_authorized:
            raise HTTPException(status_code=403,
detail='Unauthorized')

    # Authorize a user for his/hers own resources only
    # If an entity with given id and user_id combination
    # is not found, auth error is raised
    def authorize_for_user_own_resources_only(
        self,
        id: int,
        get_entity_by_id_and_user_id: Callable[[int, int], Any],
        request: Request
    ) -> None:
        jwt_user_id = self.__get_jwt_user_id(request)
```

```
        try:
            get_entity_by_id_and_user_id(id, jwt_user_id)
        except HTTPException as error:
            if error.status_code == 404:
                raise HTTPException(status_code=403,
detail='Unauthorized')
            # Log error details
            raise HTTPException(status_code=500,
detail=self.IAM_ERROR)

    def authorize_if_user_has_one_of_roles(
        self, allowed_roles: list[str], request: Request
    ) -> None:
        claims = self.__decode_jwt_claims(
            request.headers.get('Authorization')
        )

        try:
            roles = benedict(claims)[self.__roles_claim_path]
        except KeyError as error:
            # Log error details
            raise HTTPException(status_code=500,
detail=self.IAM_ERROR)

        user_is_authorized = any(
            [True for role in roles if role in allowed_roles]
        )
        if not user_is_authorized:
            raise HTTPException(status_code=403,
detail='Unauthorized')

    def __decode_jwt_claims(
        self, auth_header: str | None
    ) -> dict[str, Any]:
        if not auth_header:
             raise HTTPException(status_code=401,
detail='Unauthenticated')

        try:
            if not self.__jwks_client:
                oidc_config_response =
requests.get(self.__oidc_config_url)
                oidc_config_response.raise_for_status()
                oidc_config = oidc_config_response.json()
                self.__jwks_client =
PyJWKClient(oidc_config['jwks_uri'])

            jwt = auth_header.split('Bearer ')[1]
            signing_key =
self.__jwks_client.get_signing_key_from_jwt(jwt)
            jwt_claims = decode(jwt, signing_key.key,
algorithms=['RS256'])
        except (
            requests.RequestException,
            KeyError,
            PyJWKClientError
        ) as error:
            # Log error details
            raise HTTPException(status_code=500,
detail=self.IAM_ERROR)
        except (IndexError, InvalidTokenError):
            raise HTTPException(status_code=403,
```

```
detail='Unauthorized')

        return jwt_claims

    def __get_jwt_user_id(self, request: Request) -> int:
        claims = self.__decode_jwt_claims(
            request.headers.get('Authorization')
        )

        try:
            sub_claim = claims['sub']
            users_response = requests.get(
                f'{self.__get_users_url}?sub=
{sub_claim}&fields=id'
            )
            users_response.raise_for_status()
            # Response JSON is expected in the form [{ "id":
12345 }]
            users = users_response.json()
        except (KeyError, requests.RequestException) as error:
            # Log error details
            raise HTTPException(status_code=500,
detail=self.IAM_ERROR)

        try:
            return users[0].id
        except (IndexError, AttributeError):
            raise HTTPException(status_code=403,
detail='Unauthorized')


authorizer = __JwtAuthorizer()
```

Below is an example API service that utilizes the above-defined `JwtAuthorizer`:

Figure 7.24

```
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse
from starlette.exceptions import HTTPException as
StarletteHTTPException

from InputOrder import InputOrder
from jwt_authorizer import authorizer
from order_service import order_service
# OrderUpdate is a DTO that should not have user_id attribute,
# because it cannot be changed
from OrderUpdate import OrderUpdate

app = FastAPI()

# Define a custom HTTPException handler that provides
# admin logging and metrics update
@app.exception_handler(StarletteHTTPException)
async def http_exception_handler(
    request: Request, error: StarletteHTTPException
):
    if error.status_code == 403:
        # Audit log an unauthorized request
```

```
    # Increment 'HTTP request failures' counter by one
    # using the following metric labels: error.status_code,
error.detail

    return JSONResponse({'error: ': str(error.detail)},
status_code=error.status_code)

@app.get('/sales-item-service/sales-items')
async def get_sales_items():
    # No authentication/authorization required
    # Send sales items

@app.post('/messaging-service/messages')
async def create_message(request: Request):
    authorizer.authorize(request)
    # Authenticated user can create a message

@app.get('/order-service/orders/{id}')
async def get_order(id: int, request: Request):
    authorizer.authorize_for_user_own_resources_only(
        id,
        order_service.get_order_by_id_and_user_id,
        request
    )

    # Get order identified with 'id'
    # and having user id of JWT's owner

@app.post('/order-service/orders')
async def create_order(order: InputOrder, request: Request):
    authorizer.authorize_for_self(
        order.user_id,
        request
    )

    # Create an order for the user
    # User cannot create orders for other users

@app.put('/order-service/orders/{id}')
async def update_order(id: int, order: OrderUpdate, request:
Request):
    authorizer.authorize_for_user_own_resources_only(
        id,
        order_service.get_order_by_id_and_user_id,
        request
    )

    # Update an order identified with 'id'
    # and user id of JWT's owner

@app.delete('/order-service/orders/{id}')
async def delete_order(id: int, request: Request):
    authorizer.authorize_if_user_has_one_of_roles(
        ['admin'], request
    )

    # Only admin user can delete an order
```

The authorization is separately coded inside each request handler in the above example. We could extract the authorization code from the request handler methods to decorators that can be used with the methods. The decorators could be implemented in a separate library, and they can accept any authorizer that implements the `Authorizer` protocol, not just the `JwtAuthorizer`:

Figure 7.25

```
from collections.abc import Callable
from functools import wraps
from typing import Any

from Authorizer import Authorizer


class AuthDecorException(Exception):
    pass


def allow_any_user(handle_request):
    return handle_request


def allow_authorized_user(authorizer: Authorizer):
    def decorate(handle_request):
        @wraps(handle_request)
        async def wrapped_handle_request(*args, **kwargs):
            try:
                authorizer.authorize(kwargs['request'])
            except KeyError:
                raise AuthDecorException(
                    "Request handler must accept 'request'
parameter"
                )
            return await handle_request(*args, **kwargs)
        return wrapped_handle_request
    return decorate


def allow_for_self(authorizer: Authorizer):
    def decorate(handle_request):
        @wraps(handle_request)
        async def wrapped_handle_request(*args, **kwargs):
            try:
                user_id = (
                    kwargs['user_id']
                    if kwargs.get('user_id')
                    else kwargs[
                        [
                            key
                            for key in kwargs.keys()
                            if key.endswith('dto')
                        ][0]
                    ].user_id
                )

authorizer.authorize_for_user_own_resources_only(
```

```
                    user_id, kwargs['request']
                )
            except (AttributeError, IndexError, KeyError):
                raise AuthDecorException("""
                    Request handler must accept 'request'
parameter,
                    'user_id' integer parameter or DTO parameter
                    with name ending with 'dto'. DTO parameter
                    must have attribute 'user_id'
                    """)
            return await handle_request(*args, **kwargs)
        return wrapped_handle_request
    return decorate


def allow_for_user_own_resources_only(
    authorizer: Authorizer,
    get_entity_by_id_and_user_id: Callable[[int, int], Any]
):
    def decorate(handle_request):
        @wraps(handle_request)
        async def wrapped_handle_request(*args, **kwargs):
            try:

authorizer.authorize_for_user_own_resources_only(
                    kwargs['id'],
                    get_entity_by_id_and_user_id,
                    kwargs['request']
                )
            except (KeyError):
                raise AuthDecorException(
                    "Request handler must accept 'id' and
'request' parameters"
                )
            return await handle_request(*args, **kwargs)
        return wrapped_handle_request
    return decorate


def allow_for_user_roles(roles: list[str], authorizer:
Authorizer):
    def decorate(handle_request):
        @wraps(handle_request)
        async def wrapped_handle_request(*args, **kwargs):
            try:
                authorizer.authorize_if_user_has_one_of_roles(
                    roles, kwargs['request']
                )
            except KeyError:
                raise AuthDecorException(
                    "Request handler must accept 'request'
parameter"
                )
            return await handle_request(*args, **kwargs)
        return wrapped_handle_request
    return decorate
```

In the above example, we implemented decorators that can take parameters. Those decorators have three levels of nested

functions compared to non-parameterized ones with only two levels of nested functions. Standard decorators without parameters can be used in the following way:

Figure 7.26

```
def my_decorator(func):
    # ...

@my_decorator
def func():
   # ...

# The above is same as:

my_decorated_func = my_decorator(func)
```

Decorators that take parameters can be used in the following way:

Figure 7.27

```
def my_decorator(arg):
    def decorate(func):
        # ...
    return decorate

@my_decorator(some_arg)
def func():
   # ...

# The above is the same as:

my_decorated_func = my_decorator(some_arg)(func)
```

In the above authorization decorator example, we used the `@wraps` decorator from the `functools` module. This is because how FastAPI works with the request handler methods. If we did not use the `@wraps` decorator, we would get an error, which states that FastAPI is expecting `args` and `kwargs` parameters for a request handler. All the authorization decorators except the `@allow_any_user` decorator require that FastAPI request handlers accept a `request` argument. The `@allow_for_self` decorator also requires that a FastAPI request handler accepts either a `user_id` argument or a DTO argument with the argument name ending with the `dto` word. This DTO argument must be an object having a `user_id` attribute.

Now, we can use the above-defined authorization decorators when defining the request handlers as follows:

Figure 7.28

```
# Imports ...

@app.get('/sales-item-service/sales-items')
@allow_any_user
async def get_sales_items():
    # ...

@app.post('/messaging-service/messages')
@allow_authorized_user(authorizer)
async def create_message(request: Request):
    # ...

@app.get('/order-service/orders/{id}')
@allow_for_user_own_resources_only(
    authorizer,
    order_service.get_order_by_id_and_user_id
)
async def get_order(id: int, request: Request):
    # ...

@app.post('/order-service/orders')
@allow_for_user_own_resources_only(authorizer)
async def create_order(
    order_dto: InputOrder, request: Request
):
    # ...

@app.put('/order-service/orders/{id}')
@allow_for_user_own_resources_only(
    authorizer,
    order_service.get_order_by_id_and_user_id
)
async def update_order(
    id: int, order: OrderUpdate, request: Request
):
    # ...

@app.delete('/order-service/orders/{id}')
@allow_for_user_roles(['admin'], authorizer)
async def delete_order(id: int, request: Request):
    # ...
```

Next, we create a function to check that all request handlers in a microservice project contain an authorization decorator. This function should be called before starting the microservice. You can put this function and its call into a microservice starter project so that all new microservices created from the starter project automatically check for the presence of an authorization decorator in all request handlers.

Figure 7.29

```
import os


class AuthDecorNotSpecifiedException(Exception):
    def __init__(self, file_name: str, line_number: int):
        self.__file_name = file_name
```

```
        self.__line_number = line_number

    def __str__(self):
        return f'Auth decorator not specified in file
{self.__file_name} line {self.__lin\
e_number}'


def ensure_request_handlers_have_auth*decor():
    for path, *, file_names in os.walk('./'):
        for file_name in file_names:
            if file_name.endswith('.py'):
                file_path_name = os.path.join(path, file_name)
                with open(file_path_name) as file:
                    lines = file.readlines()
                prev_line = ''
                for line_index, line in enumerate(lines):
                    line = line.strip()
                    if any(
                        [
                            prev_line.startswith(decorator)
                            for decorator in [
                                '@app.get',
                                '@app.put',
                                '@app.patch',
                                '@app.post',
                                '@app.delete',
                            ]
                        ]
                    ):
                        if not line.startswith('@allow_'):
                            line_number = line_index + 1
                            raise
AuthDecorNotSpecifiedException(
                                file_name, line_number
                            )
                    prev_line = line


if os.environ.get('ENV') == 'DEVELOPMENT':
    ensure_request_handlers_have_auth_decor()
```

The above code walks all Python files in the current directory and its subdirectories, reads their contents, and verifies that an authorization decorator (`@allow_xxx`) is present after the following decorators: `@app.get/put/patch/post/delete`.

7.4.2: Password Policy

Implement a password policy requiring strong passwords and prefer passphrases over passwords. A passphrase is supposed to contain multiple words. Passphrases are harder to guess by attackers and easier to remember by users than strong passwords. Allow passphrases to contain Unicode characters.

This allows users to create passphrases using their mother tongue.

You should require that passwords are strong and match the following criteria:

- At least 12 characters long
- At least one uppercase letter
- At least one lowercase letter
- At least one number
- At least one special character
- May not contain the username
- May not contain too many identical digits or letters, e.g., a password containing “111111”, “aaaaaa,” or “1a1a1a1a1a” should be denied
- May not contain too many consecutive numbers or letters, e.g., a password containing “12345”, “56789”, “abcdef”, or “klmno” should be denied
- May not contain too many adjacent letters in the keyboard, e.g., a password containing “qwerty” should be denied
- May not contain a black-listed word: black-list all commonly used, easy-to-guess passwords.

Machine-to-machine (non-human-related) passwords (like database passwords) should be automatically generated separately for each production environment during the deployment. These passwords should be random and significantly longer than 12 characters.

7.4.3: Cryptography

The following are the key security features to implement related to cryptography:

- Do not transmit data in clear text

    - You don’t need to implement HTTPS in all the microservices because you can set up a service mesh, like Istio, and configure it to implement mTLS between services
- Do not store sensitive information like personally identifiable information (PII) in clear text
    - Encrypt sensitive data before storing it in a database and decrypt it upon fetching from the database
    - Remember to identify which data is classified as sensitive according to privacy laws, regulatory requirements, or business needs
    - Do not use legacy protocols such as FTP and SMTP for transporting sensitive data
    - Discard sensitive data as soon as possible or use tokenization (e.g., PCI DSS compliant) or even truncation
    - Do not cache sensitive data
- Do not use old/weak cryptographic algorithms. Use robust algorithms like SHA-256 or AES-256
- Do not allow the use of default/weak passwords or default encryption keys in a production environment
    - You can implement validation logic for passwords/encryption keys in microservices. This validation logic should be automatically activated when the microservice runs in production. The validation logic should be the following: If passwords/encryption keys supplied to the microservice are not strong enough, the microservice should not run at all but exit with an error

7.4.3.1: Encryption Key Lifetime and Rotation

Encryption keys should be rotated (i.e., changed) when one or more of the following criteria is met:

- The current key is known to be compromised, or there is a suspicion of compromise
- A specified period of time has elapsed (this is known as the cryptoperiod)
- The key has been used to encrypt a particular amount of data
- There is a significant change to the security the used encryption algorithm provides (for example, a new attack was announced)

Encryption key rotation should happen so that all existing data is decrypted and encrypted with the new key. This will happen gradually, so each encrypted database table row must contain an id of the used encryption key. When all existing data is encrypted with the new key, meaning all references to it are removed, the old key can be destroyed.

7.4.4: Denial-of-service (DoS) Prevention

DoS prevention should happen at least in the following ways:

- Establish request rate limiting for microservices. This can be done at the API gateway level or by the cloud provider
- Use a Captcha to prevent non-human (robotic) users from performing potentially expensive operations like creating new resources or fetching large resources, like large files, for example

7.4.5: Database Security

- Connection from a microservice to a database must be secured using TLS. In a Kubernetes environment, you can implement this by taking a service mesh (like Istio) into use and configuring mTLS between all services in the environment
- Database credentials (username and password) must be stored in a secure location, like in a Secret in a

Kubernetes environment. Never store credentials in source code

- Use a strong password, preferably an automatically generated one for the specific environment. The password must be random and long enough. The password should be at least 32 characters long if the database engine allows it
- Configure different database user accounts for admin and regular usage. Assign minimum privileges to both user accounts. Have separate passwords for each account. The regular database user can usually only execute the following SQL statements: SELECT, INSERT, UPDATE, and DELETE. Only an administrator database user can create/modify/drop tables/indexes, etc.

7.4.6: SQL Injection Prevention

- Use parameterized SQL statements. Do not concatenate user-supplied data directly to an SQL statement string
- Remember that you cannot use parameterization in all parts of an SQL statement. If you must put user-supplied data into an SQL statement without parameterization, sanitize/validate it first. For example, for `LIMIT,` you must validate that the user-supplied value is an integer and in a given range
- Migrate to use ORM (Object Relational Mapping)
- Use proper limiting on the number of fetched records within queries to prevent mass disclosure of records
- Verify the correct shape of at least the first query result row. Do not send the query result to the client if the shape of the data in the first row is wrong, e.g., it contains the wrong fields.

7.4.7: OS Command Injection Prevention

You should not allow user-supplied data to be used when executing OS commands in a shell. For example, don’t allow

the following:

Figure 7.30

```
import os

user_supplied_dir = ...
os.system(f'mkdir {user_supplied_dir}')
```

A malicious user can supply, for example, the following kind of directory: `some_dir && rm -rf /.`

Instead, use a specific function provided by the `os` module:

Figure 7.31

```
import os

user_supplied_dir = ...
os.mkdir(user_supplied_dir)
```

7.4.8: Security Configuration

By default, the security context for containers should be the following:

- Container should not be privileged
- All capabilities are dropped
- Container filesystem is read-only
- Only a non-root user is allowed to run inside the container
- Define the non-root user and group under which the container should run
- Disallow privilege escalation

The *DevSecOps principles* chapter later in the book gives an example of the above Docker container security configuration.

Implement the sending of security-related HTTP response headers in the API gateway:

- `X-Content-Type-Options: nosniff`
- `Strict-Transport-Security: max-age: ; includeSubDomains`

- `X-Frame-Options: DENY`
- `Content-Security-Policy: frame-ancestors 'none'`
- `Content-Type: application/json`
- If caching is not specifically enabled and configured, the following header should be set: `Cache-Control: no-store`
- `Access-Control-Allow-Origin: https://your_domain_here`

If you are returning HTML instead of JSON, you should replace/add the following response headers:

- `Content-Security-Policy: default-src 'none'`
- `Referrer-Policy: no-referrer`

Disable browser features that are not needed/wanted using the `Permissions-Policy` response header. The below example turns off all the listed features:

Figure 7.32

```
Permissions-Policy: accelerometer=(), ambient-light-sensor=(),
  autoplay=(), battery=(), camera=(), cross-origin-isolated=(),
  display-capture=(), document-domain=(), encrypted-media=(),
  execution-while-not-rendered=(), execution-while-out-of-
viewport=(),
  fullscreen=(), geolocation=(), gyroscope=(), keyboard-map=(),
  magnetometer=(), microphone=(), midi=(), navigation-override=
(),
  payment=(), picture-in-picture=(), publickey-credentials-get=
(),
  screen-wake-lock=(), sync-xhr=(), usb=(), web-share=(), xr-
spatial-tracking=()
```

Read more about HTTP response security headers.

7.4.9: Automatic Vulnerability Scanning

Implement automatic vulnerability scanning in microservice CI pipelines and/or the container registry at regular intervals. It is vital to configure container vulnerability scanning in the container registry (e.g., Docker or one provided by the cloud vendor). This scanning should preferably happen once a day. All software components of the software system should be

scanned. You should correct critical or high-severity vulnerabilities as soon as possible.

7.4.10: Integrity

Use only container images with tags that have an SHA digest. If an attacker succeeds in publishing a malicious container image with the same tag, the SHA digest prevents that malicious image from being taken into use. Ensure you use libraries and dependencies from trusted sources, like PyPi. You can also host internal mirrors of repositories to avoid accidentally using any untrusted repository. Ensure a review process exists for all code (source, deployment, infrastructure) and configuration changes so that no malicious code can be introduced into your software system.

7.4.11: Error Handling

Ensure that error messages in API responses do not contain sensitive information or details of the implementation. Do not add stack traces to error responses transmitted to clients in a production environment.

For example, suppose an API request produces an internal server error related to connectivity to an IAM system. In that case, you should not reveal implementation details in the error response, like talk about *Keycloak 18.06* if that's what you are using, but use an abstract term, like the *IAM system*. Suppose an attacker gets an error response revealing details about a 3rd party software component and its version. In that case, the attacker can exploit possible vulnerabilities of the particular software component.

7.4.12: Logging

When writing log entries, never write any of the below to the log:

- Session ids
- Access tokens

- Personally identifiable information (PII)
- Passwords
- Database connection strings
- Encryption keys
- Information that is not legal to collect
- Information that the end-user has opted out of the collection

7.4.13: Audit Logging

Auditable end-user-related events, such as logins, failed logins, unauthorized or invalid requests, and high-value transactions, should be logged and stored in an external audit logging system. The audit logging system should automatically detect suspicious action related to an end-user and alert about it. See also OWASP Logging Vocabulary Cheat Sheet

7.4.14: Input Validation

Always validate input from untrusted sources, like from an end-user. There are many ways to implement validation, and several libraries exist for that purpose. Let's assume you implement entities and data transfer objects (DTOs). The best way to ensure proper validation is to require that each DTO property must have a validation decorator. If a property in a DTO does not require any validation, annotate that property with a special annotation, like `@any_value`, for example. Don't use entities in data transfer. Always use DTOs in both directions, in data transfer from the client to the server and from the server to the client. This is because the entities usually contain some data that is not expected from clients and can contain sensitive/confidential information that should not be exposed to clients. The input DTOs verify that the client supplies correct data: All needed properties are supplied with correct types and semantics, and no extra properties are supplied. If you need dynamic validation, i.e., enforce a business rule, put that validation code into the entity class. It is

essential to filter out extra properties because otherwise, those might end up in the database if a schemaless database table (e.g., a MongoDB collection) is used. Disallowing a schemaless database table in a production system is good practice.

For example, a `User` entity might have an attribute `is_admin`. You should not expect a `User` entity as input from a client, but you should expect a DTO, `InputUser`, which has the same attributes as the `User` entity, except specific attributes like `id`, `created_at_timestamp` and `is_admin`. Similarly, if the `User` entity contains a `password` attribute, that attribute should not be sent to a client. For this reason, you need to define an `OutputUser` DTO, which has the same attributes as the `User` entity except the `password` attribute. When you use DTOs both in input and output data transfer, you safeguard the service against situations where you add a new sensitive or internal attribute to an entity, and that new sensitive or internal attribute should not be received from or transmitted to clients.

Remember to validate data from all untrusted sources:

- Command line arguments
- Environment variables
- Standard input (stdin)
- Data from the file system
- Data from a socket (network input)
- Input from the user interface (UI)

You don’t need to validate function arguments in all functions, but always validate data from an untrusted source before passing that value to other functions in the software component. For example, don’t access environment variables directly (using `os.environ`) all over the code, but create a dedicated class that provides controlled access to environment variables. That class should validate the environment variable values (correct type, allowed values, allowed value range, etc.). If the validation fails, a default value can be returned or an error raised.

7.4.14.1: Validating Numbers

When validating numeric values, always validate that a value is in a specified range. For example, if you use an unvalidated number to check if a loop should end and that number is huge, it can cause a denial of service (DoS).

If a number should be an integer, don't allow floating-point values.

7.4.14.2: Validating Strings

When validating a string, always validate the maximum length of the string first. Only after that should additional validation be performed. Validating a long string using a regular expression can cause a regular expression denial of service (ReDoS). You should avoid crafting your own regular expressions for validation purposes. Instead, use a ready-made library that contains battle-tested code. Consider also using the Google RE2 library. It is safer than regular expression functionality provided by many language runtimes, and your code will be less susceptible to ReDoS attacks.

7.4.14.3: Validating Timestamps

Timestamps (or times or dates) are usually given as an integer or string. Apply needed validation to a timestamp/time/date value. For example, you can validate if a timestamp is in the future or past or if a timestamp is earlier or later than another timestamp.

7.4.14.4: Validating Arrays

When validating an array, you should validate the size of the array. It should not be too small or large. You can validate the uniqueness of values if needed. Also, after validating the size of the array, remember to validate each value separately.

7.4.14.5: Validating Objects

Validate an object by validating each attribute of the object separately. Remember to validate nested objects also.

7.4.14.6: Validating Files Uploaded to Server

- Ensure the file name extension of the uploaded file is one of the allowed extensions
- Ensure the file is not larger than a defined maximum size
- Check the uploaded file against viruses and malware
- If the uploaded file is compressed (e.g., a zip file) and you are going to unzip it, verify the following before unzipping:
    - Target path is acceptable
    - Estimated decompressed size is not too large
- When storing an uploaded file on the server side, pay attention to the following:
    - Do not use a file name supplied by the user, but use a new filename to store the file on the server
    - Do not let the user choose the path where the uploaded file is stored on the server

8: API Design Principles

This chapter presents design principles for both frontend-facing and inter-microservice APIs. First, frontend-facing API design is discussed, and then inter-microservice API design is covered.

8.1: Frontend Facing API Design Principles

Most frontend-facing APIs should be HTTP-based JSON-based RPC, REST, or GraphQL APIs. Use GraphQL especially when the API handles heavily nested resources or clients want to decide what fields queries should return. For subscription-based APIs, use Server-Sent Events (SSE) or GraphQL subscriptions, and for real-time bidirectional communication, use WebSocket. If you transfer a lot of data or binary data between the frontend and backend, consider implementing the API using gRPC and gRPC Web. gRPC Web is not covered in this book. gRPC uses Protocol Buffers to binary encode data and is thus more efficient than JSON encoding.

In the coming examples, we use *camelCase* for JSON property names as it is a de-facto standard in APIs. Additional benefit of using camelCase instead of *snake_case* is that the implementation programming language (Python) is not immediately revealed to clients. It is always good to keep implementation details as hidden as possible.

8.1.1: JSON-based RPC API Design Principle

> ***Design a JSON-based RPC API to perform a single action (procedure) for an API endpoint.***

As the name suggests, JSON-RPC APIs are for executing remote procedure calls using JSON-encoded payloads. The remote procedure argument is a JSON object in the HTTP request body. The remote procedure return value is a JSON object in the HTTP response body. A client calls a remote procedure by issuing an HTTP POST request where it specifies the procedure's name in the URL path and gives the argument for the remote procedure call in the request body in JSON.

Below is an example request for a translation service's *translate* procedure:

Figure 8.1

```
POST /translation-service/translate HTTP/1.1
Content-Type: application/json

{
  "text": "Ich liebe dich"
  "fromLanguage": "German",
  "toLanguage": "English"
}
```

The API server shall respond with an HTTP status code and include the procedure's response in the HTTP response body in JSON.

For the above request, you get the following response:

Figure 8.2

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "translatedText": "I love you"
}
```

A JSON-RPC specification exists that defines one way to create JSON-based RPC APIs. I do not follow that specification in the examples below because one can create a JSON-based RPC API in many ways. But as an example, the above example rewritten using the *JSON-RPC specification* would look like the following:

Figure 8.3

```
POST /translation-service HTTP/1.1
Content-Type: application/json

{
  "jsonrpc": "2.0",
  "method": "translate",
```

```
  "params": {
    "text": "Ich liebe dich"
    "fromLanguage": "German",
    "toLanguage": "English"
  }
  "id": 1
}
```

And the response would look like as follows:

Figure 8.4

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "jsonrpc": "2.0",
  "result": "I love you",
  "id": 1
}
```

Let's have another example with a *web-page-search-service*:

Figure 8.5

```
POST /web-page-search-service/search-web-pages HTTP/1.1
Content-Type: application/json

{
  "containingText": "Software design patterns"
}
```

Figure 8.6

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "url": "https://...",
    "title": "...",
    "date": "...",
    "contentExcerpt": "..."
  },
  More results here ...
]
```

You can create a complete service using JSON-based RPC instead of REST or GraphQL. Below are five remote procedures defined for a *sales-item-service*. The procedures are for basic CRUD operations. The benefit of using JSON-based RPC instead of REST, GraphQL, or gRPC is that you don't have to learn or use conventions of any specific technology.

Figure 8.7

```
POST /sales-item-service/create-sales-item HTTP/1.1
Content-Type: application/json

{
  "name": "Sample sales item",
  "price": 20
}
```

Figure 8.8

```
POST /sales-item-service/get-sales-items HTTP/1.1
```

Figure 8.9

```
POST /sales-item-service/get-sales-item-by-id HTTP/1.1
Content-Type: application/json

{
  "id": 1
}
```

Figure 8.10

```
POST /sales-item-service/update-sales-item HTTP/1.1
Content-Type: application/json

{
  "id": 1,
  "name": "Sample sales item name modified",
  "price": 30
}
```

Figure 8.11

```
POST /sales-item-service/delete-sales-item-by-id HTTP/1.1
Content-Type: application/json

{
  "id": 1
}
```

Figure 8.12

```
POST /sales-item-service/delete-sales-items HTTP/1.1
```

You can easily create API endpoints for the above service. Below, they are implemented with FastAPI.

Figure 8.13

```
from typing import Any

from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

# In production code, add additional validations
```

```
# to DTO attributes, like maximum length for the 'name'
class InputSalesItem(BaseModel):
    name: str
    price: int


class OutputSalesItem(BaseModel):
    id: int
    name: str
    price: int


class SalesItemUpdate(InputSalesItem):
    id: int


class Id(BaseModel):
    id: int


@app.post("/sales-item-service/create-sales-item",
response_model=OutputSalesItem)
async def create_sales_items(sales_item: InputSalesItem) -> Any:
    # ...

@app.post("/sales-item-service/get-sales-items",
response_model=list[OutputSalesItem])
async def get_sales_items() -> Any:
    # ...

@app.post("/sales-item-service/get-sales-item-by-id",
response_model=OutputSalesItem)
async def get_sales_item_by_id(id: Id) -> Any:
    # ...

@app.post("/sales-item-service/update-sales-item",
response_model=OutputSalesItem)
async def update_sales_item(sales_item_update: SalesItemUpdate)
-> Any:
    # ...

@app.post("/sales-item-service/delete-sales-item-by-id")
async def delete_sales_item_by_id(id: Id) -> None:
    # ...
```

You can version your API by adding a version number to the URL. In the below example, the new API version 2 allows a new procedure argument `someNewParam` to be supplied for the `search-web-pages` procedure.

Figure 8.14

```
POST /web-page-search-service/v2/search-web-pages HTTP/1.1
Content-Type: application/json

{
  "containingText": "Software design patterns"
  "someNewParam": "..."
}
```

8.1.2: REST API Design Principle

Design a REST API for interaction with a resource (or resources) using CRUD (create, read, update, delete) operations.

Many APIs fall into the category of performing CRUD operations on resources. Let's create an example REST API called *sales-item-service* for performing CRUD operations on sales items. You can also define non-CRUD endpoints for a REST API. For example, you can define some JSON-based RPC endpoints if needed.

You can also remodel an RPC-style API to support CRUD operations. Suppose you need to create an API for starting and stopping some processes. Instead of creating a JSON-based RPC API with `start-process` and `stop-process` procedures, you can create a CRUD-based REST API where you create a resource to start a process and delete a resource to stop a process, i.e., a process is a resource you can perform CRUD operations on.

8.1.2.1: Creating a Resource

Creating a new resource using a REST API is done by sending an HTTP POST request to the API's resource endpoint. The API's resource endpoint should be named according to the resources it handles. The resource endpoint name should be a noun and always given in the plural form, for example, for the *sales-item-service* handling sales items, the resource endpoint should be *sales-items*, and for an *order-service* handling orders, the resource endpoint should be called *orders*.

You give the resource to be created in the HTTP request body in JSON. To create a new sales item, you can issue the following request:

Figure 8.15

```
POST /sales-item-service/sales-items HTTP/1.1
Content-Type: application/json

{
```

```
  "name": "Sample sales item",
  "price": 20
}
```

The server will respond with the HTTP status code 201 *Created*. The server can add properties to the resource upon creation. Typically, the server will add an `id` property to the created resource but can also add other properties. The server will respond with the created resource in the HTTP response body in JSON. Below is a response to the sales item creation request. You can notice that the server added the `id` property to the resource. Other properties that are usually added are the creation timestamp and the version of the resource (the version of a newly created resource should be one).

Figure 8.16

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 1,
  "name": "Sample sales item",
  "price": 20
}
```

If the supplied resource to be created is somehow invalid, the server should respond with the HTTP status code 400 *Bad Request* and explain the error in the response body. The response body should be in JSON format containing information about the error, like the error code and message.

To make API error responses consistent, use the same error response format throughout all the APIs in a software system. Below is an example of an error response:

Figure 8.17

```
{
  "statusCode": 500,
  "statusText": "Internal Server Error",
  "errorCode": "IAMError",
  "errorMessage": "Unable to connect to the Identity and Access
Management service"
  "errorDescription": "Describe the error in more detail here,
if relevant/needed..."
  "stackTrace": "Call stack trace here..."
}
```

NOTE! In the above example, the `stackTrace` property should NOT be included in the production environment by default because it can reveal internal implementation details to possible attackers. Use it only in development and other internal environments, and if needed, enable it in the production environment only for a short time to conduct debugging. The `errorCode` property is useful for updating error counter metric(s). Use it as a label for the error counter(s). There will be more discussion about metrics in the coming *DevSecOps principles* chapter.

If the created resource is huge, there is no need to return the resource to the caller and waste network bandwidth. You can return the added properties only. For example, if the server only adds the `id` property, it is possible to return only the `id` in the response body as follows:

Figure 8.18

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 1
}
```

The request sender can construct the created resource by merging the sent resource object with the received resource object.

Ensure that no duplicate resources are created.

When a client tries to create a new resource, the resource creation request may fail so that the resource was created successfully on the server, but the client did not receive a response on time, and the request failed due to timeout. From the server's point of view, the request was successful, but from the client's point of view, the request's status was indeterminate. The client, of course, needs to re-issue the time-outed request, and if it succeeds, the same resource is created twice on the server side (with two distinct IDs), which is probably unwanted in most cases.

Suppose a resource contains a unique property, like a user's email. In that case, it is impossible to create a duplicate resource if the server is correctly implemented (= the unique property is marked as a unique column in the database table definition). In many cases, such a unique field does not exist in the resource. In those cases, the client can supply a universally unique identifier (UUID), like`creationUuid`. The role of the server is to check if a resource with the same `creationUuid` was already created and to fail the creation of a duplicate resource. As an alternative to the UUID approach, the server can ask for verification from the client if the creation of two identical resources is intended in case the server receives two identical resources from the same client in a short period of time.

8.1.2.2: Reading Resources

Reading resources with a REST API is done by sending an HTTP GET request to the API's resource endpoint. To read all sales items, you can issue the following request:

Figure 8.19

```
GET /sales-item-service/sales-items HTTP/1.1
```

The server will respond with the HTTP status code 200 *OK* and a JSON array of resources in the response body or an empty array if none is found. Below is an example response to a request to get the sales items:

Figure 8.20

```
HTTP/1.1 200 OK
Content-Type: application/json

[
  {
    "id": 1,
    "name": "Sample sales item",
    "price": 20
  }
]
```

To read a single resource by its id, add the resource's id to the request URL as follows:

Figure 8.21

```
GET /sales-item-service/sales-items/<id> HTTP/1.1
```

The following request can be issued to read the sales item identified with id 1:

Figure 8.22

```
GET /sales-item-service/sales-items/1 HTTP/1.1
```

The response to the above request will contain a single resource:

Figure 8.23

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "id": 1,
  "name": "Sample sales item",
  "price": 20
}
```

The server responds with the HTTP status code 404 *Not Found* if the requested resource is not found.

You can define parameters in the URL query string to filter what resources to read. A query string is the last part of the URL and is separated from the URL path by a question mark (?) character. A query string can contain one or more parameters separated by ampersand (&) characters. Each query string parameter has the following format: `<query-parameter-name>=<query-parameter-value>`. Below is an example request with two query parameters: *name-contains* and *price-greater-than*.

Figure 8.24

```
GET /sales-item-service/sales-items?name-contains=Sample&price-greater-than=10 HTTP/1.1
```

The above request gets sales items whose name contains the string *Sample* and whose price is greater than 10.

To define a filter, you can specify a query parameter in the following format: `<fieldName>[-<condition>]=<value>`, for example:

- `price=10`

- `price-not-equal=10`
- `price-less-than=10`
- `price-less-than-equal=10`
- `price-greater-than=10`
- `price-greater-than-equal=10`
- `name-starts-with=Sample`
- `name-ends-with=item`
- `name-contains=Sample`
- `createdAtTimestamp-before=2022-08-02T05:18:00Z`
- `createdAtTimestamp-after=2022-08-02T05:18:00Z`
- `images.url-starts-with=https`

Remember that when implementing the server side and adding the above-given parameters to an SQL query, you must use a parameterized SQL query to prevent SQL injection attacks because an attacker can send malicious data in the query parameters.

Other actions like projection, sorting, and pagination for the queried resources can also be defined with query parameters in the URL:

Figure 8.25

```
GET /sales-item-service/sales-items?fields=id,name&sort-
by=price:asc&offset=0&limit=100 H\
TTP/1.1
```

The above request gets sales items sorted by price (ascending). The number of fetched sales items is limited to 100. Sales items are fetched starting from the offset 0, and the response contains only fields *id* and *name* for each sales item.

The *fields* parameter defines what resource fields (properties) are returned in the response. The wanted fields are defined as a comma-separated list of field names. If you want to define sub-resource fields, those can be defined with the dot notation, for example:

Figure 8.26

```
fields=id,name,images.url
```

The *sort-by* query parameter defines sorting using the following format:

```
sort-by=<fieldName>:asc|desc,[<fieldName>:asc|desc]
```

For example:

Figure 8.27

```
sort-by=price:asc,images.rank:asc
```

In the above example, the resources are returned sorted first by ascending price and secondarily by image's rank.

The *limit* and *offset* parameters are used for pagination. The *limit* query parameter defines the maximum number of resources that can be returned. The *offset* query parameter specifies the offset from which resources are returned. You can also paginate sub-resources by giving the *offset* and *limit* in the form of `<sub-resource>:<number>`. Below is an example of using pagination query parameters:

Figure 8.28

```
offset=0&limit=50,images:5
```

The above query parameters define that the first page of 50 sales items is fetched, and each sales item contains the first five images of the sales item. Instead of *offset* and *limit* parameters, you can use *page* and *page-size* parameters. The *page* parameter defines the page number, and the *page-size* defines how many resources a page should contain.

Remember to validate user-supplied data to prevent SQL injection attacks when implementing the server side and adding data from URL query parameters to an SQL query. For example, field names in the *fields* query parameter should only contain characters allowed in an SQL column name. Similarly, the value of the *sort-by* parameter should only contain characters allowed in an SQL column name and words *asc* and *desc*. And finally, the values of the *offset* and *limit* (or *page* and *page-size*) parameters must be integers. You should also validate the *limit/page-size* parameter against the maximum

allowed value because you should not allow clients to fetch too many resources at a time.

Some HTTP servers log the URL of an HTTP GET request. For this reason, it is not recommended to put sensitive information in the URL. Sensitive information should be put into the request body. Also, browsers can have a limit for the maximum length of a URL. If you have a query string that is thousands of characters long, you should give parameters in the request body instead. You should not put a request body to an HTTP GET request. What you should do is issue the request using the HTTP POST method instead, for example:

Figure 8.29

```
POST /sales-item-service/sales-items HTTP/1.1
Content-Type: application/json
X-HTTP-Method-Override: GET

{
  "fields": ["name"],
  "sortBy": "price:asc",
  "limit": 100
}
```

The server can confuse the above request with a sales item creation request because the URL and the HTTP method are identical to a resource creation request. For this reason, a custom HTTP request header *X-HTTP-Method-Override* has been added to the request. The server should read the custom header and treat the above request as a GET request. The *X-HTTP-Method-Override* header tells the server to override the request method with the method supplied in the header.

8.1.2.3: Updating Resources

A resource is updated with a REST API by sending an HTTP PUT or PATCH request to the API's resource endpoint. To update the sales item identified with id 1, you can issue the following request:

Figure 8.30

```
PUT /sales-item-service/sales-items/1 HTTP/1.1
Content-Type: application/json

{
  "name": "Sample sales item name modified",
```

```
  "price": 30
}
```

The server will respond without content:

Figure 8.31

```
HTTP/1.1 204 No Content
```

Instead of sending no content, the server can return the updated resource in the response. This is needed if the server modifies the resource during the update process. The server will respond with the HTTP status code 404 *Not Found* if the requested resource is not found.

If the supplied resource in the request is invalid, the server should respond with the HTTP status code 400 *Bad Request*. The response body should contain an error object in JSON.

HTTP PUT request will replace the existing resource with the supplied resource. You can also modify an existing resource partially using the HTTP PATCH method:

Figure 8.32

```
PATCH /sales-item-service/sales-items/1 HTTP/1.1
Content-Type: application/json

{
  "price": 30
}
```

The above request only modifies the price property of the sales item identified with id 1. Other properties remain intact. You can do bulk updates by specifying a filter in the URL, for example:

Figure 8.33

```
PATCH /sales-item-service/sales-items?price-less-than=10
HTTP/1.1
Content-Type: application/json

{
  "price": 10
}
```

The above example will update the price property of each resource where the price is currently less than ten. On the server side, the API endpoint could use the following

parameterized SQL statement to implement the update functionality:

Figure 8.34

```
UPDATE salesitems SET price = %s WHERE price < %s
```

The above SQL statement will only modify the price column; other columns remain intact.

Use resource versioning when needed.

When you get a resource from the server and try to update it, someone else may have updated it after you got it but before trying to update it. This can be okay if you don't care about other clients' updates. But sometimes, you want to ensure no one else has updated the resource before you update it. In that case, you should use resource versioning. In the resource versioning, there is a version field in the resource, which is incremented by one during each update. If you get a resource with version *x* and then try to update the resource, giving back the same version *x* to the server, but someone else has updated the resource to version *x + 1*, your update will fail because of the version mismatch (*x* != *x + 1*). The server should respond with the HTTP status code 409 *Conflict*. After receiving the conflict response, you can fetch the latest version of the resource from the server and, based on the resource's new state, decide whether your update is still relevant or not, and retry the update.

The server should assign the resource version value to the HTTP response header ETag. A client can use the received ETag value in a conditional HTTP GET request by assigning the received ETag value to the request header If-None-Match. The server will return the requested resource only if it has a newer version. Otherwise, the server returns nothing with the HTTP status code 304 *Not Modified*. This brings the advantage of not re-transferring an unmodified resource from the server to the client, which can be especially beneficial when the resource is large, or the connection between the server and the client is slow.

8.1.2.4: Deleting Resources

Deleting a resource with a REST API is done by sending an HTTP DELETE request to the API's resource endpoint. To delete the sales item identified with id 1, you can issue the following request:

Figure 8.35

```
DELETE /sales-item-service/sales-items/1 HTTP/1.1
```

The server will respond without content:

Figure 8.36

```
HTTP/1.1 204 No Content
```

If the requested resource has already been deleted, the API should still respond with the HTTP status code 204 *No Content*, meaning a successful operation. It should not respond with the HTTP status code 404 *Not Found*.

To delete all sales items, you can issue the following request:

Figure 8.37

```
DELETE /sales-item-service/sales-items HTTP/1.1
```

To delete sales items using a filter, you can issue the following kind of request:

Figure 8.38

```
DELETE /sales-item-service/sales-items?price-less-than=10
HTTP/1.1
```

On the server side, the API endpoint handler can use the following parameterized SQL query to implement the deleting functionality:

Figure 8.39

```
DELETE FROM salesitems WHERE price < %s
```

8.1.2.5: Executing Non-CRUD Actions on Resources

Sometimes, you need to perform non-CRUD actions on resources. In those cases, you can issue an HTTP POST request and put the name of the action (a verb) after the

resource name in the URL. The below example will perform a *deposit* action on an account resource:

Figure 8.40

```
POST /account-balance-service/accounts/12345678912/deposit
HTTP/1.1
Content-Type: application/json

{
  "amountInCents": 2510
}
```

Similarly, you can perform a withdrawal action:

Figure 8.41

```
POST /account-balance-service/accounts/12345678912/withdraw
HTTP/1.1
Content-Type: application/json

{
  "amountInCents": 2510
}
```

8.1.2.6: Resource Composition

A resource can be composed of other resources. There are two ways to implement resource composition: Nesting resources or linking resources. Let's have an example of nesting resources first. A sales item resource can contain one or more image resources. We don't want to return all images when a client requests a sales item because images can be large and are not necessarily used by the client. What we could return is a set of small thumbnail images. For a client to get the full images of a sales item, we could implement an API endpoint for image resources. The following API call can be issued to get images for a specific sales item:

Figure 8.42

```
GET /sales-item-service/sales-items/<id>/images HTTP/1.1
```

You can also add a new image for a sales item:

Figure 8.43

```
POST /sales-item-service/sales-items/<id>/images HTTP/1.1
```

Also, other CRUD operations could be made available:

Figure 8.44

```
PUT /sales-item-service/sales-
items/<salesItemId>/images/<imageId> HTTP/1.1
```

Figure 8.45

```
DELETE /sales-item-service/sales-
items/<salesItemId>/images/<imageId> HTTP/1.1
```

The problem with this approach is that the *sales-item-service* will grow in size, and if you need to add more nested resources in the future, the size will grow even more, making the microservice too complex and being possibly responsible for too many things.

A better alternative might be to create a separate microservice for the nested resources. This will enable the utilization of the best-suited technologies to implement the microservice. Regarding the sales item images, the *sales-item-image-service* could employ a cloud object storage to store images, and the *sales-item-service* could utilize a standard relational database for storing sales items.

When having a separate microservice for sales item images, you can get the images for a sales item by issuing the following request:

Figure 8.46

```
GET /sales-item-image-service/sales-item-images?salesItemId=
<salesItemId> HTTP/1.1
```

You can notice that the *sales-item_service* and *sales-item-image-service* are now linked by the *salesItemId*.

Note that the *sales-item-image-service* should be a service aggregated by the *sales-item-service*. The higher-level *sales-item-service* calls the lower-level *sales-item-image-service* because a sales item is a root aggregate, and sales item images are child entities that should not be accessed directly but only via the root aggregate, according to DDD. This helps with enforcing business rules. For example, let's hypothesize that a particular type of sales item should have at least *x* images. This kind of business rule should be enforced by the *sales-item-service*. The *sales-item-image-service* cannot do it because it does not have (and should not have) detailed

information about the sales item itself. It only has the sales item's id.

8.1.2.7: HTTP Status Codes

Use the following HTTP status codes:

HTTP Status Code	When to Use
200 OK	Successful API operations with the GET method
201 Created	Successful API operations with the POST method
202 Accepted	The request has been accepted for processing. This can be used as the response status code for an asynchronous operation request. For example, a POST request can get a response with this status code and a link to the resource that will eventually be created. That link will return 404 *Not Found* until the asynchronous creation is complete.
204 No Content	Successful API operations with the PUT, PATCH, or DELETE method
400 Bad Request	Client error in API operations, e.g., invalid data supplied by the client
401 Unauthorized	Client does not provide an authorization header in the request
403 Forbidden	Client provides an authorization header in the request, but the user is not authorized to perform the API operation
404 Not Found	When requesting a non-existent resource with the GET, PUT, or PATCH method
405 Method Not Allowed	When a client tries to use the wrong method for an API endpoint
406 Not Acceptable	When a client requests a response in a format that the server cannot produce, e.g., requests XML, but the server provides only JSON
409 Conflict	When a client is trying to update a resource that has been updated after the client got the resource
413 Payload Too Large	When a client tries to supply a too large payload in a request. To prevent DoS attacks, do not accept arbitrarily large payloads from clients

HTTP Status Code	When to Use
429 Too Many Requests	Configure rate limiting in your API gateway to send this status code when the request rate is exceeded
500 Internal Server Error	When a server error occurs, for example, an exception is thrown
503 Service Unavailable	Server's connections to dependent services fail. This indicates that clients should retry the request after a while because this issue is usually temporary.

8.1.2.8: HATEOAS and HAL

Hypermedia as the Engine of Application State (HATEOAS) can be used to add hypermedia/metadata to a requested resource. Hypertext Application Language (HAL) is a convention for defining hypermedia (metadata), such as links to external resources. Below is an example response to a request that fetches the sales item with id 1234. The sales item is owned by the user with id 5678. The response provides a link to the fetched resource itself and another link to fetch the user (account) that owns the sales item:

Figure 8.47

```
{
  "_links": {
    "self": {
      "href": "https://.../sales-item-service/sales-items/1234"
    },
    "userAccount": {
      "href": "https://.../user-account-service/user-
accounts/5678"
    }
  },
  "id": 1234,
  "name": "Sales item xyz"
  "userAccountId": 5678
}
```

When fetching a collection of sales items for page 3 using HAL, we can get the following kind of response:

Figure 8.48

```
{
  "_links": {
```

```
    "self": {
      "href": "https://.../sales-items?page=3"
    },
    "first": {
      "href": "https://...sales-items"
    },
    "prev": {
      "href": "https://.../sales-items?page=2"
    },
    "next": {
      "href": "https://.../sales-items?page=4"
    },
  },
  "count": 25,
  "total": 1500,
  "_embedded": {
    "salesItems": [
      {
        "_links": {
           "self": {
             "href": "https://.../sales-items/123"
           }
        },
        "id": 123,
        "name": "Sales item 123"
      },
      {
        "_links": {
           "self": {
             "href": "https://.../sales-items/124"
           }
        },
        "id": 124,
        "name": "Sales item 124"
      },
      .
      .
      .
    ]
  }
}
```

The above response contains links to fetch the first, current, previous, and last page of sales items. It also contains information that there are 1500 sales items, and a page lists 25. The `_embedded` property contains a `salesItems` property containing the 25 sales items with links to themselves and the sales item data.

8.1.2.9: API Versioning

You can introduce a new version of an API using a versioning URL path segment. Below are example endpoints for API version 2:

Figure 8.49

```
GET /sales-item-service/v2/sales-items HTTP/1.1
...
```

8.1.2.10: Documentation

If you need to document or provide interactive online documentation for a REST API, there are two ways:

1. Spec-first: create a specification for the API and then generate code from the specification
2. Code-first: implement the API and then generate the API specification from the code

Tools like Swagger and Postman can generate both static and interactive documentation for your API based on the API specification. You should specify APIs using the OpenAPI specification.

When using the first alternative, you can specify your API using the OpenAPI specification language. You can use tools like SwaggerHub or Postman to write the API specification. Swagger Codegen offers code-generation tools for multiple languages. Code generators generate code based on the OpenAPI specification. They can generate client-side code in addition to the server-side code.

When using the second alternative, you can use a web framework-specific way to generate the API specification from the API implementation. For example, with FastAPI, you get automatic API specification generation. By default, the OpenAPI schema is served in JSON format at the following endpoint: `/openapi.json`. This URL is configurable. FastAPI also provides Swagger UI interactive docs and client at the following URL: `/docs`. This URL is also customizable. ReDoc interactive docs are available at `/redoc`, which is also configurable. For example, setting OpenAPI schema to be served at `/my-service/v1/openapi.json` and setting Swagger UI to be served at `/my-service/v1/docs` and disabling ReDoc can be achieved in the following way:

Figure 8.50

```
from fastapi import FastAPI

app = FastAPI(
    openapi_url='/my-service/v1/openapi.json',
    docs_url='/my-service/v1/docs',
    redoc_url=None
)
```

I prefer to use the second approach of writing the code first. I like it better when I don't have to work with both auto-generated and handwritten code. Many web frameworks offer automatic generation of the OpenAPI schema and interactive documentation from the source code.

8.1.2.11: Implementation Example

Let's implement *sales-item-service* API endpoints for CRUD operations on sales items using FastAPI. We use the *clean microservice design principle* introduced earlier and write the API endpoints inside a controller class:

Figure 8.51

```
from dependency_injector.wiring import Provide
from fastapi import APIRouter, Request

from ..decorators.audit_log import audit_log
from ..decorators.increment_counter import increment_counter
from ..dtos.InputSalesItem import InputSalesItem
from ..dtos.OutputSalesItem import OutputSalesItem
from ..service.SalesItemService import SalesItemService


class RestSalesItemController:
    # Sales item service is provided by dependency injection
    __sales_item_service: SalesItemService =
Provide['sales_item_service']

    def __init__(self):
        self.__router = APIRouter()

        self.__router.add_api_route(
            '/sales-items/',
            self.create_sales_item,
            methods=['POST'],
            status_code=201,
            response_model=OutputSalesItem,
        )

        self.__router.add_api_route(
            '/sales-items/',
            self.get_sales_items,
            methods=['GET'],
            response_model=list[OutputSalesItem],
```

```
        )

        self.__router.add_api_route(
            '/sales-items/{id_}',
            self.get_sales_item,
            methods=['GET'],
            response_model=OutputSalesItem,
        )

        self.__router.add_api_route(
            '/sales-items/{id_}',
            self.update_sales_item,
            methods=['PUT'],
            status_code=204,
            response_model=None,
        )

        self.__router.add_api_route(
            '/sales-items/{id_}',
            self.delete_sales_item,
            methods=['DELETE'],
            status_code=204,
            response_model=None,
        )

    @property
    def router(self):
        return self.__router

    def create_sales_item(
        self, input_sales_item: InputSalesItem
    ) -> OutputSalesItem:
        return self.__sales_item_service.create_sales_item(
            input_sales_item
        )

    def get_sales_items(self) -> list[OutputSalesItem]:
        return self.__sales_item_service.get_sales_items()

    def get_sales_item(self, id_: str) -> OutputSalesItem:
        return self.__sales_item_service.get_sales_item(id_)

    def update_sales_item(
        self, id_: str, sales_item_update: InputSalesItem
    ) -> None:
        return self.__sales_item_service.update_sales_item(
            id_, sales_item_update
        )

    def delete_sales_item(self, id_: str, request: Request) ->
None:
        return self.__sales_item_service.delete_sales_item(id_)
```

The above controller is not production quality. The following must be added:

- Audit logging

- Observability, e.g., updating metric(s)
- Authorization

All of the above could be and probably should be implemented using decorators, for example:

Figure 8.52

```
@allow_for_user_roles(['admin'], authorizer)
@audit_log
@increment_counter(Counters.request_attempts)
def create_sales_item(
    self,
    input_sales_item: InputSalesItem,
    request: Request
) -> OutputSalesItem:
    return self.__sales_item_service.create_sales_item(
        input_sales_item
    )
```

The authorization decorator `@allow_for_user_roles(['admin'], authorizer)` is the same as discussed in the previous chapter. The `@audit_log` decorator can be implemented in the following way:

Figure 8.53

```
from functools import wraps


def audit_log(handle_request):
    @wraps(handle_request)
    def wrapped_handle_request(*args, **kwargs):
        method = kwargs['request'].method
        url = kwargs['request'].url
        client_host = kwargs['request'].client.host
        # The below printed text should be written to audit log
        print(f'API endpoint: {method} {url} accessed from:
{client_host}')
        return handle_request(*args, **kwargs)

    return wrapped_handle_request
```

The `@increment_counter` decorator can be implemented in the following way:

Figure 8.54

```
from functools import wraps


def increment_counter(counter):
    def decorate(handle_request):
        @wraps(handle_request)
        def wrapped_handle_request(*args, **kwargs):
            method = kwargs['request'].method
```

```
            url = kwargs['request'].url
            # Increment counter by one with 'api_endpoint' label
            counter.increment(1, {'api_endpoint': f'{method}
{url}'})
            return handle_request(*args, **kwargs)

        return wrapped_handle_request

    return decorate
```

Notice how the above decorators are general purpose and not specific to this *sales-item-service* API. Instead of adding the decorators to the controller class methods, you might be better off creating decorators that can be added to the service class methods. In that case, you need to supply the needed information from the controller methods to the service methods, like the client's host for the audit logging decorator and the JWT for the authorization decorators. You can group these two into a `ClientInfo` object passed from the controller to the service class. The service class decorators then operate with that object.

The DTOs (objects that specify what data is transferred (input or output) between clients and the server) are defined using pydantic validation library:

Figure 8.55

```
from pydantic import BaseModel, HttpUrl, PositiveInt

from ..entities.SalesItemImage import (
    SalesItemImage as SalesItemImageEntity,
)


class SalesItemImage(BaseModel):
    id: PositiveInt
    rank: PositiveInt
    url: HttpUrl

    class Config:
        orm_mode = True

    class Meta:
        orm_model = SalesItemImageEntity
```

Figure 8.56

```
from pydantic import BaseModel, Field

from .SalesItemImage import SalesItemImage


class InputSalesItem(BaseModel):
```

```
    name: str = Field(max_length=256)
    # We accept negative prices for sales items that act
    # as discount items
    priceInCents: int
    images: list[SalesItemImage] = Field(max_items=25)

    class Config:
        orm_mode = True
```

Figure 8.57

```
from pydantic import BaseModel, Field, PositiveInt

from .SalesItemImage import SalesItemImage


class OutputSalesItem(BaseModel):
    id: str
    createdAtTimestampInMs: PositiveInt
    name: str = Field(max_length=256)
    priceInCents: int
    images: list[SalesItemImage] = Field(max_items=25)

    class Config:
        orm_mode = True
```

Notice that we have specified validation for each attribute in all three DTO classes. This is important because of security. For example, string and list attributes should have maximum length validators to prevent possible denial of service attacks. Output DTOs should have validation as well. This is important because of security. Output validation can protect against injection attacks that try to return data that has an invalid shape. In FastAPI, output DTOs are also used in the automatic documentation of the API schema and when automatically generating client code.

The `SalesItemService` protocol looks like the following:

Figure 8.58

```
from typing import Protocol

from ..dtos.InputSalesItem import InputSalesItem
from ..dtos.OutputSalesItem import OutputSalesItem


class SalesItemService(Protocol):
    def create_sales_item(
        self, input_sales_item: InputSalesItem
    ) -> OutputSalesItem:
        pass

    def get_sales_items(self) -> list[OutputSalesItem]:
        pass
```

```
    def get_sales_item(self, id_: str) -> OutputSalesItem:
        pass

    def update_sales_item(
        self, id_: str, sales_item_update: InputSalesItem
    ) -> None:
        pass

    def delete_sales_item(self, id_: str) -> None:
        pass
```

Next, we can implement the above protocol:

Figure 8.59

```
from dependency_injector.wiring import Provide

from ..dtos.InputSalesItem import InputSalesItem
from ..dtos.OutputSalesItem import OutputSalesItem
from ..errors.EntityNotFoundError import EntityNotFoundError
from ..repositories.SalesItemRepository import
SalesItemRepository
from ..service.SalesItemService import SalesItemService


class SalesItemServiceImpl(SalesItemService):
    # Sales item repository is provided by DI
    __sales_item_repository: SalesItemRepository = Provide[
        'sales_item_repository'
    ]

    def create_sales_item(
        self, input_sales_item: InputSalesItem
    ) -> OutputSalesItem:
        sales_item =
self.__sales_item_repository.save(input_sales_item)
        return OutputSalesItem.from_orm(sales_item)

    def get_sales_items(self) -> list[OutputSalesItem]:
        return [
            OutputSalesItem.from_orm(sales_item)
            for sales_item in
self.__sales_item_repository.find_all()
        ]

    def get_sales_item(self, id_: str) -> OutputSalesItem:
        sales_item = self.__sales_item_repository.find(id_)
        if sales_item is None:
            raise EntityNotFoundError('Sales item', id_)
        return OutputSalesItem.from_orm(sales_item)

    def update_sales_item(
        self, id_: str, sales_item_update: InputSalesItem
    ) -> None:
        return self.__sales_item_repository.update(id_,
sales_item_update)

    def delete_sales_item(self, id_: str) -> None:
        return self.__sales_item_repository.delete(id_)
```

Below is the definition of the `SalesItemRepository` protocol:

Figure 8.60

```
from typing import Protocol

from ..dtos.InputSalesItem import InputSalesItem
from ..entities.SalesItem import SalesItem


class SalesItemRepository(Protocol):
    def save(self, input_sales_item: InputSalesItem) ->
SalesItem:
        pass

    def find_all(self) -> list[SalesItem]:
        pass

    def find(self, id_: str) -> SalesItem | None:
        pass

    def update(self, id_: str, sales_item_update:
InputSalesItem) -> None:
        pass

    def delete(self, id_: str) -> None:
        pass
```

Various implementations for the `SalesItemRepository` are presented in the next chapter, where we focus on database principles. The next chapter provides three different implementations for the repository: Object-Relational Mapping (ORM), parameterized SQL queries, and MongoDB.

For error handling, we depend on the `except` block provided by the FastAPI web framework. We could raise errors of the FastAPI's `HTTPException` type in our business logic, but then we would be coupling our web framework with business logic, which is not desired. Remember how in the *clean microservice design principle*, the dependency goes only from the web framework (controller) towards business logic, not vice versa. If we used web framework-specific error classes in our business and logic, and we would like to migrate the microservice to a different web framework, we would have to refactor the whole business logic concerning raised errors.

What we should do is introduce a base error class for our microservice and provide a custom error handler for FastAPI. The custom error handler translates our business logic-specific errors into HTTP responses. The possible errors the

microservice can raise should all derive from the base error class. The `ApiError` class below is a general-purpose base error class for any API.

Figure 8.61

```
from typing import Final


class ApiError(Exception):
    def __init__(
        self,
        status_code: int,
        status_text: str,
        message: str,
        code: str | None = None,
        description: str | None = None,
        cause: Exception | None = None,
    ):
        self.__status_code: Final = status_code
        self.__status_text: Final = status_text
        self.__message: Final = message
        self.__code: Final = code
        self.__description: Final = description
        self.__cause: Final = cause

    @property
    def status_code(self) -> int:
        return self.__status_code

    @property
    def status_text(self) -> str:
        return self.__status_text

    @property
    def message(self) -> str:
        return self.__message

    @property
    def cause(self) -> Exception | None:
        return self.__cause

    @property
    def code(self) -> str | None:
        return self.__code

    @property
    def description(self) -> str | None:
        return self.__description

    def __str__(self) -> str:
        return self.__message
```

The `code` property could also be named `type`. The idea behind that property is to tell what kind of an error is in question. This property can be used on the server side as a label for failure metrics, and on the client side, special handling for particular

error codes can be implemented. If you want, you can even add one more property to the above class, namely `recovery_action`. This optional property contains information about recovery steps for an actionable error. For example, a database connection error might have a `recovery_action` property value: *Please retry after a while. If the problem persists, contact the technical support at .*

Below is the base error class for the *sales-item-service*:

Figure 8.62

```
from ..errors.ApiError import ApiError


class SalesItemServiceError(ApiError):
    pass
```

Let's then define one error class that is used by the API:

Figure 8.63

```
from .SalesItemServiceError import SalesItemServiceError


class EntityNotFoundError(SalesItemServiceError):
    def __init__(self, entity_name: str, entity_id: str):
        super().__init__(
            404,
            'Not Found',
            f'{entity_name} with id {entity_id} not found',
            'EntityNotFound',
        )
```

Let's implement a custom error handler for our API. Notice how the handler is general purpose and can be used with any API having its errors derived from the `ApiError`.

Figure 8.64

```
# Imports ...

app = FastAPI()

@app.exception_handler(SalesItemServiceError)
def handle_sales_item_service_error(
    request: Request, error: ApiError
):
    # Log error.cause

    # Increment 'request_failures' counter by one
    # with three labels:
    # api_endpoint=f'{request.method} {request.url}'
    # status_code=error.status_code
    # error_code=error.code
```

```
    return JSONResponse(
        status_code=error.status_code,
        content={
            'statusCode': error.status_code,
            'statusText': error.status_text,
            'errorCode': error.code,
            'errorMessage': error.message,
            'errorDescription': error.description,
            # get_stack_trace returns stack trace only
            # when environment is not production
            # otherwise it returns None
            'stackTrace': get_stack_trace(error.cause),
        },
    )
```

Now, if the business logic raises the following error:

Figure 8.65

```
raise EntityNotFoundError('Sales item', '10')
```

The following API response should be expected in a production environment (Notice how the `stackTrace` is null when the service is running in the production environment):

Figure 8.66

```
HTTP/1.1 404 Not Found
Content-Type: application/json

{
  "statusCode": 404,
  "statusText": "Not Found",
  "errorCode": "EntityNotFound",
  "errorMessage": "Sales item with id 10 not found",
  "errorDescription": null,
  "stackTrace": null
}
```

You should also add specific error handlers for DTO validation errors and other possible errors:

Figure 8.67

```
@app.exception_handler(RequestValidationError)
def handle_request_validation_error(
    request: Request, error: RequestValidationError
):
    # Audit log

    # Increment 'request_failures' counter by one
    # with three labels:
    # api_endpoint=f'{request.method} {request.url}'
    # status_code=400
    # error_code='RequestValidationError'

    return JSONResponse(
        status_code=400,
```

```
        content={
            'statusCode': 400,
            'statusText': 'Bad Request',
            'errorCode': 'RequestValidationError',
            'errorMessage': 'Request validation failed',
            'errorDescription': str(error),
            'stackTrace': None,
        },
    )


@app.exception_handler(Exception)
def handle_unspecified_error(request: Request, error:
Exception):
    # Increment 'request_failures' counter by one
    # with labels:
    # api_endpoint=f'{request.method} {request.url}'
    # status_code=500
    # error_code='UnspecifiedError'

    return JSONResponse(
        status_code=500,
        content={
            'statusCode': 500,
            'statusText': 'Internal Server Error',
            'errorCode': 'UnspecifiedError',
            'errorMessage': 'Unspecified internal error',
            'errorDescription': str(error),
            'stackTrace': get_stack_trace(error),
        },
    )
```

The rest of the API service source code files look like the following:

Figure 8.68

```
from dependency_injector import containers, providers

from .controllers.RestSalesItemController import
RestSalesItemController
from .controllers.StrawberryGraphQlSalesItemController import (
    StrawberryGraphQlSalesItemController,
)
from .repositories.MongoDbSalesItemRepository import (
    MongoDbSalesItemRepository,
)
from .repositories.OrmSalesItemRepository import
OrmSalesItemRepository
from .repositories.ParamSqlSalesItemRepository import (
    ParamSqlSalesItemRepository,
)
from .service.SalesItemServiceImpl import SalesItemServiceImpl


class DiContainer(containers.DeclarativeContainer):
    wiring_config = containers.WiringConfiguration(
        modules=[
            '.service.SalesItemServiceImpl',
            '.controllers.RestSalesItemController',
```

```
            '.controllers.AriadneGraphQlSalesItemController',
            '.controllers.StrawberryGraphQlSalesItemController',
            '.controllers.GrpcSalesItemController',
            '.repositories.OrmSalesItemRepository',
            '.repositories.ParamSqlSalesItemRepository',
            '.repositories.MongoDbSalesItemRepository',
        ]
    )

    sales_item_service =
providers.Singleton(SalesItemServiceImpl)
    sales_item_repository = providers.Singleton(
        ParamSqlSalesItemRepository
    )
    order_controller =
providers.Singleton(RestSalesItemController)
```

Figure 8.69

```
import os
import traceback

from pydantic import BaseModel


def is_pydantic(object: object):
    return type(object).__class__.__name__ == 'ModelMetaclass'


def to_entity_dict(dto: BaseModel):
    entity_dict = dict(dto)

    for key, value in entity_dict.items():
        try:
            if (
                isinstance(value, list)
                and len(value)
                and is_pydantic(value[0])
            ):
                entity_dict[key] = [
                    item.Meta.orm_model(**to_entity_dict(item))
                    for item in value
                ]
            elif is_pydantic(value):
                entity_dict[key] = value.Meta.orm_model(
                    **to_entity_dict(value)
                )
        except AttributeError:
            raise AttributeError(
                f'Found nested Pydantic model in {dto.__class__}
but Meta.orm_model was n\
ot specified.'
            )

    return entity_dict


def get_stack_trace(error: Exception | None):
    return (
        repr(traceback.format_exception(error))
        if error and os.environ.get('ENV') != 'production'
```

```
        else None
    )
```

Figure 8.70

```
import os

from fastapi import FastAPI, Request
from fastapi.exceptions import RequestValidationError
from fastapi.responses import JSONResponse

from .DiContainer import DiContainer
from .errors.SalesItemServiceError import SalesItemServiceError
from .utils import get_stack_trace

# Remove the below setting of the env variable for production
code!
# mysql+pymysql://root:password@localhost:3306/salesitemservice
# mongodb://localhost:27017/salesitemservice
os.environ[
    'DATABASE_URL'
] =
'mysql+pymysql://root:password@localhost:3306/salesitemservice'


di_container = DiContainer()
app = FastAPI()


@app.exception_handler(SalesItemServiceError)
def handle_sales_item_service_error(
    request: Request, error: SalesItemServiceError
):
    # Log error.cause

    # Increment 'request_failures' counter by one
    # with labels:
    # api_endpoint=f'{request.method} {request.url}'
    # status_code=error.status_code
    # error_code=error.code

    return JSONResponse(
        status_code=error.status_code,
        content={
            'statusCode': error.status_code,
            'statusText': error.status_text,
            'errorCode': error.code,
            'errorMessage': error.message,
            'errorDescription': error.description,
            # get_stack_trace returns stack trace only
            # when environment is not production
            # otherwise it returns None
            'stackTrace': get_stack_trace(error.cause),
        },
    )


@app.exception_handler(RequestValidationError)
def handle_request_validation_error(
    request: Request, error: RequestValidationError
):
```

```
    # Audit log

    # Increment 'request_failures' counter by one
    # with labels:
    # api_endpoint=f'{request.method} {request.url}'
    # status_code=400
    # error_code='RequestValidationError'

    return JSONResponse(
        status_code=400,
        content={
            'statusCode': 400,
            'statusText': 'Bad Request',
            'errorCode': 'RequestValidationError',
            'errorMessage': 'Request validation failed',
            'errorDescription': str(error),
            'stackTrace': None,
        },
    )


@app.exception_handler(Exception)
def handle_unspecified_error(request: Request, error:
Exception):

    # Increment 'request_failures' counter by one
    # with labels:
    # api_endpoint=f'{request.method} {request.url}'
    # status_code=500
    # error_code='UnspecifiedError'

    return JSONResponse(
        status_code=500,
        content={
            'statusCode': 500,
            'statusText': 'Internal Server Error',
            'errorCode': 'UnspecifiedError',
            'errorMessage': 'Unspecified internal error',
            'errorDescription': str(error),
            'stackTrace': get_stack_trace(error),
        },
    )


order_controller = di_container.order_controller()
app.include_router(order_controller.router)
```

8.1.3: GraphQL API Design

Divide API endpoints into queries and mutations. Compared to REST, REST GET requests are GraphQL queries, and REST POST/PUT/PATCH/DELETE requests are GraphQL mutations. With GraphQL, you can name your queries and mutations with descriptive names.

retry after a while. If the problem persists, The rest of the API service source code files look like the following: Let's create a GraphQL schema that defines needed types and API endpoints for the *sales-item-service*. After the example, we will discuss the details of the schema below and the schema language in general.

Figure 8.71

```
type Image {
  id: Int!
  rank: Int!
  url: String!
}

type SalesItem {
  id: ID!
  createdAtTimestampInMs: String!
  name: String!
  priceInCents: Int!
  images(
    sortByField: String = "rank",
    sortDirection: SortDirection = ASC,
    offset: Int = 0,
    limit: Int = 5
  ): [Image!]!
}

input InputImage {
  id: Int!
  rank: Int!
  url: String!
}

input InputSalesItem {
  name: String!
  priceInCents: Int!
  images: [InputImage!]!
}

enum SortDirection {
  ASC
  DESC
}

type IdResponse {
  id: ID!
}

type Query {
  salesItems(
    sortByField: String = "createdAtTimestamp",
    sortDirection: SortDirection = DESC,
    offset: Int = 0,
    limit: Int = 50
  ): [SalesItem!]!

  salesItem(id: ID!): SalesItem!
```

```
  salesItemsByFilters(
    nameContains: String,
    priceGreaterThan: Float
  ): [SalesItem!]!
}

type Mutation {
  createSalesItem(salesItem: InputSalesItem!): SalesItem!

  updateSalesItem(
    id: ID!,
    salesItem: InputSalesItem
  ): IdResponse!

  deleteSalesItem(id: ID!): IdResponse!
}
```

The above GraphQL schema defines several types used in API requests and responses. A GraphQL `type` specifies an object type: Its properties and the types of those properties. A type specified with the `input` keyword is an input-only type (input DTO type). GraphQL defines the primitive (scalar) types: `Int` (32-bit), `Float`, `String`, `Boolean`, and `ID`. You can define an array type with the notation: `[<Type>]`. By default, types are nullable. If you want a non-nullable type, add an exclamation mark (!) after the type name. You can define an enumerated type with the `enum` keyword. The `Query` and `Mutation` types are special GraphQL types used to define queries and mutations. The above example defines three queries and four mutations that clients can execute. You can add parameters for a type property. We have added parameters for all the queries (queries are properties of the `Query` type), mutations (mutations are properties of the `Mutation` type), and the `images` property of the `SalesItem` type.

In the above example, I have named all the queries with names that describe the values they return, i.e., there are no verbs in the query names. It is possible to name queries starting with a verb (like the mutations). For example, you can add *get* to the beginning of the names of the above-defined queries if you prefer.

There are two ways to implement a GraphQL API:

- Schema first
- Code first (schema is generated from the code)

Let's first focus on the schema-first approach and implement the above-specified API using the Ariadne library. Initially, we define fake implementations (returning static objects) for some of the API endpoints (queries/mutations):

Figure 8.72

```
import time

from ariadne import MutationType, QueryType, gql,
make_executable_schema
from ariadne.asgi import GraphQL

schema = gql(
    """
type Image {
  id: Int!
  rank: Int!
  url: String!
}

type SalesItem {
  id: ID!
  createdAtTimestampInMs: String!
  name: String!
  priceInCents: Int!
  images(
    sortByField: String = "rank",
    sortDirection: SortDirection = ASC,
    offset: Int = 0,
    limit: Int = 5
  ): [Image!]!
}

input InputImage {
  id: Int!
  rank: Int!
  url: String!
}

input InputSalesItem {
  name: String!
  priceInCents: Int!
  images: [InputImage!]!
}

enum SortDirection {
  ASC
  DESC
}

type IdResponse {
  id: ID!
}

type Query {
  salesItems(
    sortByField: String = "createdAtTimestampInMs",
    sortDirection: SortDirection = DESC,
    offset: Int = 0,
```

```
    limit: Int = 50
  ): [SalesItem!]!

  salesItem(id: ID!): SalesItem!

  salesItemsByFilters(
    nameContains: String,
    priceGreaterThan: Float
  ): [SalesItem!]!
}

type Mutation {
  createSalesItem(inputSalesItem: InputSalesItem!): SalesItem!

  updateSalesItem(
    id: ID!,
    inputSalesItem: InputSalesItem
  ): IdResponse!

  deleteSalesItem(id: ID!): IdResponse!
}
"""
)

query = QueryType()


@query.field('salesItems')
def resolve_sales_items(*, **kwargs):
    if kwargs['offset'] == 0:
        return [
            {
                'id': 1,
                'createdAtTimestampInMs': '12345678999877',
                'name': 'sales item',
                'priceInCents': 1095,
                'images': [{'id': 1, 'rank': 2, 'url': 'url'}],
            }
        ]
    return []


@query.field('salesItem')
def resolve_sales_item(*, id):
    return {
        'id': id,
        'createdAtTimestampInMs': '12345678999877',
        'name': 'sales item',
        'priceInCents': 1095,
        'images': [{'id': 1, 'rank': 2, 'url': 'url'}],
    }


mutation = MutationType()


@mutation.field('createSalesItem')
def resolve_create_sales_item(*, **kwargs):
    return {
        'id': 100,
        'createdAtTimestampInMs': str(round(time.time() *
1000)),
```

```
        **kwargs['inputSalesItem'],
    }


@mutation.field('deleteSalesItem')
def resolve_delete_sales_item(*, id):
    return {'id': id}


executable_schema = make_executable_schema(schema, [query,
mutation])

app = GraphQL(executable_schema)
```

In the above example, the `gql` function validates the schema and raises a descriptive `GraphQLSyntaxError` if there is an issue or returns the original string if it is correct. We created a resolver function for the first two queries in the schema and resolvers for creating and deleting a sales item. You can start the GraphQL server with the following command (You should have uvicorn already installed using *pip*):

Figure 8.73

```
uvicorn app:app
```

After running the server, browse to the URL: *http://127.0.0.1:8000/*. You will see the GraphiQL UI, where you can execute queries and mutations. Enter the following query to the left pane in the UI.

Figure 8.74

```
query salesItems {
  salesItems(offset: 0) {
    id
    createdAtTimestampInMs
    name
    priceInCents,
    images {
      url
    }
  }
}
```

You should get the following response on the right side pane:

Figure 8.75

```
{
  "data": {
    "salesItems": [
      {
        "id": "1",
        "createdAtTimestampInMillis": "12345678999877",
```

```
        "name": "sales item",
        "priceInCents": 1095,
        "images": [
          {
            "url": "url"
          }
        ]
      }
    ]
  }
}
```

You can also try to create a new sales item:

Figure 8.76

```
mutation create {
  createSalesItem(inputSalesItem: {
    priceInCents: 4095
    name: "test sales item"
    images: []
  }) {
    id,
    createdAtTimestampInMs,
    name,
    priceInCents,
    images {
      id
    },
  }
}
```

Below is the response you would get, except for the timestamp being the current time:

Figure 8.77

```
{
  "data": {
    "createSalesItem": {
      "id": "100",
      "createdAtTimestampInMillis": "1694798999418",
      "name": "test sales item",
      "priceInCents": 4095,
      "images": []
    }
  }
}
```

To delete a sales item, you can issue:

Figure 8.78

```
mutation delete {
  deleteSalesItem(id: 1) {
   id
  }
}
```

Figure 8.79

```
{
  "data": {
    "deleteSalesItem": {
      "id": "1"
    }
  }
}
```

Let's replace the dummy static implementations in our Ariadne GraphQL controller with actual calls to the sales item service:

Figure 8.80

```
from ariadne import MutationType, QueryType, gql,
make_executable_schema
from dependency_injector.wiring import Provide

from ..dtos.InputSalesItem import InputSalesItem
from ..service.SalesItemService import SalesItemService

sales_item_service: SalesItemService =
Provide['sales_item_service']

schema = gql(
    """
type Image {
  id: Int!
  rank: Int!
  url: String!
}

type SalesItem {
  id: ID!
  createdAtTimestampInMs: String!
  name: String!
  priceInCents: Int!
  images: [Image!]!
}

input InputImage {
  id: Int!
  rank: Int!
  url: String!
}

input InputSalesItem {
  name: String!
  priceInCents: Int!
  images: [InputImage!]!
}

type IdResponse {
  id: ID!
}

type Query {
  salesItems: [SalesItem!]!
```

```
  salesItem(id: ID!): SalesItem!
}

type Mutation {
  createSalesItem(inputSalesItem: InputSalesItem!): SalesItem!

  updateSalesItem(
    id: ID!,
    inputSalesItem: InputSalesItem
  ): IdResponse!

  deleteSalesItem(id: ID!): IdResponse!
}
"""
)

query = QueryType()


@query.field('salesItems')
def resolve_sales_items(*_):
    return sales_item_service.get_sales_items()


@query.field('salesItem')
def resolve_sales_item(*_, id: str):
    return sales_item_service.get_sales_item(id)


mutation = MutationType()


@mutation.field('createSalesItem')
def resolve_create_sales_item(*_, inputSalesItem):
    input_sales_item = InputSalesItem.parse_obj(inputSalesItem)
    return
sales_item_service.create_sales_item(input_sales_item)


@mutation.field('updateSalesItem')
def resolve_update_sales_item(*_, id: str, inputSalesItem):
    sales_item_update = InputSalesItem.parse_obj(inputSalesItem)
    sales_item_service.update_sales_item(id, sales_item_update)
    return {'id': id}


@mutation.field('deleteSalesItem')
def resolve_delete_sales_item(*_, id: str):
    sales_item_service.delete_sales_item(id)
    return {'id': id}


executable_schema = make_executable_schema(schema, [query,
mutation])
```

Notice in the above code that we must remember to validate the input for the two mutations. We can do that by converting the input dict to a Pydantic model using the `parse_obj`

method. We should add authorization, audit logging, and metric updates to make the example more production-like. This can be done by creating decorators in a similar way we created earlier in the REST API example. The decorators can get the request object from the `info.context` dict: `info.context['request']`.

GraphQL error handling differs from REST API error handling. A GraphQL API responses do not provide different HTTP response status codes. A GraphQL API response is always sent with the status code *200 OK*. When an error occurs while processing a GraphQL API request, the response body object includes an `errors` array. You should raise an error in your GraphQL type resolvers when a query or mutation fails. You can use the same `ApiError` base error class used in the earlier REST API example. As shown below, we need to add an error formatter to handle the custom API errors. The error objects should always have a `message` field. Additional information about the error can be supplied in an `extensions` object, which can contain any properties.

Suppose a `salesItem` query results in an `EntityNotFoundError`. Then the API response would have a `null` for the `data` property and `errors` property present, as shown below:

Figure 8.81

```
{
  "data": null,
  "errors": [
    {
      "message": "Sales item not found with id 1",
      "extensions": {
        "statusCode": 404,
        "statusText": "Not Found",
        "errorCode": "EntityNotFound",
        "errorDescription": null
        "stackTrace": null
      }
    }
  ]
}
```

Below is the code for the *app_graphql.py* module:

Figure 8.82

```
import os
from typing import Any
```

```python
from ariadne import format_error, unwrap_graphql_error
from ariadne.asgi import GraphQL
from pydantic import ValidationError

from .controllers.AriadneGraphQlSalesItemController import (
    executable_schema,
)
from .DiContainer import DiContainer
from .errors.SalesItemServiceError import SalesItemServiceError
from .utils import get_stack_trace

# Remove this setting of env variable for production code!
# mysql+pymysql://root:password@localhost:3306/salesitemservice
# mongodb://localhost:27017/salesitemservice
os.environ['DATABASE_URL'] =
'mongodb://localhost:27017/salesitemservice'


di_container = DiContainer()


def format_custom_error(
    graphql_error, debug: bool = False
) -> dict[str, Any]:
    error = unwrap_graphql_error(graphql_error)

    if isinstance(error, SalesItemServiceError):
        return {
            'message': error.message,
            'extensions': {
                'statusCode': error.status_code,
                'statusText': error.status_text,
                'errorCode': error.code,
                'errorDescription': error.description,
                'stackTrace': get_stack_trace(error.cause),
            },
        }

    if isinstance(error, ValidationError):
        return {
            'message': 'Request validation failed',
            'extensions': {
                'statusCode': 400,
                'statusText': 'Bad Request',
                'errorCode': 'RequestValidationError',
                'errorDescription': str(error),
                'stackTrace': None,
            },
        }

    if isinstance(error, Exception):
        return {
            'message': 'Unspecified internal error',
            'extensions': {
                'statusCode': 500,
                'statusText': 'Internal Server Error',
                'errorCode': 'UnspecifiedError',
                'errorDescription': str(error),
                'stackTrace': get_stack_trace(error),
            },
        }
```

```
    else:
        return format_error(graphql_error, debug)


app = GraphQL(executable_schema,
error_formatter=format_custom_error)
```

The Ariadne GraphQl version of the *sales-item-service* can be run with the below command. (We assume the service source code is placed in a package *salesitemservice*, and we are currently located in the parent directory of that).

Figure 8.83

```
uvicorn salesitemservice.app_graphql:app
```

As an alternative to the described error handling mechanism, it is also possible to return an error as a query/mutation return value. This can be done, e.g., by returning a union type from a query or mutation. This approach requires a more complex GraphQL schema and more complex resolvers on the server side. Here is an example:

Figure 8.84

```
# ...

type Error {
    message: String!
    # Other possible properties
}

union SalesItemOrError = SalesItem | Error

type Mutation {
  createSalesItem(inputSalesItem: InputSalesItem!):
SalesItemOrError!
}
```

In the *createSalesItem* query resolver, you must add a try-except block to handle an error situation and respond with an `Error` object in case of an error.

You can also specify multiple errors:

Figure 8.85

```
# ...

type ErrorType1 {
    # ...
}
```

```
type ErrorType2 {
    # ...
}

type ErrorType3 {
    # ...
}

union SalesItemOrError = SalesItem | ErrorType1 | ErrorType2 |
ErrorType3

type Mutation {
  createSalesItem(inputSalesItem: InputSalesItem!):
SalesItemOrError!
}
```

The above example would require making the *createSalesItem* resolvers to catch multiple different errors and responding with an appropriate error object as a result.

Also, the client-side code will be more complex because of the need to handle the different types of responses for a single operation (query/mutation). For example:

Figure 8.86

```
mutation {
  createSalesItem(inputSalesItem: {
    price: 200
    name: "test sales item"
    images: []
  }) {
    __typename
    ...on SalesItem {
      id,
      createdAtTimestampInMillis
    }
    ...on ErrorType1 {
      # Specify fields here
    }
    ...on ErrorType2 {
      # Specify fields here
    }
    ...on ErrorType3 {
      # Specify fields here
    }
}
```

This approach has a downside: the client must still be able to handle possible errors reported in the response's `errors` array.

In a GraphQL schema, you can add parameters for a primitive (scalar) property. That is useful for implementing conversions. For example, we could define the `SalesItem` type with a parameterized `priceInCents` property:

Figure 8.87

```
enum Currency {
  USD,
  GBP,
  EUR,
  JPY
}

type SalesItem {
  id: ID!
  createdAtTimestampInMillis: String!
  name: String!
  priceInCents(currency: Currency = USD): Int!
  images(
    sortByField: String = "rank",
    sortDirection: SortDirection = ASC,
    offset: Int = 0,
    limit: Int = 5
  ): [Image!]!
}
```

Now, clients can supply a currency parameter for the `price` property in their queries to get the price in different currencies. The default currency is *USD* if no currency parameter is supplied.

Below are two example queries that a client could perform against the earlier defined GraphQL schema:

Figure 8.88

```
{
  # gets the name, price in euros and the first 5 images
  # for the sales item with id "1"
  salesItem(id: "1") {
    name
    price(currency: EUR)
    images
  }

  # gets the next 5 images for the sales item 1
  salesItem(id: "1") {
    images(offset: 5)
  }
}
```

In real life, consider limiting the fetching of resources only to the previous or the next page (or the next page only if you are implementing infinite scrolling on the client side). Then, clients cannot fetch random pages. This prevents attacks where a malicious user tries to fetch a page with a huge page number (like 10,000, for example), which can cause extra load for the server or, at the extreme, a denial of service.

Below is an example where clients can only query the first, next, or previous page. When a client requests the first page, the page cursor can be empty, but when the client requests the previous or the next page, it must give the current page cursor as a query parameter.

Figure 8.89

```
type PageOfSalesItems {
  # Contains the page number encrypted and
  # encoded as a Base64 value.
  pageCursor: String!

  salesItems: [SalesItem!]!
}

enum Page {
  FIRST,
  NEXT,
  PREVIOUS
}

type Query {
  pageOfSalesItems(
    page: Page = FIRST,
    pageCursor: String = ""
  ): PageOfSalesItems!
}
```

Let's have another example with GraphQL and use the code-first approach, this time with the Strawberry library. When implementing production code, we should follow the *clean microservice design principle*. We should be able to share the services, repositories, DTOs, errors, and entities with the earlier *sales-item-service* REST API example and only define a separate controller for the GraphQL API as we did with the Ariadne library. The example below implements only two API endpoints (getting and creating a sales item) to keep the example shorter.

Figure 8.90

```
import strawberry
from dependency_injector.wiring import Provide
from strawberry.fastapi import GraphQLRouter
from strawberry.types import Info

from ..graphqltypes.IdReponse import IdResponse
from ..graphqltypes.InputSalesItem import InputSalesItem
from ..graphqltypes.OutputSalesItem import OutputSalesItem
from ..service.SalesItemService import SalesItemService

sales_item_service: SalesItemService =
Provide['sales_item_service']
```

```
class StrawberryGraphQlSalesItemController:
    @strawberry.type
    class Query:
        @strawberry.field
        def salesItems(self, info: Info) ->
list[OutputSalesItem]:
            output_sales_items =
sales_item_service.get_sales_items()

            return [
                OutputSalesItem.from_pydantic(output_sales_item)
                for output_sales_item in output_sales_items
            ]

        @strawberry.field
        def salesItem(self, info: Info, id: str) ->
OutputSalesItem:
            output_sales_item =
sales_item_service.get_sales_item(id)
            return
OutputSalesItem.from_pydantic(output_sales_item)

    @strawberry.type
    class Mutation:
        @strawberry.mutation
        def createSalesItem(
            self, info: Info, inputSalesItem: InputSalesItem
        ) -> OutputSalesItem:
            output_sales_item =
sales_item_service.create_sales_item(
                inputSalesItem.to_pydantic()
            )

            return
OutputSalesItem.from_pydantic(output_sales_item)

        @strawberry.mutation
        def updateSalesItem(
            self, info: Info, id: str, inputSalesItem:
InputSalesItem
        ) -> IdResponse:
            sales_item_service.update_sales_item(
                id, inputSalesItem.to_pydantic()
            )
            return IdResponse(id=id)

        @strawberry.mutation
        def deleteSalesItem(self, info: Info, id: str) ->
IdResponse:
            sales_item_service.delete_sales_item(id)
            return IdResponse(id=id)

    __schema = strawberry.Schema(query=Query, mutation=Mutation)
    __router = GraphQLRouter(__schema, path='/graphql')

    @property
    def router(self):
        return self.__router
```

We must add authorization, audit logging, and metrics updates to make our controller more production-like. We can implement decorators similar to those in the earlier REST API example. When the decorators need to access the request, it can be done via the *info* parameter:

```
info.context['request']
```

In addition to the above controller, we must define strawberry types, which can be based on existing *pydantic* classes. Here are the strawberry types:

Figure 8.91

```
import strawberry

from ..dtos.InputSalesItem import InputSalesItem
from .InputSalesItemImage import InputSalesItemImage


@strawberry.experimental.pydantic.input(model=InputSalesItem)
class InputSalesItem:
    name: strawberry.auto
    priceInCents: strawberry.auto
    images: list[InputSalesItemImage]
```

Figure 8.92

```
import strawberry

from ..dtos.SalesItemImage import SalesItemImage


@strawberry.experimental.pydantic.input(
    model=SalesItemImage, all_fields=True
)
class InputSalesItemImage:
    pass
```

Figure 8.93

```
import strawberry

from ..dtos.OutputSalesItem import OutputSalesItem
from .OutputSalesItemImage import OutputSalesItemImage


@strawberry.experimental.pydantic.type(model=OutputSalesItem)
class OutputSalesItem:
    id: strawberry.auto
    createdAtTimestampInMs: str
    name: strawberry.auto
    priceInCents: strawberry.auto
    images: list[OutputSalesItemImage]
```

Figure 8.94

```
import strawberry

from ..dtos.SalesItemImage import SalesItemImage


@strawberry.experimental.pydantic.type(
    model=SalesItemImage, all_fields=True
)
class OutputSalesItemImage:
    pass
```

8.1.4: Subscription-Based API Design

Design a subscription-based API when you want clients to be able to subscribe to small, incremental changes to large objects or when clients want to receive low-latency real-time updates.

8.1.4.1: Server-Sent Events (SSE)

Server-Sent Events (SSE) is a uni-directional push technology enabling a client to receive updates from a server via an HTTP connection.

Let's showcase the SSE capabilities with a real-life example. The below example defines a *subscribe-to-loan-app-summaries* API endpoint for clients to subscribe to loan application summaries. A client will show loan application summaries in a list view in its UI. Whenever a new summary for a loan application is available, the server will send a loan application summary event to clients that will update their UIs by adding a new loan application summary. The below example uses FastAPI and the sse-starlette library.

Figure 8.95

```
import json

from fastapi import FastAPI, Request
from sse_starlette.sse import EventSourceResponse

loan_app_summaries = []

app = FastAPI()

def get_loan_app_summary():
    if len(loan_app_summaries) > 0:
        return loan_app_summaries.pop(0)
    return None
```

```
@app.get('/subscribe-to-loan-app-summaries')
async def subscribe_to_loan_app_summaries(request: Request):
    async def generate_loan_app_summary_events():
        while True:
            if await request.is_disconnected():
                break

            loan_app_summary = get_loan_app_summary()
            if loan_app_summary:
                yield json.dumps(loan_app_summary)

    return EventSourceResponse(
        generate_loan_app_summary_events()
    )

@app.post('/loan-app-summaries')
async def create_loan_app_summary(
    request: Request
) -> None:
    loan_app_summary = await request.json()
    loan_app_summaries.append(loan_app_summary)
```

Next, we can implement the web client in JavaScript and define the following React functional component:

Figure 8.96

```
import React, { useEffect, useState } from 'react';

export default function LoanAppSummaries() {
  const [ loanAppSummaries, setLoanAppSummaries ] =
useState([]);

  // Define an effect to be executed on component mount
  useEffect(() => {
    // Create new event source
    // Hardcoded dev environment URL is used here for
demonstration
    // purposes
    const eventSource =
      new EventSource('http://localhost:8000/subscribe-to-loan-
app-summaries');

    // Listen to server sent events and add a new
    // loan application summary to the head of
    // loanAppSummaries array
    eventSource.addEventListener('message', (messageEvent) => {
      try {
        const loanAppSummary = JSON.parse(messageEvent.data);

        if (loanAppSummary) {
          setLoanAppSummaries([loanAppSummary,
...loanAppSummaries]);
        }
      } catch {
        // Handle error
      }
    });
```

```
    eventSource.addEventListener('error', (errorEvent) => {
      // Handle error
    });

    // Close the event source on component unmount
    return function cleanup() { eventSource.close(); }
  }, [loanAppSummaries]);

  // Render loan application summary list items
  const loanAppSummaryListItems =
    loanAppSummaries.map(({ ... }) =>
       (<li key={key here...}>render here...</li>));

  return (
    <ul>{loanAppSummaryListItems}</ul>
  );
}
```

8.1.4.2: GraphQL Subscriptions

Let's have an example of a GraphQL subscription. The below GraphQL schema defines one subscription for a post's comments. It is not relevant what a post is. It can be a blog post or social media post, for example. We want a client to be able to subscribe to a post's comments.

Figure 8.97

```
type PostComment {
  id: ID!,
  text: String!
}

type Subscription {
  postComment(postId: ID!): PostComment
}
```

On the client side, we can have the below JavaScript code to define a subscription named `postCommentText` that subscribes to a post's comments and returns the text property of comments:

Figure 8.98

```
import { gql } from '@apollo/client';

const POST_COMMENT_SUBSCRIPTION = gql`
  subscription postCommentText($postId: ID!) {
    postComment(postID: $postId) {
      text
    }
  }
`;
```

If a client executes the above query for a particular post (defined with the `postId` parameter), the following kind of response can be expected:

Figure 8.99

```
{
  "data": {
    "postComment": {
      "text": "Nice post!"
    }
  }
}
```

To be able to use GraphQL subscriptions, you must implement support for them both on the server and client side. In practice, this means setting up WebSocket communication because GraphQL uses it to implement subscriptions. For the server side, you can find instructions for the *Ariadne* library here: https://ariadnegraphql.org/docs/subscriptions. And for the client side, you can find instructions for the *Apollo client* here: https://www.apollographql.com/docs/react/data/subscriptions/etting-up-the-transport

After the server and client-side support for subscriptions are implemented, you can use the subscription in your React component:

Figure 8.100

```
import { useState } from 'react';
import { gql, useSubscription } from '@apollo/client';

const POST_COMMENT_SUBSCRIPTION = gql`
  subscription subscribeToPostComment($postId: ID!) {
    postComment(postID: $postId) {
      id
      text
    }
  }
`;

export default function SubscribedPostCommentsView({ postId }) {
  const [ postComments, setPostComments ] = useState([]);

  const { data } = useSubscription(POST_COMMENT_SUBSCRIPTION,
                                   { variables: { postId } });

  if (data?.postComment) {
    setPostComments([...postComments, data.postComment]);
  }

  const postCommentListItems =
    postComments.map(( { id, text }) =>
      (<li key={id}>{text}</li>));
```

```
  return <ul>{postCommentListItems}</ul>;
}
```

8.1.5: WebSocket Example

Below is a chat messaging application consisting of a WebSocket server implemented with FastAPI and a WebSocket client implemented with React. The server uses Kafka and Redis. There can be multiple instances of the server running. These instances are stateless except for storing WebSocket connections for locally connected clients. First, we list the source code files for the server side.

A new Redis client is created using the redis-py library:

Figure 8.101

```
import os

from redis import Redis

# The current version of official Python documentation
# does not tell what errors the 'int' constructor can raise,
# but it can raise a 'TypeError' if the argument type
# is not convertible to an integer, and it can raise
# 'ValueError' if the argument value is not convertible
# to an integer
try:
    port = int(os.environ.get('REDIS_PORT'))
except (TypeError, ValueError):
    port = 6379

redis_client = Redis(
    host=os.environ.get('REDIS_HOST') or 'localhost',
    port=port,
    username=os.environ.get('REDIS_USERNAME'),
    password=os.environ.get('REDIS_PASSWORD'),
)
```

The below `KafkaMsgBrokerAdminClient` class is used to create topics in Kafka:

Figure 8.102

```
from typing import Protocol

from WebSocketExampleError import WebSocketExampleError


class ChatMsgBrokerAdminClient(Protocol):
    class CreateTopicError(WebSocketExampleError):
        pass
```

```
    def try_create_topic(self, name: str) -> None:
        pass
```

Figure 8.103

```
import os

from ChatMsgBrokerAdminClient import ChatMsgBrokerAdminClient
from confluent_kafka import KafkaError, KafkaException
from confluent_kafka.admin import AdminClient
from confluent_kafka.cimpl import NewTopic


class KafkaChatMsgBrokerAdminClient(ChatMsgBrokerAdminClient):
    def __init__(self):
        self.__admin_client = AdminClient(
            {
                'bootstrap.servers':
os.environ.get('KAFKA_BROKERS'),
                'client.id': 'chat-messaging-service',
            }
        )

    def try_create_topic(self, name: str) -> None:
        topic = NewTopic(name)

        try:
            topic_name_to_creation_dict = (
                self.__admin_client.create_topics([topic])
            )
            topic_name_to_creation_dict[name].result()
        except KafkaException as error:
            if error.args[0].code() !=
KafkaError.TOPIC_ALREADY_EXISTS:
                raise self.CreateTopicError(error)
```

Users of the chat messaging application are identified with phone numbers. On the server side, we store the WebSocket connection for each user in the `phone_nbr_to_conn_map`:

Figure 8.104

```
from Connection import Connection

phone_nbr_to_conn_map: dict[str, Connection] = {}
```

Figure 8.105

```
from typing import Any, Protocol

from WebSocketExampleError import WebSocketExampleError


class Connection(Protocol):
    class Error(WebSocketExampleError):
        pass

    async def try_connect(self) -> None:
        pass
```

```
    async def try_send_json(self, message: dict[str, Any]) ->
None:
        pass

    async def try_send_text(self, message: str) -> None:
        pass

    async def try_receive_json(self) -> dict[str, str]:
        pass

    async def try_close(self) -> None:
        pass
```

Figure 8.106

```
from typing import Any

from Connection import Connection
from fastapi import WebSocket, WebSocketException


class WebSocketConnection(Connection):
    def __init__(self, websocket: WebSocket):
        self.__websocket = websocket

    async def try_connect(self) -> None:
        try:
            await self.__websocket.accept()
        except WebSocketException:
            raise self.Error()

    async def try_send_json(self, message: dict[str, Any]) ->
None:
        try:
            await self.__websocket.send_json(message)
        except WebSocketException:
            raise self.Error()

    async def try_send_text(self, message: str) -> None:
        try:
            await self.__websocket.send_text(message)
        except WebSocketException:
            raise self.Error()

    async def try_receive_json(self) -> dict[str, str]:
        try:
            return await self.__websocket.receive_json()
        except WebSocketException:
            raise self.Error()

    async def try_close(self) -> None:
        try:
            return await self.__websocket.close()
        except WebSocketException:
            raise self.Error()
```

The below module is the WebSocket server. The server accepts connections from clients. When it receives a chat message from a client, it will first parse and validate it. Then, it will

store the message in persistent storage (using a separate *chat-message-service* REST API, not implemented here). The server gets the recipient's server information from a Redis cache. It sends the chat message to the recipient's WebSocket connection or produces the chat message to a Kafka topic where another microservice instance can consume the chat message and send it to the recipient's WebSocket connection. The Redis cache stores a hash map where the users' phone numbers are mapped to the server instance they are currently connected. A UUID identifies a microservice instance.

Figure 8.107

```
from typing import Protocol

from .Connection import Connection


class ChatMsgServer(Protocol):
    async def handle(
        self, connection: Connection, phone_number: str
    ) -> None:
        pass
```

Figure 8.108

```
import json
from typing import Final

from ChatMsgBrokerProducer import ChatMsgBrokerProducer
from ChatMsgServer import ChatMsgServer
from Connection import Connection
from fastapi import WebSocket, WebSocketDisconnect,
WebSocketException
from KafkaChatMsgBrokerProducer import
KafkaChatMsgBrokerProducer
from phone_nbr_to_conn_map import phone_nbr_to_conn_map
from PhoneNbrToInstanceUuidCache import
PhoneNbrToInstanceUuidCache
from redis_client import redis_client
from RedisPhoneNbrToInstanceUuidCache import (
    RedisPhoneNbrToInstanceUuidCache,
)
from WebSocketConnection import WebSocketConnection


class WebSocketChatMsgServer(ChatMsgServer):
    def __init__(self, instance_uuid: str):
        self.__instance_uuid: Final = instance_uuid
        self.__conn_to_phone_nbr_map: Final[dict[Connection,
str]] = {}
        self.__chat_msg_broker_producer: Final = (
            KafkaChatMsgBrokerProducer()
        )
        self.__cache: Final = RedisPhoneNbrToInstanceUuidCache(
            redis_client
        )
```

```
    async def handle(
        self, connection: Connection, phone_number: str
    ) -> None:
        try:
            await connection.try_connect()
            phone_nbr_to_conn_map[phone_number] = connection
            self.__conn_to_phone_nbr_map[connection] =
phone_number
            self.__cache.try_store(phone_number,
self.__instance_uuid)

            while True:
                chat_message: dict[
                    str, str
                ] = await connection.try_receive_json()

                # Validate chat_message ...
                # Store chat message permanently using another
API ...
                recipient_phone_nbr =
chat_message.get('recipientPhoneNbr')

                recipient_instance_uuid = (
                    self.__cache.retrieve_instance_uuid(
                        recipient_phone_nbr
                    )
                )

                await self.__try_send(
                    chat_message, recipient_instance_uuid
                )
        except WebSocketDisconnect:
            self.__disconnect(connection)
        except PhoneNbrToInstanceUuidCache.Error:
            # Handle error ...
        except Connection.Error:
            # Handle error ...
        except ChatMsgBrokerProducer.Error:
            # Handle error ...

    def close(self) -> None:
        for connection in self.__conn_to_phone_nbr_map.keys():
            try:
                connection.try_close()
            except Connection.Error:
                pass

        self.__chat_msg_broker_producer.close()

    async def __try_send(
        self,
        chat_message: dict[str, str],
        recipient_instance_uuid: str | None,
    ) -> None:
        if recipient_instance_uuid == self.__instance_uuid:
            # Recipient has active connection on
            # the same server instance as sender
            recipient_conn = phone_nbr_to_conn_map.get(
                chat_message.get('recipientPhoneNbr')
            )
```

```
            if recipient_conn:
                await recipient_conn.try_send_json(chat_message)
        elif recipient_instance_uuid:
            # Recipient has active connection on different
            # server instance compared to sender
            chat_message_json = json.dumps(chat_message)

            self.__chat_msg_broker_producer.try_produce(
                chat_message_json, topic=recipient_instance_uuid
            )

    def __disconnect(self, connection: Connection) -> None:
        phone_number =
self.__conn_to_phone_nbr_map.get(connection)

        if phone_number:
            del phone_nbr_to_conn_map[phone_number]

        del self.__conn_to_phone_nbr_map[connection]

        try:
            self.__cache.try_remove(phone_number)
        except PhoneNbrToInstanceUuidCache.Error:
            # Handle error ...
```

Figure 8.109

```
from typing import Protocol

from WebSocketExampleError import WebSocketExampleError


class PhoneNbrToInstanceUuidCache(Protocol):
    class Error(WebSocketExampleError):
        pass

    def retrieve_instance_uuid(
        self, phone_number: str | None
    ) -> str | None:
        pass

    def try_store(self, phone_number: str, instance_uuid: str) -
> None:
        pass

    def try_remove(self, phone_number: str) -> None:
        pass
```

Figure 8.110

```
from PhoneNbrToInstanceUuidCache import
PhoneNbrToInstanceUuidCache
from redis import Redis, RedisError


class
RedisPhoneNbrToInstanceUuidCache(PhoneNbrToInstanceUuidCache):
    def __init__(self, redis_client: Redis):
        self.__redis_client = redis_client

    def retrieve_instance_uuid(
```

```
        self, phone_number: str | None
    ) -> str | None:
        if phone_number:
            try:
                return self.__redis_client.hget(
                    'phoneNbrToInstanceUuidMap', phone_number
                )
            except RedisError:
                pass

        return None

    def try_store(self, phone_number: str, instance_uuid: str) -
> None:
        try:
            self.__redis_client.hset(
                'phoneNbrToInstanceUuidMap', phone_number,
instance_uuid
            )
        except RedisError:
            raise self.Error()

    def try_remove(self, phone_number: str) -> None:
        try:
            self.__redis_client.hdel(
                'phoneNbrToInstanceUuidMap', [phone_number]
            )
        except RedisError:
            raise self.Error()
```

Figure 8.111

```
from typing import Protocol

from WebSocketExampleError import WebSocketExampleError


class ChatMsgBrokerProducer(Protocol):
    class Error(WebSocketExampleError):
        pass

    def try_produce(self, chat_message_json: str, topic: str):
        pass

    def close(self):
        pass
```

Figure 8.112

```
import os

from ChatMsgBrokerProducer import ChatMsgBrokerProducer
from confluent_kafka import KafkaException, Producer


class KafkaChatMsgBrokerProducer(ChatMsgBrokerProducer):
    def __init__(self):
        config = {
            'bootstrap.servers':
os.environ.get('KAFKA_BROKERS'),
            'client.id': 'chat-messaging-service',
```

```
        }

        self.__producer = Producer(config)

    def try_produce(self, chat_message_json: str, topic: str):
        def handle_error(error: KafkaException):
            if error is not None:
                raise self.Error()

        try:
            self.__producer.produce(
                topic, chat_message_json,
on_delivery=handle_error
            )

            self.__producer.poll()
        except KafkaException:
            raise self.Error()

    def close(self):
        try:
            self.__producer.flush()
        except KafkaException:
            pass
```

The `KafkaChatMsgBrokerConsumer` class defines a Kafka consumer that consumes chat messages from a particular Kafka topic and sends them to the recipient's WebSocket connection:

Figure 8.113

```
from typing import Protocol


class ChatMsgBrokerConsumer(Protocol):
    def consume_chat_msgs(self) -> None:
        pass

    def stop(self) -> None:
        pass

    def close(self) -> None:
        pass
```

Figure 8.114

```
import json
import os

from ChatMsgBrokerConsumer import ChatMsgBrokerConsumer
from confluent_kafka import Consumer, KafkaException
from Connection import Connection
from phone_nbr_to_conn_map import phone_nbr_to_conn_map


class KafkaChatMsgBrokerConsumer(ChatMsgBrokerConsumer):
    def __init__(self, topic: str):
        self.__topic = topic
```

```
        config = {
            'bootstrap.servers':
os.environ.get('KAFKA_BROKERS'),
            'group.id': 'chat-messaging-service',
            'auto.offset.reset': 'smallest',
            'enable.partition.eof': False,
        }

        self.__consumer = Consumer(config)
        self.__is_running = True

    def consume_chat_msgs(self) -> None:
        self.__consumer.subscribe([self.__topic])

        while self.__is_running:
            try:
                messages = self.__consumer.poll(timeout=1)

                if messages is None:
                    continue

                for message in messages:
                    if message.error():
                        raise KafkaException(message.error())
                    else:
                        chat_message_json = message.value()
                        chat_message =
json.loads(chat_message_json)

                        recipient_conn =
phone_nbr_to_conn_map.get(

chat_message.get('recipientPhoneNbr')
                        )

                        if recipient_conn:

recipient_conn.try_send_text(chat_message_json)
            except KafkaException:
                # Handle error ...
            except Connection.Error:
                # Handle error ...

    def stop(self) -> None:
        self.__is_running = False

    def close(self):
        self.__consumer.close()
```

Figure 8.115

```
import sys
from threading import Thread
from uuid import uuid4

from fastapi import FastAPI, WebSocket
from KafkaChatMsgBrokerAdminClient import
KafkaChatMsgBrokerAdminClient
from KafkaChatMsgBrokerConsumer import
KafkaChatMsgBrokerConsumer
```

```
from WebSocketChatMsgServer import WebSocketChatMsgServer
from WebSocketConnection import WebSocketConnection

instance_uuid = str(uuid4())

# Create a Kafka topic for this particular microservice instance
try:

KafkaChatMsgBrokerAdminClient().try_create_topic(instance_uuid)
except KafkaChatMsgBrokerAdminClient.CreateTopicError:
    # Log error
    sys.exit(1)

# Create and start a Kafka consumer to consume and send
# chat messages for recipients that are connected to
# this microservice instance
chat_msg_broker_consumer =
KafkaChatMsgBrokerConsumer(topic=instance_uuid)

chat_msg_consumer_thread = Thread(
    target=chat_msg_broker_consumer.consume_chat_msgs
)

chat_msg_consumer_thread.start()

app = FastAPI()
chat_msg_server = WebSocketChatMsgServer(instance_uuid)


@app.websocket('/chat-messaging-service/{phone_number}')
async def handle_websocket(websocket: WebSocket, phone_number:
str):
    connection = WebSocketConnection(websocket)
    await chat_msg_server.handle(connection, phone_number)


@app.on_event('shutdown')
def shutdown_event():
    chat_msg_broker_consumer.stop()
    chat_msg_consumer_thread.join()
    chat_msg_broker_consumer.close()
    chat_msg_server.close()
```

For the web client, we have the below code. An instance of the `ChatMessagingService` class connects to a chat messaging server via WebSocket. It listens to messages received from the server and dispatches an action upon receiving a chat message. The class also offers a method for sending a chat message to the server.

Figure 8.116

```
import store from "./store";

class ChatMessagingService {
  wsConnection;
  connectionIsOpen = false;
  lastChatMessage;
```

```
  constructor(dispatch, userPhoneNbr) {
    this.wsConnection =
        new WebSocket(`ws://localhost:8080/chat-messaging-
service/${userPhoneNbr}`);

    this.wsConnection.addEventListener('open', () => {
      this.connectionIsOpen = true;
    });

    this.wsConnection.addEventListener('error', () => {
      this.lastChatMessage = null;
    });

    this.wsConnection.addEventListener(
      'message',
      ({ data: chatMessageJson }) => {
        const chatMessage = JSON.parse(chatMessageJson);

        store.dispatch({
          type: 'receivedChatMessageAction',
          chatMessage
        });
      });

    this.wsConnection.addEventListener('close', () => {
      this.connectionIsOpen = false;
    });
  }

  send(chatMessage) {
    this.lastChatMessage = chatMessage;

    if (this.connectionIsOpen) {
        this.wsConnection.send(JSON.stringify(chatMessage));
    } else  {
      // Send message to REST API
    }
  }

  close() {
    this.connectionIsOpen = false;
    this.wsConnection.close();
  }
}

export let chatMessagingService;

export default function createChatMessagingService(
  userPhoneNbr
) {
  chatMessagingService =
    new ChatMessagingService(store.dispatch, userPhoneNbr);

  return chatMessagingService;
}
```

Figure 8.117

```
import React from 'react';
import ReactDOM from 'react-dom/client';
```

```
import { Provider } from 'react-redux'
import ChatApp from './ChatApp';
import store from './store'
import './index.css';

const root =
ReactDOM.createRoot(document.getElementById('root'));

root.render(
  <Provider store={store}>
    <ChatApp/>
  </Provider>
);
```

The chat application `ChatApp` parses the user's and contact's phone numbers from the URL and then renders a chat view between the user and the contact:

Figure 8.118

```
import React, { useEffect } from 'react';
import queryString from "query-string";
import ContactChatView from "./ContactChatView";
import createChatMessagingService from "./ChatMessagingService";

const { userPhoneNbr, contactPhoneNbr } =
  queryString.parse(window.location.search);

export default function ChatApp() {
  useEffect(() => {
    const chatMessagingService =
      createChatMessagingService(userPhoneNbr);

    return function cleanup() {
      chatMessagingService.close();
    }
  }, []);

  return (
    <div>
      User: {userPhoneNbr}
      <ContactChatView
        userPhoneNbr={userPhoneNbr}
        contactPhoneNbr={contactPhoneNbr}
      />
    </div>
  );
}
```

The `ContactChatView` component renders chat messages between a user and a contact:

Figure 8.119

```
import React, { useEffect, useRef } from 'react';
import { connect } from "react-redux";
import store from './store';

function ContactChatView({
```

```
  userPhoneNbr,
  contactPhoneNbr,
  chatMessages,
  fetchLatestChatMessages
}) {
  const inputElement = useRef(null);

  useEffect(() => {
    fetchLatestChatMessages(userPhoneNbr, contactPhoneNbr);
  }, [contactPhoneNbr,
      fetchLatestChatMessages,
      userPhoneNbr]
  );

  function sendChatMessage() {
    if (inputElement?.current.value) {
      store.dispatch({
        type: 'sendChatMessageAction',
        chatMessage: {
          senderPhoneNbr: userPhoneNbr,
          recipientPhoneNbr: contactPhoneNbr,
          message: inputElement.current.value
        }
      });
    }
  }

  const chatMessageElements = chatMessages
    .map(({ message, senderPhoneNbr }, index) => {
      const messageIsReceived =
        senderPhoneNbr === contactPhoneNbr;

      return (
        <li
          key={index}
          className={messageIsReceived ? 'received' : 'sent'}>
            {message}
        </li>
      );
    });

  return (
    <div className="contactChatView">
      Contact: {contactPhoneNbr}
      <ul>{chatMessageElements}</ul>
      <input ref={inputElement}/>
      <button onClick={sendChatMessage}>Send</button>
    </div>
  );
}

function mapStateToProps(state) {
  return {
    chatMessages: state
  };
}

export default connect(mapStateToProps)(ContactChatView);
```

Figure 8.120

```
import { createStore } from 'redux';
import { chatMessagingService } from "./ChatMessagingService";

function chatMessagesReducer(state = [], { type, chatMessage })
{
  switch (type) {
    case 'receivedChatMessageAction':
      return state.concat([chatMessage]);
    case 'sendChatMessageAction':
      chatMessagingService.send(chatMessage);
      return state.concat([chatMessage]);
    default:
      return state;
  }
}

const store = createStore(chatMessagesReducer)
export default store;
```

Figure 8.121

```
.contactChatView {
  width: 420px;
}

.contactChatView ul {
  padding-inline-start: 0;
  list-style-type: none;
}

.contactChatView li {
  margin-top: 15px;
  width: fit-content;
  max-width: 180px;
  padding: 10px;
  border: 1px solid #888;
  border-radius: 20px;
}

.contactChatView li.received {
  margin-right: auto;
}

.contactChatView li.sent {
  margin-left: auto;
}
```

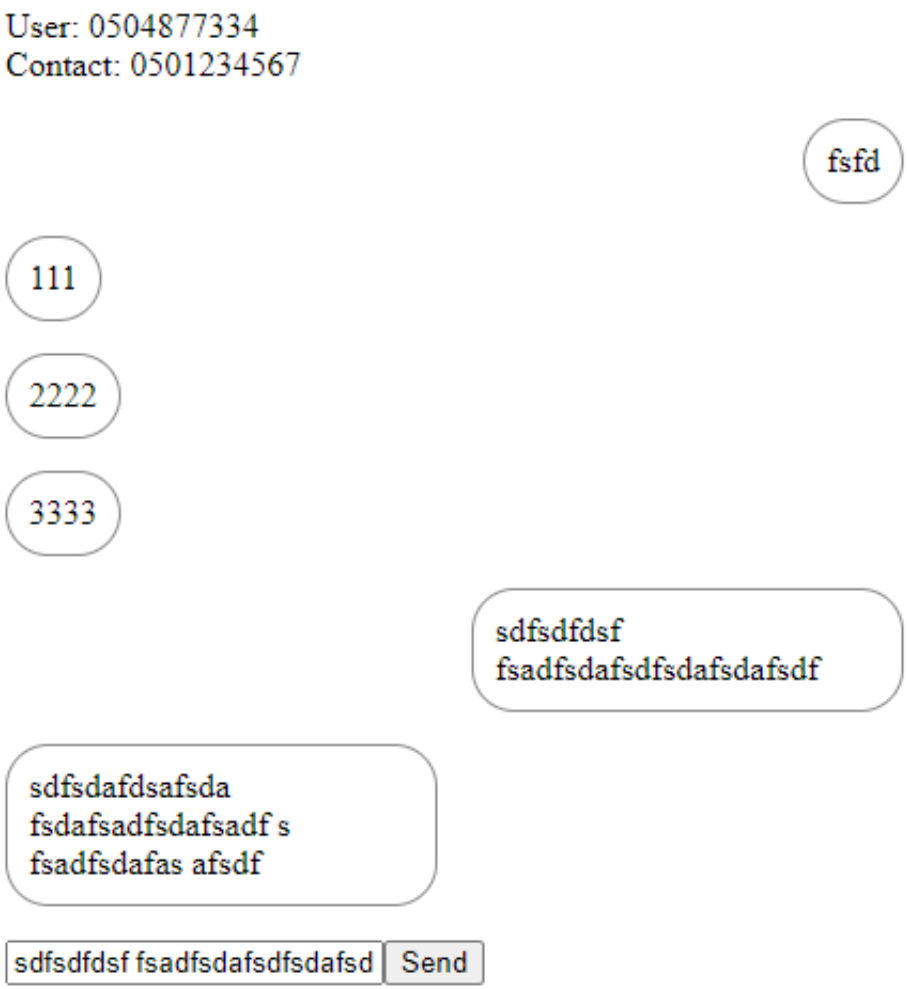

Figure 8.122. Chat Messaging Application Views for Two Users

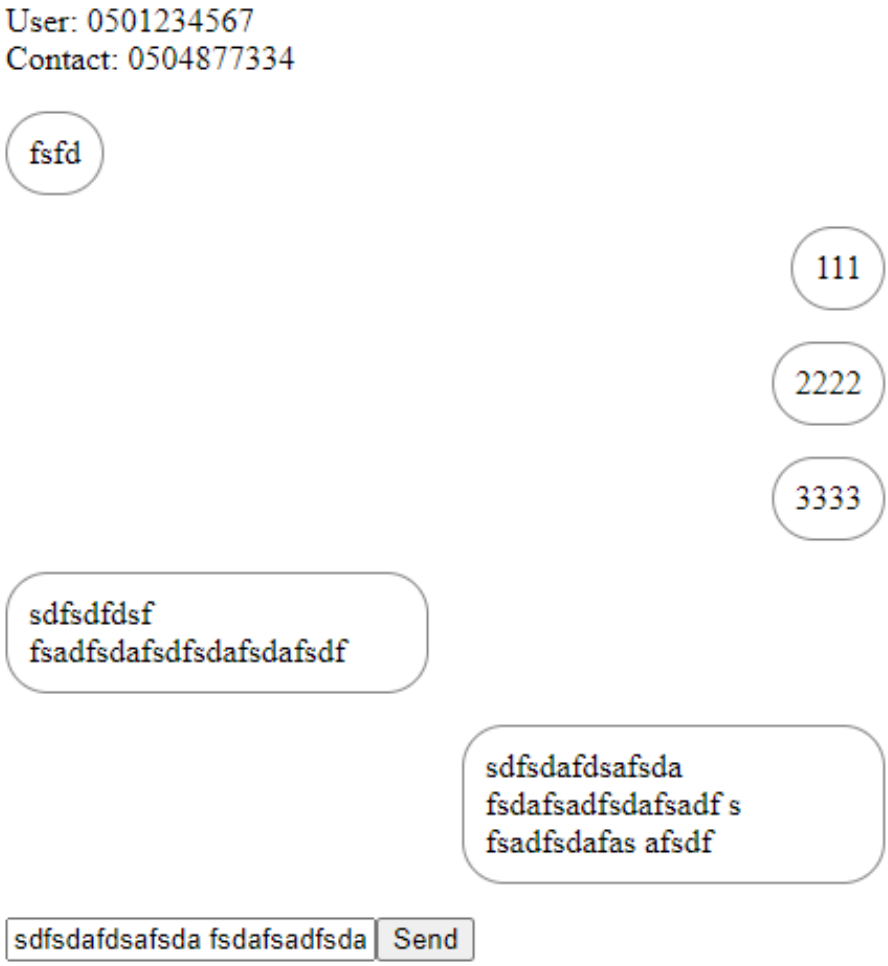

Figure 8.123. Chat Messaging Application Views for Two Users

8.2: Inter-Microservice API Design Principles

Inter-microservice APIs can be divided into two categories based on the type of communication: synchronous and asynchronous. Synchronous communication should be used when an immediate response to an issued request is expected. Asynchronous communication can be used when no response to a request is expected, or the response is not immediately required.

8.2.1: Synchronous API Design Principle

Use HTTP-based RPC or REST APIs with JSON data encoding, preferably with HTTP/2 or HTTP/3 transport, when requests and responses are not very large and do not contain much binary data. Suppose you have large requests or responses or a lot of binary data. In that case, you are better off encoding the data in Avro binary format (Content-Type: avro/binary) instead of JSON or using a gRPC-based API. gRPC always encodes data in a binary format (Protocol Buffers).

8.2.1.1: gRPC-Based API Design Example

Let's have an example of a gRPC-based API. First, we must define the needed Protocol Buffers types. They are defined in a file named with the extension *.proto*. The syntax of *proto* files is pretty simple. We define the service by listing the remote procedures. A remote procedure is defined with the following syntax: `rpc <procedure-name> (<argument-type>) returns (<return-type>) {}`. A remote procedure always accepts exactly one argument and returns one value. A message type is defined with the below syntax:

Figure 8.124

```
message <type-name> {
  [optional] [repeated] <field-type> <field-name> [= <field-
index>];
  ...
}
```

Below are the Protocol Buffers definitions for the *sales-item-service*:

Figure 8.125. sales_item_service.proto

```
syntax = "proto3";

option objc_class_prefix = "SIS";

package salesitemservice;

service SalesItemService {
  rpc createSalesItem (InputSalesItem) returns (OutputSalesItem)
{}
  rpc getSalesItems (GetSalesItemsOptions) returns
(OutputSalesItems) {}
  rpc getSalesItem (Id) returns (OutputSalesItem) {}
  rpc updateSalesItem (SalesItemUpdate) returns (Nothing) {}
  rpc deleteSalesItem (Id) returns (Nothing) {}
```

```
}

message GetSalesItemsOptions {
  optional string sortByField = 1;
  optional string sortDirection = 2;
  optional uint64 offset = 3;
  optional uint64 limit = 4;
}

message Nothing {}

message Image {
  uint64 id = 1;
  uint64 rank = 2;
  string url = 3;
}

message InputSalesItem {
  string name = 1;
  float price = 2;
  repeated Image images = 3;
}

message SalesItemUpdate {
  uint64 id = 1;
  string name = 2;
  float price = 3;
  repeated Image images = 4;
}

message OutputSalesItem {
  uint64 id = 1;
  uint64 createdAtTimestampInMillis = 2;
  string name = 3;
  float price = 4;
  repeated Image images = 5;
}

message Id {
  uint64 id = 1;
}

message OutputSalesItems {
  repeated OutputSalesItem salesItems = 1;
}

message ErrorDetails {
    optional string code = 1;
    optional string description = 2;
}
```

In the above example, the `getSalesItems` procedure returns an object that contains an array of sales items. gRPC offers the possibility to stream data in both directions. For example, we could make the `getSalesItems` procedure a streaming method, and then we did not need the properties `offset` and `limit` in

the `GetSalesItemsArg`. We could define a streaming `getSalesItems` procedure as follows:

Figure 8.126

```
// ...

service SalesItemService {
  // ...
  rpc getSalesItems (GetSalesItemsArg) returns (stream
OutputSalesItem) {}
  // ...
}

// ...
```

After completing the *proto* file, we must generate code for the gRPC server. Let's install the *grpcio-tools* library:

Figure 8.127

```
pip install grpcio-tools
```

Then, we can generate code with the following command:

Figure 8.128

```
python -m grpc_tools.protoc -I. --python_out=. --pyi_out=. --
grpc_python_out=. sales_item\
_service.proto
```

After executing the above command, three files should be generated in the directory. Creating the actual server code requires the following two steps:

- Implementing the generated *servicer* interface with functions that perform the actual "work" of the service.
- Run a gRPC server that listens for client requests and transmits responses.

We need to install the following libraries:

Figure 8.129

```
pip install grpcio grpcio-status
```

Let's implement the gRPC controller for the *sales-item-service*:

Figure 8.130

```python
from dependency_injector.wiring import Provide
from google.protobuf import any_pb2, json_format
from google.rpc import code_pb2, status_pb2
from grpc_status import rpc_status
from pydantic import ValidationError

from ..dtos.InputSalesItem import InputSalesItem as
PydanticInputSalesItem
from ..errors.SalesItemServiceError import SalesItemServiceError
from ..grpc.proto_to_dict import proto_to_dict
from ..grpc.sales_item_service_pb2 import (
    ErrorDetails,
    GetSalesItemsArg,
    Id,
    InputSalesItem,
    Nothing,
    OutputSalesItem,
    OutputSalesItems,
    SalesItemUpdate,
)
from ..grpc.sales_item_service_pb2_grpc import
SalesItemServiceServicer
from ..service.SalesItemService import SalesItemService
from ..utils import get_stack_trace


def map_http_status_code_to_grpc_status_code(error: Exception):
    # Map HTTP status code here to
    # respective gRPC status code ...
    # Mapping info is available here:
    # https://cloud.google.com/apis/design/errors#error_model
    return code_pb2.INTERNAL


def create_status_from(error: Exception) -> status_pb2.Status:
    detail = any_pb2.Any()

    if isinstance(error, SalesItemServiceError):
        grpc_status_code =
map_http_status_code_to_grpc_status_code(error)
        message = error.message

        detail.Pack(
            ErrorDetails(
                code=error.code,
                description=error.description,
                # get_stack_trace returns stack trace only
                # when environment is not production
                # otherwise it returns None
                stackTrace=get_stack_trace(error.cause),
            )
        )
    elif isinstance(error, ValidationError):
        grpc_status_code = code_pb2.INVALID_ARGUMENT
        message = 'Request validation failed'
        detail.Pack(
            ErrorDetails(
                code='RequestValidationError',
description=str(error)
            )
        )
```

```
    else:
        grpc_status_code = code_pb2.INTERNAL
        message = 'Unspecified internal error'
        detail.Pack(
            ErrorDetails(
                code='UnspecifiedError',
                description=str(error),
                stackTrace=get_stack_trace(error),
            )
        )

    return status_pb2.Status(
        code=grpc_status_code,
        message=message,
        details=[detail],
    )


class GrpcSalesItemController(SalesItemServiceServicer):
    __sales_item_service: SalesItemService =
Provide['sales_item_service']

    def createSalesItem(
        self, input_sales_item: InputSalesItem, context
    ) -> OutputSalesItem:
        try:
            input_sales_item_dict =
proto_to_dict(input_sales_item)

            input_sales_item = PydanticInputSalesItem.parse_obj(
                input_sales_item_dict
            )

            output_sales_item_dict = (
                self.__sales_item_service.create_sales_item(
                    input_sales_item
                ).dict()
            )

            output_sales_item = OutputSalesItem()

            json_format.ParseDict(
                output_sales_item_dict, output_sales_item
            )

            return output_sales_item
        except Exception as error:
            self.__abort_with(error, context)

    def getSalesItems(
        self, get_sales_items_arg: GetSalesItemsArg, context
    ) -> OutputSalesItems:
        try:
            # NOTE! Here we don't use the input message
            # 'get_sales_items_arg' because our current
            # business logic does not support it
            output_sales_items = (
                self.__sales_item_service.get_sales_items()
            )

            output_sales_items = [
                json_format.ParseDict(
```

```
                    output_sales_item.dict(), OutputSalesItem()
                )
                for output_sales_item in output_sales_items
            ]

            return
OutputSalesItems(salesItems=output_sales_items)
        except Exception as error:
            self.__abort_with(error, context)

    def getSalesItem(self, id: Id, context):
        try:
            output_sales_item_dict = (

self.__sales_item_service.get_sales_item(id.id).dict()
            )

            output_sales_item = OutputSalesItem()

            json_format.ParseDict(
                output_sales_item_dict, output_sales_item
            )

            return output_sales_item
        except Exception as error:
            self.__abort_with(error, context)

    def updateSalesItem(self, sales_item_update:
SalesItemUpdate, context):
        try:
            id_ = sales_item_update.id
            sales_item_update_dict =
proto_to_dict(sales_item_update)

            sales_item_update =
PydanticInputSalesItem.parse_obj(
                sales_item_update_dict
            )

            self.__sales_item_service.update_sales_item(
                id_, sales_item_update
            )

            return Nothing()
        except Exception as error:
            self.__abort_with(error, context)

    def deleteSalesItem(self, id: Id, context):
        try:
            self.__sales_item_service.delete_sales_item(id.id)
            return Nothing()
        except Exception as error:
            self.__abort_with(error, context)

    @staticmethod
    def __abort_with(error: Exception, context):
        status = create_status_from(error)
        context.abort_with_status(rpc_status.to_status(status))
```

For production, you must add audit logging and metrics updating to each gRPC procedure implementation. As shown in earlier examples, you can create and use decorators for those purposes. If your service is located in the backend and used by trusted backend clients only, you might not need authorization, but if you need it, you can plug in your own authorization mechanism, e.g., token-based.

Below is the gRPC server code:

Figure 8.131

```
import os
from concurrent import futures

import grpc

from .controllers.GrpcSalesItemController import
GrpcSalesItemController
from .DiContainer import DiContainer
from .grpc.sales_item_service_pb2_grpc import (
    add_SalesItemServiceServicer_to_server,
)

di_container = DiContainer()

# Remove this setting of env variable for production code!
# mysql+pymysql://root:password@localhost:3306/salesitemservice
# mongodb://localhost:27017/salesitemservice
os.environ[
    'DATABASE_URL'
] =
'mysql+pymysql://root:password@localhost:3306/salesitemservice'


def serve():
    server =
grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    add_SalesItemServiceServicer_to_server(
        GrpcSalesItemController(), server
    )
    server.add_insecure_port('[::]:50051')
    server.start()
    server.wait_for_termination()


serve()
```

You can run the server with the following command from a directory above the *salesitemservice* directory:

Figure 8.132

```
python -m salesitemservice.app_grpc
```

Below is an example of a gRPC client that performs operations using the above server:

Figure 8.133

```
from grpc_status import rpc_status

import grpc

from .grpc.sales_item_service_pb2 import (
    ErrorDetails,
    GetSalesItemsArg,
    Id,
    Image,
    InputSalesItem,
    SalesItemUpdate,
)
from .grpc.sales_item_service_pb2_grpc import
SalesItemServiceStub


def run():
    with grpc.insecure_channel('localhost:50051') as channel:
        sales_item_service = SalesItemServiceStub(channel)
        input_sales_item = InputSalesItem(
            name='Test',
            priceInCents=950,
            images=[
                Image(id=11, rank=1,
url='http://server.com/images/1')
            ],
        )

        try:
            sales_item = sales_item_service.createSalesItem(
                input_sales_item
            )

            id_ = sales_item.id
            print(f'Sales item with id {id_} created')

            sales_items_response =
sales_item_service.getSalesItems(
                GetSalesItemsArg()
            )

            print(
                f'Nbr of sales items fetched:
{len(sales_items_response.salesItems)}'
            )

            sales_item_service.updateSalesItem(
                SalesItemUpdate(
                    id=id_,
                    name='Test 2',
                    priceInCents=1950,
                    images=[
                        Image(
                            id=11, rank=1,
url='http://server.com/images/1'
                        )
```

```
                    ],
                )
            )
            print(f'Sales item with id {id_} updated')
            sales_item =
sales_item_service.getSalesItem(Id(id=id_))
            print(f'Sales item named {sales_item.name} fetched')
            sales_item_service.deleteSalesItem(Id(id=id_))
            print(f'Sales item with id {id_} deleted')
        except grpc.RpcError as error:
            status = rpc_status.from_call(error)
            if status:
                print(f'gRPC status code: {status.code}')
                for detail in status.details:
                    error_details = ErrorDetails()
                    detail.Unpack(error_details)
                    print(f'Error code: {error_details.code}')
                    print(f'Error message: {status.message}')
                    print(
                        f'Error description:
{error_details.description}'
                    )
            else:
                print(str(error))


if __name__ == '__main__':
    run()
```

You can run the client with the following command from a directory above the *salesitemservice* directory:

Figure 8.134

```
python -m salesitemservice.grpc_client
```

8.2.2: Asynchronous API Design Principle

> ***Use asynchronous APIs when requests are request-only (fire-and-forget, i.e., no response is expected) or when the response is not immediately expected.***

8.2.2.1: Request-Only Asynchronous API Design

In request-only asynchronous APIs, the request sender does not expect a response. Such APIs are typically implemented using a message broker. The request sender will send a JSON or other format request to a topic in the message broker, where the request recipient consumes the request asynchronously.

Different API endpoints can be specified in a request using a `procedure` property, for example. You can name the `procedure` property as you wish, e.g., `action`, `method`, "operation, `apiEndpointetc. Parameters for the procedure can be supplied in aparameters` property. Below is an example request in JSON:

Figure 8.135

```
{
  "procedure": "<procedure name>",
  "parameters": {
    "<parameterName1>": "parameter value 1",
    "<parameterName2>": "parameter value 2",
    // ...
  }
}
```

Let's have an example with an email-sending microservice that implements a request-only asynchronous API and handles the sending of emails. We start by defining a message broker topic for the microservice. The topic should be named after the microservice, for example, *email-sending-service*.

In the *email-sending-service*, we define the following request schema for an API endpoint that sends an email:

Figure 8.136

```
{
  "procedure": "sendEmailMessage",
  "parameters": {
    "fromEmailAddress": "...",
    "toEmailAddresses": ["...", "...", ...],
    "subject": "...",
    "message": "..."
  }
}
```

Below is an example request that some other microservice can produce to the *email-sending-service* topic in the message broker to be handled by the *email-sending-service*:

Figure 8.137

```
{
  "procedure": "sendEmailMessage",
  "parameters": {
    "fromEmailAddress": "sender@domain.com",
    "toEmailAddresses": ["receiver@domain.com"],
    "subject": "Status update",
    "message": "Hi, Here is my status update ..."
  }
}
```

8.2.2.2: Request-Response Asynchronous API Design

A request-response asynchronous API microservice receives requests from other microservices and then produces responses asynchronously. Request-response asynchronous APIs are typically implemented using a message broker. The request sender will send a request to a topic where the request recipient consumes the request asynchronously and then produces a response or responses to a message broker topic or topics. Each participating microservice should have a topic named after the microservice in the message broker.

The request format is the same as defined earlier, but the response has a `response` or `result` property instead of the `parameters` property, meaning that responses have the following format:

Figure 8.138

```
{
   "procedure": "<procedure name>",
   "response": {
     "propertyName1": "property value 1",
     "propertyName2": "property value 2",
     // ...
   }
}
```

Below is an example where a *loan-application-service* requests a *loan-eligibility-assessment-service* to assess a loan eligibility. The *loan-application-service* sends the following JSON-format request to the message broker's *loan-eligibility-assessment-service* topic:

Figure 8.139

```
{
  "procedure": "assessLoanEligibility",
  "parameters": {
    "userId": 123456789012,
    "loanApplicationId": 5888482223,
    // Other parameters...
  }
}
```

The *loan-eligibility-assessment-service* responds to the above request by sending the following JSON-format response to the message broker's *loan-application-service* topic:

Figure 8.140

```
{
  "procedure": "assessLoanEligibility",
  "response": {
     "loanApplicationId": 5888482223,
     "isEligible": true,
     "amountInDollars": 10000,
     "interestRate": 9.75,
     "termInMonths": 120
  }
}
```

Below is an example response when the loan application is rejected:

Figure 8.141

```
{
  "procedure": "assessLoanEligibility",
  "response": {
    "loanApplicationId": 5888482223,
    "isEligible": false
  }
}
```

Alternatively, request and response messages can be treated as events with some data. When we send events between microservices, we call it an event-driven architecture. For event-driven architecture, we must decide if we have a single or multiple topics for the software system in the message broker. If all the microservices share a single topic in the software system, then each microservice will consume each message from the message broker and decide if they should act on it. This approach is suitable except when large events are produced to the message broker. When large events are produced, each microservice must consume those large events even if they don't need to react to them. This will unnecessarily consume a lot of network bandwidth if the number of microservices is also high. The other extreme is to create a topic for each microservice in the message broker. This approach causes extra network bandwidth consumption if a large event must be produced to multiple topics to be handled by multiple microservices. You can also create a

hybrid model with a broadcast topic or topics and individual topics for specific microservices.

Below are the earlier request and response messages written as events:

Figure 8.142

```
{
  "event": "AssessLoanEligibility",
  "data": {
    "userId": 123456789012,
    "loanApplicationId": 5888482223,
    // ...
  }
}
```

Figure 8.143

```
{
  "event": "LoanApproved",
  "data": {
     "loanApplicationId": 5888482223,
     "isEligible": true,
     "amountInDollars": 10000,
     "interestRate": 9.75,
     "termInMonths": 120
  }
}
```

Figure 8.144

```
{
  "procedure": "LoanRejected",
  "response": {
    "loanApplicationId": 5888482223,
    "isEligible": false
  }
}
```

9: Databases And Database Principles

This chapter presents principles for selecting and using databases. Principles are presented for the following database types:

- Relational databases
- Document databases
- Key-value databases
- Wide column databases
- Search engines

Relational databases are also called SQL databases because accessing a relational database involves issuing SQL statements. Databases of the other database types are called NoSQL databases because they don't support SQL at all, or they support only a subset of it, possibly with some additions and modifications.

9.1: Relational Databases

> ***Relational databases are multipurpose databases that suit many needs. Choose a relational database if you are not aware of all the requirements you have for a database now or in the future.***

For example, if you don't know what kind of database table relations and queries you need now or will need in the future, you should consider using a relational database that is well-suited for different kinds of queries.

9.1.1: Structure of Relational Database

Data in a relational database is organized in the following hierarchy:

- Logical databases (or schemas)
    - Tables
        - Columns

A table consists of columns and rows. Data in a database is stored as rows in the tables. Each row has a value for each column in the table. A special 'NULL' value is used if a row does not have a value for a particular column. You can specify if null values are allowed for a column or not.

A microservice should have a single logical database (or schema). Some relational databases have one logical database (or schema) available by default and in other databases, you must create a logical database (or schema) by yourself.

9.1.2: Use Object Relational Mapper (ORM) Principle

> ***Use an object-relational mapper (ORM) to avoid the need to write SQL and to avoid making your microservice potentially vulnerable to SQL injection attacks. Use an ORM to automatically map the database rows to objects that can be serialized to JSON.***

This section presents examples using the SQLAlchemy library's ORM functionality. An ORM uses *entities* as building blocks for the database schema. Each entity class in a microservice is reflected as a table in the database. SQLAlchemy uses the term *model* instead of *entity*. I use the term entity in this book because it is a commonly used term, e.g., in domain-driven design. Use the same name for an entity and the database table, except the table name should be plural. Below is an example of a `SalesItem` entity class. Before defining the actual entity class(es), we need to declare a `Base` entity class:

Figure 9.1. Base.py

```
from sqlalchemy.orm import DeclarativeBase

class Base(DeclarativeBase):
    pass
```

Figure 9.2. SalesItem.py

```
from sqlalchemy import BigInteger, Double, String
from sqlalchemy.orm import Mapped
from sqlalchemy.orm import mapped_column

from Base import Base


class SalesItem(Base):
    __tablename__ = 'salesitems'

    id: Mapped[int] = mapped_column(
        BigInteger(), primary_key=True, autoincrement=True
    )

    name: Mapped[str] = mapped_column(String(256))
    price: Mapped[float] = mapped_column(Double())
```

Name the related table in the plural, e.g., `SalesItem` entities are stored in a table named `salesitems`. In this book, I write all identifiers in lowercase. The case sensitivity of a database depends on the database and the operating system it is running on. For example, MySQL is case-sensitive only on Linux systems.

The attributes of an entity map to columns of the entity table, meaning that the `salesitems` table has the following columns:

- id
- name
- price

Each entity table should have a primary key defined. The primary key must be unique for each row in the table. In the above example, we give the `primary_key=True` argument for the `mapped_column` function to define that the particular column should be a primary key and contain a unique value for each row. We also define that the database should generate an automatically incremented value for the `id` column (The default value for the `autoincrement` parameter is True, so it is not specified anymore in further examples).

ORM can create database tables according to entity definitions in code. Below is an example SQL statement that the ORM generates to create a table for storing `SalesItem` entities:

Figure 9.3

```
CREATE TABLE salesitems (
    id BIGINT NOT NULL AUTO_INCREMENT,
    name VARCHAR(256) NOT NULL,
    price DOUBLE NOT NULL,
    PRIMARY KEY (id)
)
```

Columns of a table can be specified as unique and nullable. We want to store sales items with unique names. Below is an example where we define the values of the `name` column in the `salesitems` table as unique (using the `UniqueConstraint`). We also add a `description` column that is nullable.

Figure 9.4

```
from typing import Optional

from Base import Base
from sqlalchemy import BigInteger, String, UniqueConstraint
from sqlalchemy.orm import Mapped, mapped_column


class SalesItem(Base):
    __tablename__ = 'salesitems'
    __table_args__ = (UniqueConstraint('name'))

    id: Mapped[int] = mapped_column(BigInteger(),
primary_key=True)
    name: Mapped[str] = mapped_column(String(256))
    price: Mapped[float] = mapped_column(Double())
    description: Mapped[Optional[str]] =
mapped_column(String(1024))
```

ORM generates the following SQL statement for creating the above-defined `salesitems` table:

Figure 9.5

```
CREATE TABLE salesitems (
    id BIGINT NOT NULL AUTO_INCREMENT,
    name VARCHAR(256) NOT NULL,
    price DOUBLE NOT NULL,
    description VARCHAR(1024),
    PRIMARY KEY (id),
    UNIQUE (name)
)
```

Let's create an entity and store it in the database. First, we have to create a database engine:

Figure 9.6

```
import os

from sqlalchemy import create_engine

engine = create_engine(os.environ.get('DATABASE_URL'))
```

For demonstration purposes, we could use an in-memory SQLite database:

Figure 9.7

```
from sqlalchemy import create_engine

engine = create_engine('sqlite://', echo=True)
```

The `echo=True` parameter defines that SQL statements generated and used by the ORM will be logged to standard output. This is handy for debugging purposes. After we have created the database engine, we must create the database tables in the database. That can be done using the following command:

Figure 9.8

```
from Base import Base


Base.metadata.create_all(engine)
```

Next, we can create a sales item and persist it to the database:

Figure 9.9

```
from SalesItem import SalesItem
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy.orm import Session

sales_item = SalesItem(name='Sample sales item', price='10')

try:
    with Session(engine) as session:
        session.add(sales_item)
        session.commit()
except SQLAlchemyError:
    # Handle error
```

ORM will generate the needed SQL statement on your behalf and execute it. Below is an example SQL statement generated by the ORM to persist a sales item (Remember that the database autogenerates the `id` column).

Figure 9.10

```
INSERT INTO salesitems (name, price)
VALUES ('Sample sales item', 10)
```

You can search for the created sales item in the database:

Figure 9.11

```
from SalesItem import SalesItem
from sqlalchemy import select
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy.orm import Session

statement = select(SalesItem).where(SalesItem.id ==
sales_item.id)

try:
    with Session(engine) as session:
        sales_item = session.scalars(statement).one()
except SQLAlchemyError:
    # Handle error
```

For the above operation, the ORM will generate the following SQL query:

Figure 9.12

```
SELECT id, name, price, description FROM salesitems WHERE id = 1
```

Then, you can modify the entity by setting the values of its attributes and using `commit` on the `session` to update the database:

Figure 9.13

```
from SalesItem import SalesItem
from sqlalchemy import select
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy.orm import Session

try:
    with Session(engine) as session:
        sales_item = session.get(SalesItem, 1)
        sales_item.price = 20
        session.commit()
except SQLAlchemyError:
    # Handle error
```

For the above operation, the ORM will generate the following SQL statement:

Figure 9.14

```
UPDATE salesitems SET price = 20 WHERE id = 1
```

Lastly, you can delete the sales item:

Figure 9.15

```
try:
    with Session(engine) as session:
        sales_item = session.get(SalesItem, 1)
        session.delete(sales_item)
        session.commit()
except SQLAlchemyError:
    # Handle error
```

The ORM will execute the following SQL statement:

Figure 9.16

```
DELETE FROM salesitems WHERE id = 1
```

Suppose your microservice executes SQL queries that do not include the primary key column in the query's WHERE clause. In that case, the database engine must perform a full table scan to find the wanted rows. Let's say you want to query sales items, the price of which is less than 10. This can be achieved with the below query:

Figure 9.17

```
# price = ...

statement = select(SalesItem).where(SalesItem.price < price)

try:
    with Session(engine) as session:
        sales_items = session.scalars(statement).all()
except SQLAlchemyError:
    # Handle error
```

The database engine must perform a full table scan to find all the sales items where the `price` column has a value below the `price` variable's value. If the database is large, this can be slow. If you perform the above query often, you should optimize those queries by creating an index. For the above query to be fast, we must create an index for the `price` column:

Figure 9.18

```
from typing import Optional

from Base import Base
from sqlalchemy import BigInteger, Double, String,
UniqueConstraint
from sqlalchemy.orm import Mapped, mapped_column


class SalesItem(Base):
```

```
    __tablename__ = 'salesitems'
    __table_args__ = (UniqueConstraint('name'),)

    id: Mapped[int] = mapped_column(BigInteger(),
primary_key=True)
    name: Mapped[str] = mapped_column(String(256))
    price: Mapped[float] = mapped_column(Double(), index=True)
    description: Mapped[Optional[str]] =
mapped_column(String(1024))
```

9.1.3: Entity/Table Relationships

Tables in a relational database can have relationships with other tables. There are three types of relationships:

- One-to-one
- One-to-many
- Many-to-many

9.1.3.1: One-To-One/Many Relationships

In this section, we focus on one-to-one and one-to-many relationships. In a one-to-one relationship, a single row in a table can have a relationship with another row in another table. In a one-to-many relationship, a single row in a table can have a relationship with multiple rows in another table.

Let's have an example with an *order-service* that can store orders in a database. Each order consists of one or more order items. An order item contains information about the bought sales item.

Figure 9.19

```
from Base import Base
from sqlalchemy import BigInteger, ForeignKey
from sqlalchemy.orm import Mapped, mapped_column, relationship


class Order(Base):
    __tablename__ = 'orders'

    id: Mapped[int] = mapped_column(BigInteger(),
primary_key=True)
    # Other fields ...
    items: Mapped[list['OrderItem']] =
relationship(lazy='joined')
```

```
class OrderItem(Base):
    __tablename__ = 'orderitems'
    __table_args__ = (
        PrimaryKeyConstraint('orderId', 'id',
name='orderitems_pk'),
    )

    id: Mapped[int]
    salesitemid: Mapped[int] = mapped_column(BigInteger())
    orderid: Mapped[int] =
mapped_column(ForeignKey('orders.id'))
```

Orders are stored in the `orders` table, and order items are stored in the `orderitems` table, which contains a join column named `orderid`. Using this join column, we can map a particular order item to a specific order. Each order item maps to precisely one sales item. For this reason, the `orderitems` table also contains a column named `salesitemid`. Sales items are stored in a different database in a separate microservice (*sales-item-service*), so there are no intra-database inter-table relationships with the *order-service* and *sales-item-service*.

Below is the SQL statement generated by the ORM for creating the `orderitems` table. The one-to-many relationship is reflected in the foreign key constraint:

Figure 9.20

```
CREATE TABLE orderitems (
    id INTEGER NOT NULL,
    salesitemid BIGINT NOT NULL,
    orderid BIGINT NOT NULL,
    CONSTRAINT orderitems_pk PRIMARY KEY (orderid, id),
    FOREIGN KEY (orderid) REFERENCES orders (id)
)
```

The ORM executes the below SQL query to fetch the order with id 123 and its order items:

Figure 9.21

```
SELECT o.id, oi.id
FROM orders o
LEFT JOIN orderitems oi ON o.id = oi.orderid
WHERE o.id = 123
```

9.1.3.2: Many-To-Many Relationships

In a many-to-many relationship, one entity has a relationship with many entities of another type, and those entities have a

relationship with many entities of the first entity type. For example, a student can attend many courses, and a course can have numerous students attending it.

Suppose we have a service that stores student and course entities in a database. Each student entity contains the courses the student has attended. Similarly, each course entity contains a list of students that have attended the course. We have a many-to-many relationship where one student can attend multiple courses, and multiple students can attend one course. This means an additional association/associative table, `studentcourse`, must be created. This new table maps a particular student to a particular course.

Figure 9.22

```
from sqlalchemy import BigInteger Column, ForeignKey, Table
from sqlalchemy.orm import (
    DeclarativeBase,
    Mapped,
    mapped_column,
    relationship,
)


class Base(DeclarativeBase):
    pass


student_course_assoc_table = Table(
    'studentcourse',
    Base.metadata,
    Column('studentid', ForeignKey('students.id'),
primary_key=True),
    Column('courseid', ForeignKey('courses.id'),
primary_key=True),
)


class Student(Base):
    __tablename__ = 'students'

    id: Mapped[int] = mapped_column(BigInteger(),
primary_key=True)
    # Other fields...

    courses: Mapped[list['Course']] = relationship(
        secondary=student_course_assoc_table,
back_populates='students'
    )


class Course(Base):
    __tablename__ = 'courses'

    id: Mapped[int] = mapped_column(BigInteger(),
```

```
primary_key=True)
    # Other fields...

    students: Mapped[list[Student]] = relationship(
        secondary=student_course_assoc_table,
back_populates='courses'
    )
```

The ORM creates the `students` and `courses` tables in addition to the `studentcourse` mapping table:

Figure 9.23

```
CREATE TABLE studentcourse (
  studentid BIGINT NOT NULL,
  courseid BIGINT NOT NULL,
  PRIMARY KEY (studentid, courseid),
  FOREIGN KEY (studentid) REFERENCES students (id),
  FOREIGN KEY (courseid) REFERENCES courses (id)
)
```

Below is an example SQL query that the ORM executes to fetch attended courses for the user identified with id 123:

Figure 9.24

```
SELECT s.id, c.id
FROM students s
LEFT JOIN studentcourse sc ON s.id = sc.studentid
LEFT JOIN courses c ON c.id = sc.courseid
WHERE s.id = 123
```

Below is an example SQL query that the ORM executes to fetch students for the course identified with id 123:

Figure 9.25

```
SELECT c.id, s.id
FROM courses c
LEFT JOIN studentcourse sc ON c.id = sc.courseid
LEFT JOIN students s ON s.id = sc.studentid
WHERE c.id = 123
```

In real-life scenarios, you don't necessarily have to or should implement many-to-many database relations inside a single microservice. For example, the above service that handles students and courses is against the *single responsibility principle* on the abstraction level of courses and students. (However, if we created a *school* microservice on a higher abstraction level, we can have students and courses tables in the same microservice) If we created a separate microservice for students and a separate microservice for courses, then there

wouldn't be many-to-many relationships between database tables in a single microservice.

9.1.3.3: Sales Item Repository Example

Let's define a `SalesItemRepository` implementation using SQLAlchemy's ORM capabilities for the *sales-item-service* API defined in the previous chapter. Let's start by defining the `Base`, `SalesItem`, and `SalesItemImage` entities:

Figure 9.26

```
from sqlalchemy.orm import DeclarativeBase


class Base(DeclarativeBase):
    pass
```

Figure 9.27

```
from sqlalchemy import BigInteger, Double, String
from sqlalchemy.orm import Mapped, mapped_column, relationship

from .Base import Base
from .SalesItemImage import SalesItemImage


class SalesItem(Base):
    __tablename__ = 'salesitems'

    id: Mapped[int] = mapped_column(BigInteger(),
primary_key=True)
    createdAtTimestampInMs: Mapped[int] =
mapped_column(BigInteger())
    name: Mapped[str] = mapped_column(String(256))
    priceInCents: Mapped[int]
    images: Mapped[list[SalesItemImage]] = relationship(
        cascade='all, delete-orphan', lazy='joined'
    )
```

Figure 9.28

```
from sqlalchemy import ForeignKey, PrimaryKeyConstraint, String
from sqlalchemy.orm import Mapped, mapped_column

from .Base import Base


class SalesItemImage(Base):
    __tablename__ = 'salesitemimages'
    __table_args__ = (
        PrimaryKeyConstraint(
            'salesItemId', 'id', name='salesitemimages_pk'
        ),
    )

    id: Mapped[int]
```

```
    rank: Mapped[int]
    url: Mapped[str] = mapped_column(String(2084))
    salesItemId: Mapped[int] =
mapped_column(ForeignKey('salesitems.id'))
```

Below is the implementation of the `OrmSalesItemRepository`:

Figure 9.29

```
import os
import time

from sqlalchemy import create_engine, select
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy.orm import sessionmaker

from ..dtos.InputSalesItem import InputSalesItem
from ..entities.Base import Base
from ..entities.SalesItem import SalesItem
from ..errors.DatabaseError import DatabaseError
from ..errors.EntityNotFoundError import EntityNotFoundError
from ..utils import to_entity_dict
from .SalesItemRepository import SalesItemRepository


class OrmSalesItemRepository(SalesItemRepository):
    def __init__(self):
        try:
            engine =
create_engine(os.environ.get('DATABASE_URL'))
            self.__SessionLocal = sessionmaker(
                autocommit=False, autoflush=False, bind=engine
            )
            Base.metadata.create_all(bind=engine)
        except SQLAlchemyError as error:
            # Log error
            raise error

    def save(self, input_sales_item: InputSalesItem) ->
SalesItem:
        try:
            with self.__SessionLocal() as db_session:
                sales_item =
SalesItem(**to_entity_dict(input_sales_item))
                sales_item.createdAtTimestampInMs = (
                    time.time_ns() / 1_000_000
                )
                db_session.add(sales_item)
                db_session.commit()
                db_session.refresh(sales_item)
                return sales_item
        except SQLAlchemyError as error:
            raise DatabaseError(error)

    def find_all(self) -> list[SalesItem]:
        try:
            with self.__SessionLocal() as db_session:
                return
db_session.scalars(select(SalesItem)).unique().all()
```

```
        except SQLAlchemyError as error:
            raise DatabaseError(error)

    def find(self, id_: str) -> SalesItem | None:
        try:
            with self.__SessionLocal() as db_session:
                return db_session.get(SalesItem, id_)
        except SQLAlchemyError as error:
            raise DatabaseError(error)

    def update(self, id_: str, sales_item_update:
InputSalesItem) -> None:
        try:
            with self.__SessionLocal() as db_session:
                sales_item = db_session.get(SalesItem, id_)

                if sales_item is None:
                    raise EntityNotFoundError('Sales item', id_)

                new_sales_item = SalesItem(
                    **to_entity_dict(sales_item_update)
                )

                sales_item.name = new_sales_item.name
                sales_item.priceInCents =
new_sales_item.priceInCents
                sales_item.images = new_sales_item.images
                db_session.commit()
        except SQLAlchemyError as error:
            raise DatabaseError(error)

    def delete(self, id_: str) -> None:
        try:
            with self.__SessionLocal() as db_session:
                sales_item = db_session.get(SalesItem, id_)
                if sales_item is not None:
                    db_session.delete(sales_item)
                    db_session.commit()
        except SQLAlchemyError as error:
            raise DatabaseError(error)
```

9.1.4: Use Parameterized SQL Statements Principle

> ***If you are not using an ORM for database access, use parameterized SQL statements to prevent potential SQL injection attacks.***

Let's use the Python MySQL connector library [mysql-connector-python]. First, we insert data to the `salesitems` table:

Figure 9.30

```
from mysql.connector import connect, Error

connection = None

try:
    connection = connect(
        host='...',
        database='...',
        user='...',
        password='...'
    )

    cursor = connection.cursor(prepared=True)
    sql_statement = 'INSERT INTO salesitems (name, price) VALUES
(%s, %s)'
    cursor.execute(sql_statement, ('Sample sales item 1', 20))
    connection.commit()
except Error as error:
    # Handle error
finally:
    if connection:
        connection.close()
```

The `%s` in the above SQL statement are placeholders for parameters in a parameterized SQL statement. The second argument for the `execute` method contains the parameter values in a tuple. When a database engine receives a parameterized query, it will replace the placeholders in the SQL statement with the supplied parameter values.

Next, we can update a row in the `salesitems` table. The below example changes the price of the sales item with id 123 to 20:

Figure 9.31

```
from mysql.connector import connect, Error

connection = None

try:
    connection = connect(
        host='...',
        database='...',
        user='...',
        password='...'
    )

    cursor = connection.cursor(prepared=True)
    sql_statement = 'UPDATE salesitems SET PRICE = %s WHERE id =
%s'
    cursor.execute(sql_statement, (20, 123))
    connection.commit()
except Error as error:
    # Handle error
finally:
    if connection:
        connection.close()
```

Let's execute a SELECT statement to get sales items with their price over 20:

Figure 9.32

```
from mysql.connector import connect, Error

connection = None

try:
    connection = connect(
        host='...',
        database='...',
        user='...',
        password='...'
    )

    cursor = connection.cursor(prepared=True)
    sql_statement = 'SELECT id, name, price FROM salesitems
WHERE price >= %s'
    cursor.execute(sql_statement, (20,))
    result = cursor.fetchall()
except Error as error:
    # Handle error
finally:
    if connection:
        connection.close()
```

In an SQL SELECT statement, you cannot use parameters everywhere. You can use them as value placeholders in the WHERE clause. You need string concatenation to put user-supplied data in other parts of an SQL SELECT statement. You should not concatenate user-supplied data without sanitation because that would open up possibilities for SQL injection attacks. Let's say you allow the microservice client to specify a sorting column:

Figure 9.33

```
import string


class ValidateColNameError(Exception):
    pass


def try_validate_col_name(column_name: str) -> str:
    allowed_chars = string.ascii_letters + string.digits + '_' +
'$'

    if all(
        col_name_char in allowed_chars for col_name_char in
column_name
    ):
        return column_name

    raise ValidateColNameError()
```

```
sort_column_name = # Unvalidated data got from client

sql_query = (
    'SELECT id, name, price FROM salesitems ORDER BY '
    + try_validate_col_name(sort_column_name)
)

# ...
```

As shown above, you must validate the `sort_column` value to contain only valid characters for a MySQL column name. If you need to get the sorting direction from the client, you should validate that value as `ASC` or `DESC`.

Figure 9.34

```
class ValidateSortDirError(Exception):
    pass


def try_validate_sort_dir(sort_dir: str) -> str:
    lower_case_sort_dir = sort_dir.lower()

    if lower_case_sort_dir == 'asc' or lower_case_sort_dir ==
'desc':
        return sort_dir

    raise ValidateSortDirError()


sort_column_name = # Unvalidated data got from client
sort_direction = # Unvalidated data got from client

sql_query = (
  'SELECT id, name, price'
  'FROM salesitems'
  'ORDER BY'
  f'{try_validate_col_name(sort_column_name)}'
  f'{try_validate_sort_dir(sort_direction)'
)

# ...
```

When you get values for a MySQL query's `LIMIT` clause from a client, you must validate that those values are integers and in a valid range. Don't allow the client to supply random, very large values. In the example below, we assume two validation functions exist: `try_validate_row_offset` and `try_validate_row_count`. The validation functions will raise an error if validation fails.

Figure 9.35

```
def try_validate_row_offset(row_offset: str) -> str:
    # Implement ...

def try_validate_row_count(row_count: str) -> str:
    # Implement ...

row_offset = # Unvalidated data got from client
row_count = # Unvalidated data got from client

try_validate_row_offset(row_offset)
try_validate_row_count(row_count)

sql_query = (
  'SELECT id, name, price'
  'FROM salesitems'
  f'LIMIT {row_offset}, {row_count}'
)

# ...
```

When you get a list of wanted column names from a client, you must validate that each of them is a valid column identifier:

Figure 9.36

```
column_names = # Unvalidated data got from client
validated_col_names = [try_validate_col_name(column_name) for
column_name in column_names]
sql_query = f'SELECT {", ".join(validated_col_names)} FROM
salesitems'

# ...
```

Let's implement the `SalesItemRepository` for the *sales-item-service* API from the previous chapter using parameterized SQL:

Figure 9.37

```
import os
import time
from typing import Any

from mysql.connector import connect
from mysql.connector.errors import Error

from ..dtos.InputSalesItem import InputSalesItem
from ..entities.SalesItem import SalesItem
from ..entities.SalesItemImage import SalesItemImage
from ..errors.DatabaseError import DatabaseError
from ..errors.EntityNotFoundError import EntityNotFoundError
from ..utils import to_entity_dict
from .SalesItemRepository import SalesItemRepository


class ParamSqlSalesItemRepository(SalesItemRepository):
    def __init__(self):
```

```
        try:
            self.__conn_config = self.__try_create_conn_config()
            self.__try_create_db_tables_if_needed()
        except Exception as error:
            # Log error
            raise (error)

    def save(self, input_sales_item: InputSalesItem) ->
SalesItem:
        connection = None

        try:
            connection = connect(**self.__conn_config)
            cursor = connection.cursor(prepared=True)

            sql_statement = (
                'INSERT INTO salesitems'
                '(createdAtTimestampInMs, name, priceInCents)'
                ' VALUES (%s, %s, %s)'
            )

            created_at_timestamp_in_ms = time.time_ns() /
1_000_000

            cursor.execute(
                sql_statement,
                (
                    created_at_timestamp_in_ms,
                    input_sales_item.name,
                    input_sales_item.priceInCents,
                ),
            )

            id_ = cursor.lastrowid

            self.__try_insert_sales_item_images(
                id_, input_sales_item.images, cursor
            )

            connection.commit()

            return SalesItem(
                **to_entity_dict(input_sales_item),
                id=id_,

createdAtTimestampInMs=created_at_timestamp_in_ms,
            )
        except Error as error:
            raise DatabaseError(error)
        finally:
            if connection:
                connection.close()

    def find_all(self) -> list[SalesItem]:
        connection = None

        try:
            connection = connect(**self.__conn_config)
            cursor = connection.cursor()

            sql_statement = (
                'SELECT s.id, s.createdAtTimestampInMs, s.name,
```

```
s.priceInCents,'
                'si.id, si.rank, si.url '
                'FROM salesitems s '
                'LEFT JOIN salesitemimages si ON si.salesItemId
= s.id'
            )

            cursor.execute(sql_statement)
            return self.__get_sales_item_entities(cursor)
        except Error as error:
            print(error)
            raise DatabaseError(error)
        finally:
            if connection:
                connection.close()

    def find(self, id_: str) -> SalesItem | None:
        if not id_.isnumeric():
            raise EntityNotFoundError('Sales item', id_)

        connection = None

        try:
            connection = connect(**self.__conn_config)
            cursor = connection.cursor(prepared=True)

            sql_statement = (
                'SELECT s.id, s.createdAtTimestampInMs, s.name,
s.priceInCents,'
                'si.id, si.rank, si.url '
                'FROM salesitems s '
                'LEFT JOIN salesitemimages si ON si.salesItemId
= s.id '
                'WHERE s.id = %s'
            )

            cursor.execute(sql_statement, (id_,))

            sales_item_entities =
self.__get_sales_item_entities(cursor)
            return sales_item_entities[0] if sales_item_entities
else None
        except Error as error:
            raise DatabaseError(error)
        finally:
            if connection:
                connection.close()

    def update(self, id_: str, sales_item_update:
InputSalesItem) -> None:
        if not id_.isnumeric():
            raise EntityNotFoundError('Sales item', id_)

        connection = None

        try:
            connection = connect(**self.__conn_config)
            cursor = connection.cursor(prepared=True)

            sql_statement = (
                'UPDATE salesitems SET name = %s, priceInCents =
%s '
```

```
                'WHERE id = %s'
            )

            cursor.execute(
                sql_statement,
                (
                    sales_item_update.name,
                    sales_item_update.priceInCents,
                    id_,
                ),
            )

            sql_statement = (
                'DELETE FROM salesitemimages WHERE salesItemId =
%s'
            )

            cursor.execute(sql_statement, (id_,))

            self.__try_insert_sales_item_images(
                id_, sales_item_update.images, cursor
            )

            connection.commit()
        except Error as error:
            raise DatabaseError(error)
        finally:
            if connection:
                connection.close()

    def delete(self, id_: str) -> None:
        if not id_.isnumeric():
            return

        connection = None

        try:
            connection = connect(**self.__conn_config)
            cursor = connection.cursor()

            sql_statement = (
                'DELETE FROM salesitemimages WHERE salesItemId =
%s'
            )

            cursor.execute(sql_statement, (id_,))
            sql_statement = 'DELETE FROM salesitems WHERE id =
%s'
            cursor.execute(sql_statement, (id_,))
            connection.commit()
        except Error as error:
            raise DatabaseError(error)
        finally:
            if connection.is_connected():
                connection.close()

    @staticmethod
    def __try_create_conn_config() -> dict[str, Any]:
        database_url = os.environ.get('DATABASE_URL')

        user_and_password = (
            database_url.split('@')[0].split('//')[1].split(':')
```

```
        )

        host_and_port = database_url.split('@')[1].split('/')
[0].split(':')
        database = database_url.split('/')[3]

        return {
            'user': user_and_password[0],
            'password': user_and_password[1],
            'host': host_and_port[0],
            'port': host_and_port[1],
            'database': database,
            'pool_name': 'salesitems',
            'pool_size': 25,
        }

    def __try_create_db_tables_if_needed(self) -> None:
        connection = connect(**self.__conn_config)
        cursor = connection.cursor()

        sql_statement = (
            'CREATE TABLE IF NOT EXISTS salesitems ('
            'id BIGINT NOT NULL AUTO_INCREMENT,'
            'createdAtTimestampInMs BIGINT NOT NULL,'
            'name VARCHAR(256) NOT NULL,'
            'priceInCents INTEGER NOT NULL,'
            'PRIMARY KEY (id)'
            ')'
        )

        cursor.execute(sql_statement)

        sql_statement = (
            'CREATE TABLE IF NOT EXISTS salesitemimages ('
            'id BIGINT NOT NULL,'
            '`rank` INTEGER NOT NULL,'
            'url VARCHAR(2084) NOT NULL,'
            'salesItemId BIGINT NOT NULL,'
            'PRIMARY KEY (salesItemId, id),'
            'FOREIGN KEY (salesItemId) REFERENCES
salesitems(id)'
            ')'
        )

        cursor.execute(sql_statement)
        connection.commit()
        connection.close()

    def __try_insert_sales_item_images(
        self, sales_item_id: str | int, images, cursor
    ):
        for image in images:
            sql_statement = (
                'INSERT INTO salesitemimages'
                '(id, `rank`, url, salesItemId)'
                'VALUES (%s, %s, %s, %s)'
            )

            cursor.execute(
                sql_statement,
                (image.id, image.rank, image.url,
sales_item_id),
```

```
            )

    def __get_sales_item_entities(self, cursor):
        id_to_sales_item_dict = {}

        for (
            id_,
            created_at_timestamp_in_ms,
            name,
            price_in_cents,
            image_id,
            image_rank,
            image_url,
        ) in cursor:
            if id_to_sales_item_dict.get(id_) is None:
                id_to_sales_item_dict[id_] = {
                    'id': id_,
                    'createdAtTimestampInMs':
created_at_timestamp_in_ms,
                    'name': name,
                    'priceInCents': price_in_cents,
                    'images': [],
                }

            if image_id is not None:
                id_to_sales_item_dict[id_]['images'].append(
                    SalesItemImage(
                        id=image_id, rank=image_rank,
url=image_url
                    )
                )

        return [
            SalesItem(**sales_item_dict)
            for sales_item_dict in
id_to_sales_item_dict.values()
        ]
```

9.1.5: Normalization Rules

Apply normalization rules to your database design.

Below are listed the three most basic database normalization rules:

- First normal form (1NF)
- Second normal form (2NF)
- Third normal form (3NF)

A database relation is often described as “normalized” if it meets the first, second, and third normal forms.

9.1.5.1: First Normal Form (1NF)

The first normal form requires that a single value exists at every intersection of a row and column, never a list of values. When considering a sales item, the first normal form states that there cannot be two different price values in the `price` column or more than one name for the sales item in the `name` column. If you need multiple names for a sales item, you must establish a one-to-many relationship between a `SalesItem` entity and `SalesItemName` entities. What this means in practice is that you remove the `name` property from the `SalesItem` entity class and create a new `SalesItemName` entity class used to store sales items' names. Then, you create a one-to-many mapping between a `SalesItem` entity and `SalesItemName` entities.

9.1.5.2: Second Normal Form (2NF)

The second normal form requires that each non-key column entirely depends on the primary key. Let's assume that we have the following columns in an `orderitems` table:

- `orderid` (primary key)
- `productid` (primary key)
- `orderstate`

The `orderstate` column only depends on the `orderid` column, not the entire primary key. The `orderstate` column is in the wrong table. It should, of course, be in the `orders` table.

9.1.5.3: Third Normal Form (3NF)

The third normal form requires that non-key columns are independent of each other.

Let's assume that we have the following columns in a `salesitems` table:

- `id` (primary key)

- `name`
- `price`
- `category`
- `discount`

Let's assume that the discount depends on the category. This table violates the third normal form because a non-key column, `discount`, depends on another non-key column, `category`. Column independence means that you can change any non-key column value without affecting any other column. If you changed the category, the discount would need to be changed accordingly, thus violating the third normal form rule.

The discount column should be moved to a new `categories` table with the following columns:

- `id` (primary key)
- `name`
- `discount`

Then we should update the `salesitems` table to contain the following columns:

- `id` (primary key)
- `name`
- `price`
- `categoryid` (a foreign key that references the `id` column in the `categories` table)

9.2: Document Databases

Use a document database in cases where complete documents (e.g., JSON objects) are typically stored and retrieved as a whole.

Document databases, like MongoDB, are useful for storing complete documents. A document is usually a JSON object containing information in arrays and nested objects. Documents are stored as such, and a whole document will be fetched when queried.

Let's consider a microservice for sales items. Each sales item contains an id, name, price, image URLs, and user reviews.

Below is an example sales item as a JSON object:

Figure 9.38

```
{
  "id": "507f191e810c19729de860ea",
  "category": "Power tools",
  "name": "Sample sales item",
  "price": 10,
  "imageUrls": ["https://url-to-image-1...",
                "https://url-to-image-2..."],
  "averageRatingInStars": 5,
  "reviews": [
     {
       "reviewerName": "John Doe",
       "date": "2022-09-01",
       "ratingInStars": 5,
       "text": "Such a great product!"
     }
  ]
}
```

A document database usually has a size limit for a single document. Therefore, the above example does not store sales item images directly inside the document but only URLs for the images. Actual images are stored in another data store more suitable for storing images, like Amazon S3.

When creating a microservice for sales items, we can choose a document database because we usually store and access whole documents. When sales items are created, they are created as JSON objects of the above shape with the `reviews` array being empty and `averageRatingInStars` being `null`. When a sales item is fetched, the whole document is retrieved from the database. When a client adds a review for a sales item, the sales item is fetched from the database. The new review is appended to the `reviews` array, a new average rating is calculated, and finally, the document is persisted with the modifications.

Below is an example of inserting one sales item to a MongoDB collection named `salesItems` using the PyMongo library. MongoDB uses the term collection instead of *table*. A MongoDB collection can store multiple documents.

Figure 9.39

```
from pymongo import MongoClient

URL = "mongodb://localhost:27017"
client = MongoClient(URL)

# Create the database for our example
database = client['sales_item_service']
sales_items_coll = database['sales_items']

sales_items_coll.insert_one({
  'category': 'Power tools',
  'name': 'Sample sales item 1',
  'price': 10,
  'images': ['https://url-to-image-1...',
             'https://url-to-image-2...'],
  'averageRatingInStars': None,
  'reviews': []
})

client.close()
```

You can find sales items for the *Power tools* category with the following query:

Figure 9.40

```
sales_items = sales_items_coll.find({ 'category': 'Power tools'
})
print(sales_items.next())
```

If clients are usually querying sales items by category, it is wise to create an index for that field:

Figure 9.41

```
# 1 means ascending index, -1 means descending index
sales_items_coll.create_index([('category', 1)])
```

When a client wants to add a new review for a sales item, you first fetch the document for the sales item:

Figure 9.42

```
sales_items_coll.find({ '_id':
ObjectId('507f191e810c19729de860ea') })
```

Then, you calculate a new value for the `averageRatingInStars` field using the existing ratings and the

new rating and add the new review to the `reviews` array and then update the document with the following command:

Figure 9.43

```
sales_items_coll.update_one(
    {'_id': ObjectId('6527a461bd3c27d2d1822508')},
    {
        '$set': {'averageRatingInStars': 5},
        '$push': {
            'reviews': {
                'reviewerName': 'John Doe',
                'date': '2022-09-01',
                'ratingInStars': 5,
                'text': 'Such a great product!',
            }
        },
    },
)
```

Clients may want to retrieve sales items sorted descending by the average rating. For this reason, you might want to change the indexing to the following:

Figure 9.44

```
sales_items_coll.create_index( [ ('category', 1),
('averageRatingInStars', -1) ] )
```

A client can issue, for example, a request to get the best-rated sales items in the *power tools* category. This request can be fulfilled with the following query that utilizes the above-created index:

Figure 9.45

```
sales_items_coll.find({'category': 'Power tools'}).sort(
    [('averageRatingInStars', -1)]
)
```

Let's implement the `SalesItemRepository` for the *sales-item-service* API from the previous chapter using MongoDB:

Figure 9.46

```
import os
import time
from typing import Any

from bson.errors import BSONError, InvalidId
from bson.objectid import ObjectId
from pymongo import MongoClient
from pymongo.errors import PyMongoError

from ..dtos.InputSalesItem import InputSalesItem
from ..entities.SalesItem import SalesItem
```

```python
from ..entities.SalesItemImage import SalesItemImage
from ..errors.DatabaseError import DatabaseError
from ..errors.EntityNotFoundError import EntityNotFoundError
from .SalesItemRepository import SalesItemRepository


class MongoDbSalesItemRepository(SalesItemRepository):
    def __init__(self):
        try:
            database_url = os.environ.get('DATABASE_URL')
            self.__client = MongoClient(database_url)
            database_name = database_url.split('/')[3]
            database = self.__client[database_name]
            self.__sales_items = database['salesitems']
        except Exception as error:
            # Log error
            raise (error)

    def save(self, input_sales_item: InputSalesItem) ->
SalesItem:
        try:
            sales_item = input_sales_item.dict() | {
                'createdAtTimestampInMs': time.time_ns() /
1_000_000
            }

            self.__sales_items.insert_one(sales_item)
            return self.__create_sales_item_entity(sales_item)

        except PyMongoError as error:
            raise DatabaseError(error)

    def find_all(self) -> list[SalesItem]:
        try:
            sales_items = self.__sales_items.find()
            return [
                self.__create_sales_item_entity(sales_item)
                for sales_item in sales_items
            ]
        except PyMongoError as error:
            raise DatabaseError(error)

    def find(self, id_: str) -> SalesItem | None:
        try:
            sales_item = self.__sales_items.find_one(
                {'_id': ObjectId(id_)}
            )

            return (
                None
                if sales_item is None
                else self.__create_sales_item_entity(sales_item)
            )
        except InvalidId:
            raise EntityNotFoundError('Sales item', id_)
        except (BSONError, PyMongoError) as error:
            raise DatabaseError(error)

    def update(self, id_: str, sales_item_update:
InputSalesItem) -> None:
        try:
            self.__sales_items.update_one(
```

```
                {'_id': ObjectId(id_)}, {'$set':
sales_item_update.dict()}
            )
        except InvalidId:
            raise EntityNotFoundError('Sales item', id_)
        except (BSONError, PyMongoError) as error:
            raise DatabaseError(error)

    def delete(self, id_: str) -> None:
        try:
            self.__sales_items.delete_one({'_id':
ObjectId(id_)})
        except InvalidId:
            pass
        except (BSONError, PyMongoError) as error:
            raise DatabaseError(error)

    @staticmethod
    def __create_sales_item_entity(sales_item: dict[str, Any]):
        id_ = sales_item['_id']
        del sales_item['_id']

        images = [
            SalesItemImage(**image) for image in
sales_item['images']
        ]

        return SalesItem(
            **(sales_item | {'id': str(id_)} | {'images':
images})
        )
```

9.3: Key-Value Database Principle

> ***Use a key-value database for fast real-time access to data stored by a key. Key-value stores usually store data in memory with a possibility for persistence.***

A simple use case for a key-value database is to use it as a cache for a relational database. For example, a microservice can store SQL query results from a relational database in the cache. Redis is a popular open-source key-value store. Let's have an example with Redis to cache an SQL query result. In the below example, we assume that the SQL query result is available as a dict:

Figure 9.47

```
import json
from redis import Redis


redis_client = Redis(host='localhost', port=6379,
```

```
decode_responses=True)
redis_client.set(sql_query_statement,
json.dumps(sql_query_result))
```

The cached SQL query result can be fetched from Redis:

Figure 9.48

```
sql_query_result_json = redis_client.get(sql_query_statement)
```

With Redis, you can create key-value pairs that expire automatically after a specified time. This is a useful feature if you are using the key-value database as a cache. You may want the cached items to expire after a while.

In addition to plain strings, Redis also supports other data structures. For example, you can store a list, queue, or hash map as a value for a key in Redis. If you store a queue in Redis, you can use it as a simple single-consumer message broker. Below is an example of producing a message to a topic in the message broker:

Figure 9.49

```
# RPUSH command (= right push) pushes a new message
# to the end of the list identified by key *topic*.
redis_client.rpush(topic, message)
```

Below is an example of consuming a message from a topic in the message broker:

Figure 9.50

```
# LPOP command (= left pop) pops a message from
# the beginning of the list identified by key *topic*
# The LPOP command removes the value from the list
message = redis_client.lpop(topic)
```

9.4: Wide-Column Databases

> ***Use a wide-column database when you have a large amount of data and know what queries you need to execute, and you want these queries to be fast.***

Table structures of a wide-column database are optimized for specific queries. With a wide-column database, storing duplicate data is okay to make the queries faster. Wide-column

databases also scale horizontally well, making them suitable for storing a large amount of data.

This section uses Apache Cassandra as an example wide-column database. Cassandra is a scalable multi-node database engine. In Cassandra, the data of a table is divided into partitions according to the table's partition key. A partition key is composed of one or more columns of the table. Each partition is stored on a single Cassandra node. You can think that Cassandra is a key-value store where the key is the partition key, and the value is another "nested" table. The rows in the "nested" table are uniquely identified by clustering columns sorted by default in ascending order. The sort order can be changed to descending if wanted.

The partition key and the clustering columns form the table's primary key. The primary key uniquely identifies a row. The order of these components always puts the partition key first and then the clustering columns (or clustering key). Let's have an example table that is used to store hotels near a particular point of interest (POI):

Figure 9.51

```
CREATE TABLE hotels_by_poi (
  poi_name text,
  hotel_distance_in_meters_from_poi int,
  hotel_id uuid,
  hotel_name text,
  hotel_address text,
  PRIMARY KEY (poi_name, hotel_distance_in_meters_from_poi,
hotel_id)
);
```

In the above example, the primary key consists of three columns. The first column (`poi_name`) is always the partition key. The partition key must be given in a query. Otherwise, the query will be slow because Cassandra must perform a full table scan because it does not know which node data is located. When the partition key is given in a SELECT statement's WHERE clause, Cassandra can find the appropriate node where the data for that particular partition resides. The two other primary key columns, `hotel_distance_in_meters_from_poi` and `hotel_id`, are the clustering columns. They define the order and uniqueness of the rows in the "nested" table.

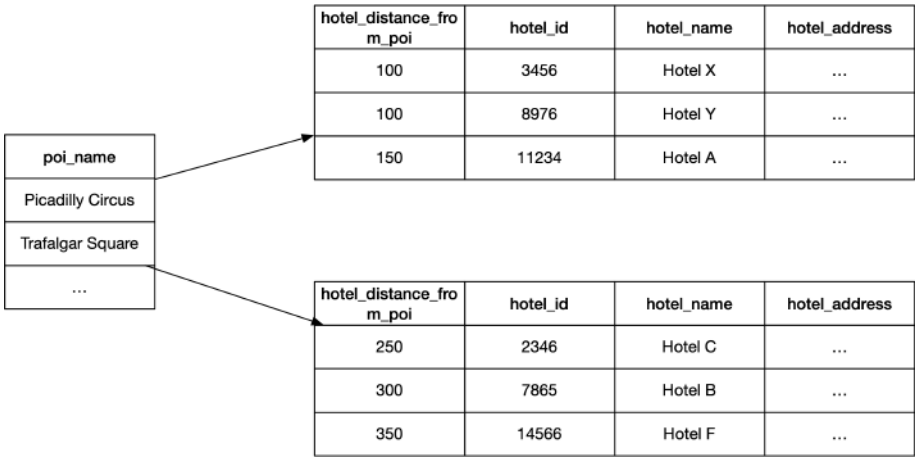

Figure 9.52. hotels_by_poi Table

The above figure shows that when you give a partition key value (`poi_name`), you have access to the respective "nested" table where rows are ordered first by the `hotel_distance_in_meters_from_poi` (ascending) and second by the `hotel_id` (ascending).

Now, it is easy for a hotel room booking client to ask the server to execute a query to find hotels near a POI given by a user. The following query will return the first 15 hotels nearest to *Piccadilly Circus* POI:

Figure 9.53

```
SELECT
  hotel_distance_in_meters_from_poi,
  hotel_id,
  hotel_name,
  hotel_address
FROM hotels_by_poi
WHERE poi_name = 'Piccadilly Circus'
LIMIT 15
```

When a user selects a particular hotel from the result of the above query, the client can request the execution of another query to fetch information about the selected hotel. The user wants to see other POIs near the selected hotel. For that query, we should create another table:

Figure 9.54

```
CREATE TABLE pois_by_hotel_id (
  hotel_id uuid,
  poi_distance_in_meters_from_hotel int,
  poi_id uuid,
  poi_name text,
```

```
  poi_address text,
  PRIMARY KEY (hotel_id, poi_distance_in_meters_from_hotel,
poi_id)
);
```

Now, a client can request the server to execute a query to fetch the nearest 20 POIs for a selected hotel. (hotel with id c5a49cb0-8d98-47e3-8767-c30bc075e529):

Figure 9.55

```
SELECT
  poi_distance_in_meters_from_hotel,
  poi_id,
  poi_name,
  poi_address
FROM pois_by_hotel_id
WHERE hotel_id = c5a49cb0-8d98-47e3-8767-c30bc075e529
LIMIT 20
```

In a real-life scenario, a user wants to search for hotels near a particular POI for a selected period. The server should respond with the nearest hotels having free rooms for the selected period. For that kind of query, we can create an additional table for storing hotel room availability:

Figure 9.56

```
CREATE TABLE availability_by_hotel_id (
  hotel_id uuid,
  accommodation_date date,
  available_room_count counter,
  PRIMARY KEY (hotel_id, accommodation_date)
);
```

The above table is updated whenever a room for a specific day is booked, or a booking for a room is canceled. The `available_room_count` column value is either decremented or incremented by one in the update procedure.

Let’s say that the following query has been executed:

Figure 9.57

```
SELECT
  hotel_distance_in_meters_from_poi,
  hotel_id,
  hotel_name,
  hotel_address
FROM hotels_by_poi
WHERE poi_name = 'Piccadilly Circus'
LIMIT 30
```

Next, we should find hotels from the result of 30 hotels that have available rooms between the 1st of September 2023 and 3rd of September 2023. We cannot use joins in Cassandra, but we can execute the following query where we specifically list the hotel ids returned by the above query:

Figure 9.58

```
SELECT hotel_id, MIN(available_room_count)
FROM availability_by_hotel_id
WHERE hotel_id IN (List the 30 hotel_ids here...) AND
      accommodation_date >= '2022-09-01' AND
      accommodation_date <= '2022-09-03'
GROUP BY hotel_id
LIMIT 15
```

As a result of the above query, we have a list of a maximum of 15 hotels for which the minimum available room count is listed. We can return a list of those maximum 15 hotels where the minimum available room count is one or more to the user.

If Cassandra's query language supported the `HAVING` clause, which it does not currently support, we could have issued the following query to get what we wanted:

Figure 9.59

```
SELECT hotel_id, MIN(available_room_count)
FROM availability_by_hotel_id
WHERE hotel_id IN (List the 30 hotel_ids here...) AND
      accommodation_date >= '2022-09-01' AND
      accommodation_date <= '2022-09-03'
GROUP BY hotel_id
HAVING MIN(available_room_count) >= 1
LIMIT 15
```

A wide-column database is also useful for storing time-series data, e.g., from IoT devices and sensors. Below is a table definition for storing measurement data in a telecom network analytics system:

Figure 9.60

```
CREATE TABLE measurements (
  measure_name text,
  dimension_name text,
  aggregation_period text,
  measure_timestamp timestamp,
  measure_value double,
  dimension_value text,
  PRIMARY KEY ((measure_name, dimension_name,
aggregation_period),
               measure_timestamp,
               measure_value,
```

```
                dimension_value)
) WITH CLUSTERING ORDER BY (
  measure_timestamp DESC,
  measure_value DESC
  dimension_value ASC
);
```

In the above table, we have defined a *compound partition key* containing three columns: `measure_name`, `dimension_name`, and `aggregation_period`. Columns for a compound partition key are given in parentheses because the first column of the primary key is always the partition key.

Suppose we have implemented a client that visualizes measurements. In the client, a user can first choose what counter/KPI (= measure name) to visualize, then select a dimension and aggregation period. Let's say that the user wants to see *dropped_call_percentage* for *cells* calculated for one minute at 2023-02-03 16:00. The following kind of query can be executed:

Figure 9.61

```
SELECT measure_value, dimension_value
FROM measurements
WHERE measure_name = 'dropped_call_percentage' AND
      dimension_name = 'cell' AND
      aggregation_period = '1min' AND
      measureTimestamp = '2022-02-03T16:00+0000'
LIMIT 50;
```

The above query returns the top 50 cells with the highest dropped call percentage for the given minute.

We can create another table to hold measurements for a selected dimension value, e.g., for a particular cell id. This table can be used to drill down to a particular dimension and see measure values in the history.

Figure 9.62

```
CREATE TABLE measurements_by_dimension (
  measure_name text,
  dimension_name text,
  aggregation_period text,
  dimension_value text,
  measure_timestamp timestamp,
  measure_value double,
  PRIMARY KEY ((measure_name,
                dimension_name,
                aggregation_period,
```

```
                dimension_value), measure_timestamp)
) WITH CLUSTERING ORDER BY (measureTimestamp DESC);
```

The below query will return dropped call percentage values for the last 30 minutes for the cell identified by *cell id* 3000:

Figure 9.63

```
SELECT measure_value, measureTimestamp
FROM measurements_by_dimension
WHERE measure_name = 'dropped_call_percentage' AND
      dimension_name = 'cell' AND
      aggregation_period = '1min' AND
      dimension_value = '3000'
LIMIT 30;
```

9.5: Search Engines

> ***Use a search engine if you have free-form text data that users should be able to query.***

A search engine (like Elasticsearch) is useful for storing information like log entries collected from microservices. You typically want to search the collected log data by the text in the log messages.

It is not necessary to use a search engine when you need to search for text data. Other databases, both document and relational, have a special index type that can index free-form text data in a column. Considering the earlier example with MongoDB, we might want a client to be able to search sales items by the text in the sales item's name. We don't need to store sales items in a search engine database. We can continue storing them in a document database (MongoDB) and introduce a *text* type index for the `name` field. That index can be created with the following MongoDB command:

Figure 9.64

```
sales_items.create_index( { 'name': 'text' } )
```

10: Concurrent Programming Principles

This chapter presents the following concurrent programming principles:

- Threading principle
- Thread safety principle
- Publish/subscribe shared state change principle

10.1: Threading Principle

> ***Modern cloud-native microservices should primarily scale out by adding more processes, not scale up by adding more threads. Use threading only when it is needed or is a good optimization.***

When developing modern cloud-native software, microservices should be stateless and automatically scale horizontally (scaling out and in via adding and removing processes). The role of threading in modern cloud-native microservices is not as prominent as earlier when software consisted of monoliths running on bare metal servers, mainly capable of scaling up or down. Nowadays, you should use threading if it is a good optimization or otherwise needed. Apart from microservices, if you have a library, standalone application, or a client software component, the situation is different, and you can use threading.

Suppose we have a software system with an event-driven architecture. Multiple microservices communicate with each other using asynchronous messaging. Each microservice instance has only a single thread that consumes messages from a message broker and processes them. If the message broker's message queue for a microservice starts growing too long, the

microservice should scale out by adding a new instance. When the load for the microservice diminishes, it can scale in by removing an instance. There is no need to use threading at all.

We could use threading in the data exporter microservice if the input consumer and the output producer were synchronous. The reason for using threads is optimization. If we had everything in a single thread and the microservice was performing network I/O (either input or output-related), the microservice would have nothing to execute because it is waiting for some network I/O to complete. Using threads, we can optimize the execution of the microservice so that it potentially has something to do when waiting for an I/O operation to complete.

Many modern input consumers and output producers are available as asynchronous implementations. If we use an asynchronous consumer and producer in the data exporter microservice, we can eliminate threading because network I/O will not block the execution of the main thread anymore. As a rule of thumb, consider using asynchronous code first, and if it is not possible or feasible, only then consider threading. Many Python libraries exist that support asynchronous operations instead of synchronous ones. Good examples are FastAPI, databases, and aiofiles.

> Python uses the Linux-native asynchronous I/O facility (libaio) or POSIX asynchronous I/O (AIO) interface that allows applications to initiate I/O operations that are performed asynchronously (i.e., in the background). The application can choose how it is notified of the completion of the I/O operation: by delivery of a signal, by instantiation of a thread, or no notification at all.

You might need a microservice to execute housekeeping tasks on a specific schedule in the background. Instead of using threading and implementing the housekeeping functionality in the microservice, consider implementing it in a separate microservice to ensure that the *single responsibility principle* is followed. For example, you can configure the housekeeping microservice to run regularly using a Kubernetes CronJob.

Threading also brings complexity to a microservice because the microservice must ensure thread safety. You will be in big trouble if you forget to implement thread safety. Threading and synchronization-related bugs are hard to find. Thread safety is a topic that is discussed later in this chapter. Threading also brings complexity when deploying a microservice because the number of vCPUs requested by the microservice depends on the number of threads used.

10.1.1: Parallel Executors

Parallel executors simplify concurrency by hiding the creation of multiple sub-processes. In Python, you can create parallel executors by using multiple processes. Below is an example using a pool of 4 sub-processes:

Figure 10.1

```
import multiprocessing
import os


def print_stdout(number: int) -> None:
    print (f'{number} {os.getpid()}')


if __name__ == '__main__':
    numbers = [1, 2, 3, 4]
    pool = multiprocessing.Pool(4)
    pool.map(print_stdout, numbers)
```

The output of the above code could be, for example:

Figure 10.2

```
1 97672
2 97671
4 97672
3 97670
```

10.2: Thread Safety Principle

> ***If you are using threads, you must ensure thread safety. Thread safety means that only one thread can access shared data simultaneously to avoid race conditions.***

Do not assume thread safety if you use a ready-made data structure or library. You must consult the documentation to see whether thread safety is guaranteed. If thread safety is not mentioned in the documentation, it can't be assumed. The best way to communicate thread safety to developers is to name things so that thread safety is explicit. For example, you could create a thread-safe collection library and have a class named `ThreadSafeList` to indicate the class is thread-safe.

The primary way in Python to ensure thread safety is to use a lock. Python does not have atomic variables.

10.2.1: Use Locking for Mutual Exclusion (Mutex)

Python has a `Lock` class in the `threading` module. The class implements primitive lock objects to achieve mutual exclusion. Once a thread has acquired a lock, subsequent attempts to acquire it block until it is released. Any thread can release it.

Let's implement a thread-safe counter using a lock object:

Figure 10.3. ThreadSafeCounter.py

```
from threading import Lock

class ThreadSafeCounter:
    def __init__(self):
        self.__lock = Lock()
        self.__counter = 0

    def increment(self) -> None:
        with self.__lock:
            self.__counter += 1

    @property
    def value(self) -> int:
       with self.__lock:
           return self.__counter
```

Python also contains a `Lock` class in the `multiprocessing` module, and it can be used to synchronize multiple sub-processes in a similar fashion.

10.2.2: Atomic Variables

Python does not have atomic variables, but you can define your own atomic variable class using locking. Below is an example of an `AtomicInt` class that uses a lock.

Figure 10.4

```
from threading import Lock

class AtomicInt():
    def __init__(self, value: int):
        self.__value = value
        self.__lock = Lock()

    def increment(self, amount: int) -> int:
        with self.__lock:
            self.__value += amount
            return self.__value

    def decrement(self, amount: int) -> int:
        with self.__lock:
            self.__value -= amount
            return self.__value

    @property
    def value(self) -> int:
        with self.__lock:
            return self.__value

    @value.setter
    def value(self, new_value: int):
        with self.__lock:
            self.__value = new_value
```

All the operations on `my_int` below can be done safely from different threads:

Figure 10.5

```
my_int = AtomicInt(0)

my_int.increment(1)
my_int.decrement(2)
print(my_int.value) # Prints -1
```

10.2.3: Concurrent Collections

Multiple threads can use concurrent collections without any additional synchronization. Below is a partial example of a thread-safe list:

Figure 10.6

```
from threading import Lock
from typing import Generic, TypeVar

T = TypeVar('T')
```

```
class ThreadSafeList(Generic[T]):
    def __init__(self):
        self.__list: list[T] = []
        self.__lock = Lock()

    def append(self, value: T) -> None:
         with self.__lock:
            self.__list.append(value)

    def pop(self, index: int) -> T:
        with self.__lock:
            return self.__list.pop(index)

    def get(self, index: int) -> T:
        with self.__lock:
            return self.__list[index]

    # Implement rest of wanted methods
```

10.3: Publish/Subscribe Shared State Change Principle

Use condition objects to publish and wait for a change to a shared state.

Condition objects are useful when you have a queue, and there is a producer and consumer for the queue in different threads. The producer thread can inform the consumer thread when a new item is placed in the queue, and the consumer thread waits for an item to be available in the queue. If you did not have a condition object, you would have to implement this using a lock and a sleep in the consumer. This is not optimal because you don't know the optimal sleep duration. You can sleep too long or too short. Below is an example of using a condition object.

Figure 10.7

```
from threading import Condition, Lock
from typing import Final, Generic, TypeVar

T = TypeVar('T')


class ThreadSafeQueue(Generic[T]):
    def __init__(self):
        self.__items: Final[list[T]] = []
        self.__item_waiter: Final = Condition()
        self.__lock: Final = Lock()
```

```
    def append(self, item: T) -> None:
        with self.__lock:
            self.__items.append(item)

    def pop_front(self) -> T:
        with self.__lock:
            return self.__items.pop(0)

    def has_item(self) -> bool:
        return len(self.__items) > 0

    @property
    def item_waiter(self):
        return self.__item_waiter


class MsgQueueProducer(Generic[T]):
    def __init__(self, queue: ThreadSafeQueue[T]):
        self.__queue = queue

    def produce(self, item: T) -> None:
        with self.__queue.item_waiter:
            self.__queue.append(item)
            self.__queue.item_waiter.notify()


class MsgQueueConsumer(Generic[T]):
    def __init__(self, queue: ThreadSafeQueue[T]):
        self.__queue = queue

    def consume(self) -> T:
        with self.__queue.item_waiter:

self.__queue.item_waiter.wait_for(self.__queue.has_item)
            return self.__queue.pop_front()
```

11: Teamwork Principles

This chapter presents teamwork principles. The following principles are described:

- Use an agile framework principle
- Define the done principle
- You write code for other people principle
- Avoid technical debt principle
- Software component documentation principle
- Code review principle
- Uniform code formatting principle
- Highly concurrent development principle
- Pair programming principle
- Well-defined development team roles principle
- Competence transfer principle

11.1: Use Agile Framework Principle

> ***Using an agile framework can bring numerous benefits to an organization, including an increase in productivity, improvements in quality, faster time-to-market, and better employee satisfaction.***

The above statements come from SAFe customer stories of some companies having adopted Scaled Agile Framework (SAFe).

An agile framework describes a standardized way of developing software, which is essential, especially in large organizations. In today's work environments, people change jobs frequently, and teams tend to change often, which can lead to a situation where there is no common understanding of

the way of working unless a particular agile framework is used. An agile framework establishes a clear division of responsibilities, and everyone can focus on what they do best.

In the *SAFe*, for example, during a program increment (PI) planning, development teams plan features for the next PI (consisting of 4 iterations, two weeks per iteration, a total of 8 weeks followed by an IP iteration). In the PI planning, teams split features into user stories and see which features fit the PI. Planned user stories will be assigned story points and stories will be placed into iterations.

Story points can be measured in concrete units, like person days. Use a slightly modified Fibonacci sequence, like 1, 2, 3, 5, 8, 13, 20, 40, to estimate the size of a user story. The benefit of using the Fibonacci sequence is that it takes into account the fact that the effort estimation accuracy decreases when the needed work amount increases. Story points can also be measured in abstract units of work. Then, you compare a user story to a so-called *golden user story* (a medium-sized user story known to all team members) and assign the effort estimate for the user story based on how much smaller or larger it is compared to the golden user story. When using abstract story points, you can also use a Fibonacci sequence, e.g., 1 (XS), 2 (S), 3 (M), 5 (L), 8 (XL). Let's say you have a medium-sized golden user story with three story points, and you need to estimate work effort for a new user story known to be bigger than the golden user story. Then, you assign either 5 or 8 story points to the new user story, depending on how much bigger it is compared to the golden user story. Similarly, if a new user story is known to be smaller than the golden user story, you assign either 2 or 1 story points to it. If the new user story is roughly the same amount of work as the golden user story, assign three story points to the user story.

There are several ways to estimate story points for user stories:

- If a user story has a single assignee, they can estimate the effort. The assignee might be the person who knows most about the particular user story.
- Team can decide together, e.g., using planning poker
- Lead developer can provide initial estimates, which are gone through with the team

When using concrete story points (person days), the team velocity for an iteration is also calculated in person days. This makes it easy to adjust the iteration velocity based on estimated ad-hoc and maintenance work, as well as public holidays and vacations. If the team uses abstract story points, the team velocity is inferred from past iterations. This method does not allow straightforward adjustments to the velocity due to team size changes, sudden changes in the amount of ad-hoc/maintenance work, and leaves.

This planning phase results in a plan the team should follow in the PI. Junior SAFe practitioners can make mistakes like underestimating the work needed to complete a user story. But this is a self-correcting issue. When teams and individuals develop, they will better estimate the needed work amount, making plans more solid. Teams and developers learn that they must make all work visible. For example, reserve time to learn new things, like a programming language or framework, and reserve time for refactoring. Being able to keep the planned schedule and sometimes even completing work early is very satisfying. This will make you feel like a true professional and boost your self-esteem.

Teams and individuals estimate their work. There is no management involved. Managers don't tell you how much time you have for something. They don't come asking when something is ready or pressure you to complete tasks earlier than estimated. Also, they don't come to you with extra work to do. All of this will make your work feel less stressful.

My personal experience with SAFe for over five years is mainly positive. I feel I can concentrate more on "the real work", which makes me happier. There are fewer meetings, fewer irrelevant emails, and fewer interruptions in the development flow in general. This is mainly because the team has a *product owner* and *scrum master* whose role is to protect the team members from any "waste" or "the management stuff" and allow them to concentrate on their work.

If a team has work that does not require effort estimation, Kanban can be used instead of Scrum. For example, in my organization, the DevOps team uses Kanban, and all development teams use Scrum. A Scrum development team can commit to delivering a feature on a certain schedule, which is not possible when using Kanban. Many development teams use Scrum to enable making commitments to the business that can make commitments to customers.

11.2: Define the Done Principle

For user stories and features, define what "done" means.

In the most optimal situation, development teams have a shared understanding of what is needed to declare a *user story* or *feature* done. Consistent results and quality from each development team can be ensured when a common definition of done exists.

When considering a user story, at least the following requirements for a done user story can be defined:

- Source code is committed to a source code repository
- Source code is reviewed
- Static code analysis is performed (No blocker/critical/major issues)
- Unit test coverage is at least X%
- CI/CD pipeline is passing
- No 3rd party software vulnerabilities
- Technical documentation is updated if needed

The product owner's (PO) role in a team is to accept user stories as done. Some of the above-mentioned requirements can be automatically checked. For example, the static code analysis should be part of every CI/CD pipeline and can also check the unit test coverage automatically. If static code analysis does not pass or the unit test coverage is unacceptable, the CI/CD pipeline should not pass.

Some additional requirements for done-ness should be defined when considering a feature because features can be delivered to customers. Below is a non-exhaustive list of some requirements for a done feature:

- Architectural design documentation is updated
- Integration tests are added/updated
- End-to-end tests are added/updated if needed

- Non-functional testing is done
- User documentation is updated and reviewed
- Threat modeling is done, and threat countermeasures (security features) are implemented
- Observability is updated if needed

To complete all the needed done-ness requirements, development teams can use tooling that helps them remember what needs to be done. For example, when creating a new user story in a tool like Jira, an existing prototype story could be cloned (or a template used). The prototype or template story should contain tasks that must be completed before a user story can be approved.

11.3: You Write Code for Other People Principle

You write code for other people and your future self.

Situations where you work alone with a piece of software are relatively rare. You cannot predict what will happen in the future. There might be someone else responsible for the code you once wrote. There are cases when you work with some code for some time and then, after several years, need to return to that code. For these reasons, writing clean code that is easy to read and understand by others and yourself in the future is essential. Remember that code is not written for a computer only but also for people. People should be able to read and comprehend code easily. Remember that code is read more often than written. At best, the code reads like beautiful prose!

11.4: Avoid Technical Debt Principle

> Technical debt is the implied cost of future rework/refactoring required when choosing an easy but limited solution instead of a better approach that could take more time.

The most common practices for avoiding technical debt are the following:

- The architecture team should design the high-level architecture (Each team should have a representative in the architecture team. Usually, it is the technical lead of the team)
- Development teams should perform domain-driven design (DDD) and object-oriented design (OOD) first, and only after that proceed with implementation
- Conduct DDD and OOD within the team with relevant senior and junior developers involved
- Don’t take the newest 3rd party software immediately into use. Instead, use mature 3rd party software that has an established position in the market
- Design for easily replacing a 3rd party software component with another 3rd party component.
- Design for scalability (for future load)
- Design for extension: new functionality is placed in new classes instead of modifying existing classes (open-closed principle)
- Utilize a plugin architecture (possibility to create plugins to add new functionality later)
- Reserve time for refactoring
- Use test-driven development (TDD)
- Use behavioral-driven development (BDD)

The top reasons for technical debt are the following (not in any particular order):

- Using niche technologies or brand-new technologies that are immature
- Not making software scalable for future processing needs
- When it is not relatively easy to replace a 3rd party software component (E.g., using custom SQL syntax does

not allow changing the database, not using the *adapter pattern* with 3rd party libraries)

- Not reviewing the architecture
- Not doing any domain-driven design and object-oriented design before starting coding
- Not engaging senior enough developers in the DDD and OOD phase
- Not understanding and using relevant design principles and patterns
    - Not programming against interfaces
    - Not easy to change a dependency (DI missing)
    - No facades
- Not reviewing code changes
- Not reserving time for refactoring
- Too small work effort estimates
- Time pressure from management
- Management does not understand the value of refactoring
- Postponing refactoring to a time point that never comes
- Forgetting to refactor (at least store the needed refactoring work items in a TODO.MD file in the source code repository)
- No unit tests, harder to refactor
- Not practicing TDD
- Duplicate code
- Not conducting the boy scout rule
- Laziness (using what comes first in mind or constantly trying to find the easiest and quickest way to do things)

11.5: Software Component Documentation Principle

> ***Each software component needs to be documented. The main idea behind documenting is quickly onboarding new people to development work.***

It is crucial that setting up a development environment for a software component is well-documented and as easy as possible. Another important thing is to let people easily understand the problem domain the software component tries to solve. Also, the object-oriented design of the software component should be documented.

Software component documentation should reside in the same source code repository where the source code is. The recommended way is to use a README.MD file in the root directory of the source code repository for documentation in Markdown format. You should split the documentation into multiple files and store additional files in the *docs* directory of the source code repository. This way, it is less likely to face a merge conflict if multiple persons edit the documentation simultaneously.

JetBrains offers a new tool called Writerside to author software documentation in Markdown format. This tool will generate a table of contents for you automatically. It allows you to produce diagrams using declarative code (with Mermaid.js) instead of drawing them by hand. It offers tools to generate a documentation website for GitHub Pages or GitLab Pages as part of the software component's CI/CD pipeline.

Below is an example table of contents that can be used when documenting a software component:

- Short description of the software component and its purpose
- Feature list
    - You can provide a link to Gherkin feature files here, and then you don't have to store the same information in two places.
- Architecture (how the software component relates to other software components)

- Data flow diagram
- OOD diagrams describing different subdomains and interfaces/classes in each subdomain with their relationships and main public methods.
    - Explanation of the design (if needed)
    - Don’t create or generate a single class diagram for the whole microservice, but create or generate multiple class diagrams for various subdomains
- API documentation (for libraries, this should be auto-generated from the source code)
- Implementation-related documentation
    - Error handling mechanism
    - Special algorithms used
    - Performance considerations
    - Major security features
- Instructions for setting up a development environment
    - The easiest way to set up a development environment is to use a development container, a concept supported by the Visual Studio Code editor. The benefit of using a development container is that you don’t have to install development tools locally, and there is no risk of using the wrong versions of the development tools
- Instructions for building the software locally
- Instructions for running unit tests locally
- Instructions for running integration tests locally
- Instructions for deploying to a test environment
- Configuration
    - Environment variables
    - Configuration files
    - Secrets

- Observability
  - Logging (levels, log format, error codes including their meaning and possible resolution instructions)
  - Metrics/SLIs
  - SLOs
  - Alarms

11.6: Code Review Principle

> ***In a code review, focus on issues a machine cannot find for you.***

Before reviewing code, a static code analysis should be performed to find any issues a machine can find. The actual code review should focus on issues that static code analyzers cannot find. You should not need to review code formatting because every team member should use the same code format, which should be ensured by an automatic formatting tool. You cannot review your own code. At least one of the reviewers should be in a senior or lead role. Things to focus on in a code review are presented in the subsequent sections.

An essential part of code review is to ensure that code is readable and understandable because code is read more often than written. You can write code once (assuming perfect code), which can be read by tens or even hundreds of developers during tens of years. When reviewing (reading) code, every misunderstanding, doubt, and WTF? moment reveals that there is room for improvement in the code readability. In a code review, finding bugs can be a secondary target because bugs can be found by a machine (static code analyzers) and with an extensive set of automated tests that should always exist.

11.6.1: Focus on Object-Oriented Design

Before starting coding, it is recommended to design the software: define subdomains, needed interfaces, and classes. The product of this initial design phase should be committed

to the source code repository and reviewed before starting coding. This way, correcting design flaws early and avoiding technical debt is easier. Fixing design flaws in a later phase might require significant effort or even a rewrite of the existing code. At least one senior developer should participate in the design.

If a design flaw or flaws are encountered in a review, and there is no time for an immediate fix, a refactoring user story or stories should be added to the team's backlog.

11.6.2: Focus on Function Specification by Unit Tests

To review each public function, the reviewer should start with the unit tests and see if they cover all the functionality. Is there a unit test missing for an error scenario, a security scenario, or an edge/corner case?

11.6.3: Focus on Proper and Uniform Naming

One thing that static code analysis tools can only partially do is ensure proper and uniform naming of things, like classes, functions, and variables. Correct naming is where the focus should be put in a code review. Renaming things is a very straightforward and fast refactoring task that can be performed automatically by modern IDEs.

11.6.4: Don't Focus on Premature Optimization

Do not focus on optimization in regular code reviews. Optimization is usually performed on a need basis after the code is ready and the performance is first measured. Focus on optimization-related issues only when the commit you are reviewing is dedicated to optimizing something.

11.6.5: Detect Possible Malicious Code

Code reviewers must ensure that the committed code does not contain malicious code.

11.7: Uniform Code Formatting Principle

> ***In a software development team, you must decide on common rules for formatting source code.***

Consistent source code formatting is vital because if team members have different source code formatting rules, one team member's small change to a file can reformat the whole file using their formatting rules, which can cause another developer to face a major merge conflict that slows down the development process. Always agree on common source code formatting rules and preferably use a tool like *Black* or *Blue* to enforce the formatting rules.

11.8: Highly Concurrent Development Principle

> ***Each team member can work with some piece of code. No one should be waiting long for someone else's work to finish.***

Concurrent development is enabled when different people modify different source code files. When several people need to alter the same files, it can cause merge conflicts. These merge conflicts cause extra work because they often must be resolved manually. This manual work can be slow, and it is error-prone. The best thing is to avoid merge conflicts as much as possible. This can be achieved in the ways described in the following sections.

11.8.1: Dedicated Microservices and Microlibraries

Microservices are small by nature, and it is possible to assign the responsibility of a microservice to a single team member. This team member can proceed with the microservice with full

velocity, and rest assured that no one else is modifying the codebase. The same goes for libraries. You should create small microlibraries (= libraries with a single responsibility) and assign the responsibility of developing a microlibrary to a single person.

11.8.2: Dedicated Domains

Sometimes, assigning a single microservice or library to a single developer is impossible. It could be because the microservice or library is relatively large and it is not feasible to split it into multiple microservices or libraries. The microservice or library should be divided into several subdomains in those cases. Each source code directory reflects a different subdomain. It is then possible to assign the responsibility of a single subdomain to a single person. The assignment of subdomains need not be fixed but can and should change as time passes. To distribute the knowledge of different domains in the team, it is advisable to rotate the responsibilities amongst the team members. Let's say you have a team of three developers developing a data exporter microservice consisting of input, transform, and output subdomains. The team can implement the microservice by assigning the responsibility of a single domain to a single developer. Then, all developers can proceed highly independently and concurrently with the implementation. In the early phase, they must collaborate to define interfacing between the different subdomains.

In the future, when new features are developed, team members can take responsibility for other domains to spread knowledge about the microservice in the team.

11.8.3: Follow Open-Closed Principle

Sometimes, you might face a situation where a single subdomain is so large that you need multiple developers. This, of course, should be a relatively rare case. When several developers modify source code files belonging to the same subdomain (i.e., in the same directory), merge conflicts may

arise. This is the case, especially when existing source code files are modified. But when developers follow the *single responsibility principle* and *open-closed principle*, they should not change existing classes (source code files) but rather implement new functionality in new classes (source code files). Using the *open-closed principle* enables developers to develop more concurrently because they primarily work with different source code files, making merge conflicts rare or at least less frequent.

11.9: Pair Programming Principle

> ***Pair programming helps produce better quality software with better design, less technical debt, better tests, and fewer bugs.***

Pair programming is something some developers like and other developers hate. It is not a one-fits-all solution. It is not take it or leave it, either. You can have a team where some developers program in pairs and others don't. Also, people's opinions about pair programming can be prejudiced. Perhaps they have never done pair programming, so how do they know if they like it or not? It is also true that choosing the right partner to pair with can mean a lot. Some pairs have better chemistry than other pairs.

Does pair programming just increase development costs? What benefits does pair programming bring?

I see pair programming as valuable, especially in situations where a junior developer pairs with a more senior developer, and in this way, the junior developer is onboarded much faster. He can "learn from the best". Pair programming can improve software design because there is always at least two persons' view of the design. Bugs can be found more easily and usually in an earlier phase (four eyes compared to two eyes only). So, even if pair programming can add some cost, it usually results in software with better quality: better design, less technical debt, better tests, and fewer bugs.

11.10: Well-Defined Development Team Roles Principle

> ***A software development team should have a well-defined role for each team member.***

A software development team does not function optimally if everyone is doing everything or if it is expected that anyone can do anything. No one is a jack of all trades. A team achieves the best results with specialists targeted for different types of tasks. Team members need to have focus areas they like to work with and where they can excel. When you are a specialist in some area, you can complete tasks belonging to that area faster and with better quality.

Below is a list of needed roles for a development team:

- Product owner (PO)
- Scrum master (SM)
- Software developer (junior/senior/lead)
- Test automation developer
- DevOps engineer
- UI designer (if the team develops UI software components)

Let's discuss each role's responsibilities in detail in the following sections.

11.10.1: Product Owner

The product owner (PO) acts as an interface between the development team and the business, which usually means product management (PM). The PO gathers requirements (non-functional and functional) from PM via discussions. The PO is responsible for prioritizing the team backlog according to the guidance from the PM and defining user stories with the team. The PO role is not usually full-time so that a single PO

can serve two small teams. When the PO gathers requirements (non-functional and functional) from the PM, the PM does not usually list all the requirements, but the PO must determine all the requirements by asking the right questions and discussing them with the PM. The PO should be a technical person and should be able to create, e.g., Gherkin feature file(s) as the acceptance criteria for a backlog feature.

11.10.2: Scrum Master

A scrum master (SM) is a servant leader and a coach for the development team. The scrum master ensures that relevant agile practices and the agile process are followed. They educate the team in agile practices. If the team has a line manager, the line manager can serve as the scrum master, but any team member can be the scrum master. Being a scrum master is a part-time job. The role of the scrum master can also be rotated between the team members.

11.10.3: Software Developer

A software developer is responsible for designing, implementing, and testing software (including unit and, in many cases, integration testing). A software developer is usually focused on one or two programming languages and a couple of technical frameworks. Typically, software developers are divided into the following categories:

- Backend developers
- Frontend developers
- Full-stack developers
- Mobile developers
- Embedded developers

A backend developer develops microservices, like APIs, running in the backend. A frontend developer develops web clients. Typically, a frontend developer uses JavaScript or TypeScript, React/Angular/Vue, HTML, and CSS. A full-stack

developer is a combination of a backend and frontend developer capable of developing backend microservices and frontend clients. A mobile developer develops software for mobile devices, like phones and tablets.

A team should have software developers at various seniority levels. Each team should have a lead developer with the best experience in the used technologies and the domain. The lead developer typically belongs to the virtual architecture team led by the system architect. There is no point in having a team with just junior or senior developers. The idea is to transfer skills and knowledge from senior to junior developers. This also works the other way around. Junior developers can have knowledge of some new technologies and practices that senior developers lack. So overall, the best team consists of a good mix of junior, medior, and senior developers.

11.10.4: Test Automation Developer

A test automation developer is responsible for developing different kinds of automated tests. Typically, test automation developers develop integration, E2E, and automated non-functional tests. A test automation developer must be proficient in at least one programming language, like Python, used to develop automated tests. Test automation developers must have a good command of BDD and ATDD and some common testing frameworks, like Behave or Robot Framework. Knowledge of some non-functional testing tools, like Apache JMeter, is appreciated. Test automation developers can also develop internal testing tools, like interface simulators and data generators. Test automation developers should form a virtual team to facilitate the development of E2E and automated non-functional tests.

11.10.5: DevOps Engineer

A DevOps engineer acts as an interface between the software development team and the software operations. A DevOps engineer usually creates CI/CD pipelines for microservices and crafts infrastructure and deployment-related code. DevOps

engineers also define alerting rules and metrics visualization dashboards that can be used when monitoring the software in production. DevOps engineers help operations personnel monitor software in production. They can help troubleshoot problems that the technical support organization cannot solve. DevOps engineer knows the environment (=infrastructure and platform) where the software is deployed, meaning that basic knowledge of at least one cloud provider (AWS/Azure/Google Cloud, etc.) and perhaps Kubernetes is required. DevOps engineers should form a virtual team to facilitate specifying DevOps-related practices and guidelines.

11.10.6: UI Designer

A UI designer is responsible for designing the final UIs based on higher-level UX/UI designs/wireframes. The UI designer will also conduct usability testing of the software.

11.11: Competence Transfer Principle

An extreme case of competence transfer happens when a person leaving a company must hand off the responsibility of a software component to another person. Many times, this other person is not familiar with the software component at all or has only little knowledge about it. To ensure a smooth transfer of competence, at least the following must be performed as part of the competence transfer:

- Demonstration of the software component features
- Explaining the architecture of the software component (how it interacts with other software components in the system)
- Explaining the (object-oriented) design of the software component
    - How the software component is split into subdomains
    - Major interfaces/classes

- Explaining the major implementation decisions for the software component:
  - Special algorithms used
  - Concurrency
  - Error handling mechanism
  - Major security features
  - Performance considerations
- Explaining the configuration of the software component
- Setting up the development environment according to the instructions in the README.MD to ensure that the instructions are correct and up-to-date.
- Building the software, executing unit tests, and executing the integration tests
- Deploying the software component to a test environment
- Explaining the CI/CD pipeline (if it differs from CI/CD pipelines of other software components)
- Explaining other possible automated functional and non-functional tests, like E2E tests, performance and stability tests
- Explaining the observability of the software component, i.e., logging, audit logging, metrics, dashboards, and alerts

12: DevSecOps

DevOps describes practices that integrate software development (Dev) and software operations (Ops). It aims to shorten the software development lifecycle through development parallelization and automation and provides continuous delivery of high-quality software. DevSecOps enhances DevOps by adding security aspects to the software lifecycle.

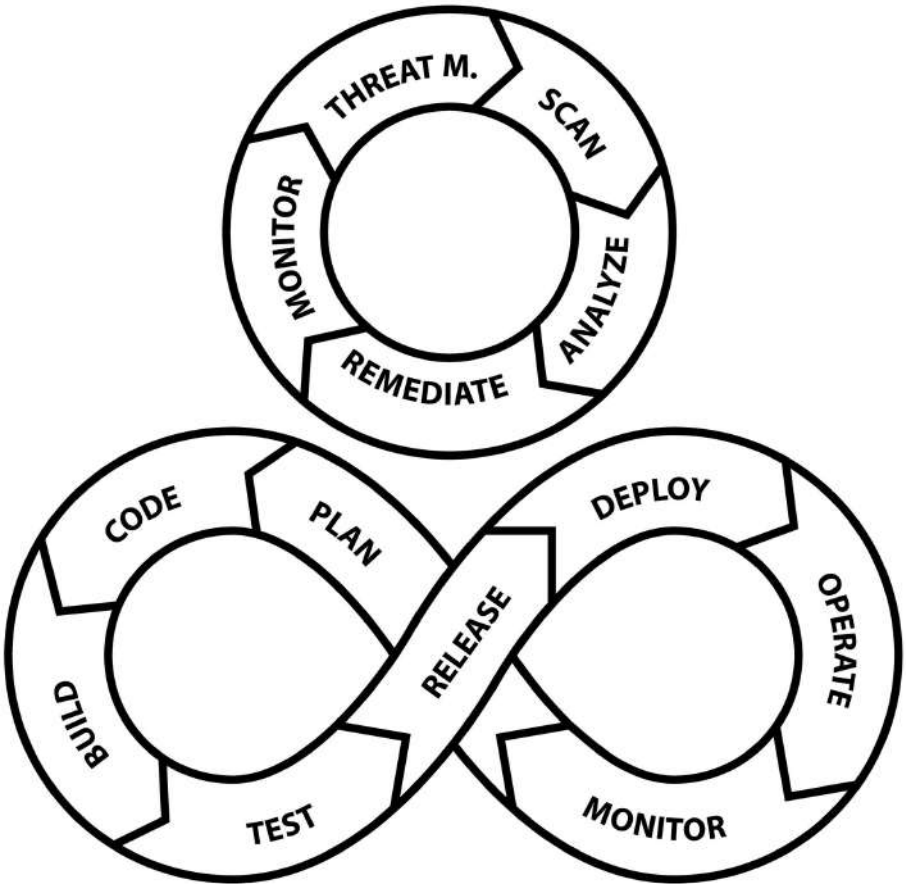

Figure 12.1. DevSecOps Diagram

A software development organization is responsible for planning, designing, and implementing software deliverables. Software operations deploy software to IT infrastructure and platforms. They monitor the deployed software to ensure it runs without problems. Software operations also provide

feedback to the software development organization through support requests, bug reports, and enhancement ideas.

12.1: SecOps Lifecycle

The SecOps lifecycle is divided into the following phases:

- Threat modeling
  - To find out what kind of security features and tests are needed
  - Implementation of threat countermeasures and mitigation. This aspect was covered in the earlier *security principles* chapter
- Scan
  - Static security analysis (also known as SAST = Static Application Security Testing)
  - Security testing (also known as DAST = Dynamic Application Security Testing)
  - Container vulnerability scanning
- Analyze
  - Analyze the results of the scanning phase, detect and remove false positives, and prioritize corrections of vulnerabilities
- Remediate
  - Fix the found vulnerabilities according to prioritization
- Monitor
  - Define SecOps-related metrics and monitor them

12.2: DevOps Lifecycle

The DevOps lifecycle is divided into the following phases:

- Plan

- Code
- Build
- Test
- Release
- Deploy
- Operate
- Monitor

Subsequent sections describe each of the phases in more detail.

12.2.1: Plan

Plan is the first phase in the DevOps lifecycle. In this phase, software features are planned, and high-level architecture and user experience (UX) are designed. This phase involves business (product management) and software development organizations.

12.2.2: Code

Code is the software implementation phase. It consists of designing and implementing software components and writing various automated tests, including unit tests, integration tests, and E2E tests. This phase also includes all other coding needed to make the software deployable. Most of the work is done in this phase, so it should be streamlined as much as possible.

The key to shortening this phase is to parallelize everything to the maximum possible extent. In the *Plan* phase, the software was architecturally split into smaller pieces (microservices) that different teams could develop in parallel. Regarding developing a single microservice, there should also be as much parallelization as possible. This means that if a microservice can be split into multiple subdomains, the development of these subdomains can be done very much in parallel. If we think about the data exporter microservice, we identified

several subdomains: input, decoding, transformations, and output. If you can parallelize the development of these four subdomains instead of developing them one after another, you can significantly shorten the time needed to complete the implementation of the microservice.

To shorten this phase even more, a team should have dedicated test automation developer(s) who can start developing automated tests in an early phase parallel to the implementation.

Providing high-quality software requires high-quality design, implementation with little technical debt, and comprehensive functional and non-functional testing. All of these aspects were handled in the earlier chapters.

12.2.3: Build and Test

The *Build and Test* phase should be automated and run as continuous integration (CI) pipelines. Each software component in a software system should have its own CI pipeline. A CI pipeline is run by a CI tool like Jenkins or GitHub Actions. A CI pipeline is defined using (declarative) code stored in the software component's source code repository. Every time a commit is made to the main branch in the source code repository, it should trigger a CI pipeline run.

The CI pipeline for a software component should perform at least the following tasks:

- Checkout the latest source code from the source code repository
- Build the software
- Perform static code analysis. A tool like SonarQube or SonarCloud can be used
    - Perform static application security testing (SAST).
- Execute unit tests
- Execute integration tests

- Perform dynamic application security testing (DAST). A tool like ZAP can be used
- Verify 3rd party license compliance and provide a software bill of materials (SBOM). A tool like Fossa can be used

12.2.4: Release

In the *Release* phase, built and tested software is released automatically. After a software component's CI pipeline is successfully executed, the software component can be automatically released. This is called continuous delivery (CD). Continuous delivery is often combined with the CI pipeline to create a CI/CD pipeline for a software component. Continuous delivery means that the software component's artifacts are delivered to artifact repositories, like Artifactory, Docker Hub, or a Helm chart repository.

A CD pipeline should perform the following tasks:

- Perform static code analysis for the code that builds a container image (e.g., Dockerfile). A tool like Hadolint can be used for *Dockerfiles*.
- Build a container image for the software component
- Publish the container image to a container registry (e.g., Docker Hub, Artifactory, or a registry provided by your cloud provider)
- Perform a container image vulnerability scan
    - Remember to enable container vulnerability scanning at regular intervals in your container registry, also
- Perform static code analysis for deployment code. Tools like Helm's *lint* command, Kubesec and Checkov can be used
- Package and publish the deployment code (for example, package a Helm chart and publish it to a Helm chart repository)

12.2.4.1: Example Dockerfile

Below is an example *Dockerfile* for an API microservice written using the FastAPI library. The Dockerfile uses Docker's multi-stage feature. First (at the install_deps stage), it installs dependencies and copies source code files to the image. The last stage (final) copies files from the install-deps stage to a distroless Python base image. You should use a distroless base image to make the image size and the attack surface smaller. A distroless image does not contain any Linux distribution inside it. Unfortunately, the below advertised *gcr.io/distroless/python* images are considered experimental at the time of writing this book and are not recommended for production.

Figure 12.2

```
FROM python:3.11 as install-deps

WORKDIR /microservice
COPY ./requirements.txt /microservice/requirements.txt
RUN pip install --no-cache-dir --upgrade -r
/microservice/requirements.txt
COPY ./app /microservice/app

FROM gcr.io/distroless/python3.11 as final
COPY --from=install-deps /microservice /microservice
WORKDIR /microservice
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port",
"80"]
```

12.2.4.2: Example Kubernetes Deployment

Below is an example Helm chart template *deployment.yaml* for a Kubernetes Deployment. The template code is given in double braces.

Figure 12.3

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "microservice.fullname" . }}
  labels:
    {{- include "microservice.labels" . | nindent 4 }}
spec:
  {{- if ne .Values.env "production" }}
  replicas: 1
  {{- end }}
  selector:
    matchLabels:
```

```
      {{- include "microservice.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      {{- with .Values.deployment.pod.annotations }}
      annotations:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      labels:
        {{- include "microservice.selectorLabels" . | nindent 8
}}
    spec:
      {{- with .Values.deployment.pod.imagePullSecrets }}
      imagePullSecrets:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      serviceAccountName: {{ include
"microservice.serviceAccountName" . }}
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.imageRegistry }}/{{
.Values.imageRepository }}:{{ .Values.im\
ageTag }}"
          imagePullPolicy: {{
.Values.deployment.pod.container.imagePullPolicy }}
          securityContext:
            {{- toYaml
.Values.deployment.pod.container.securityContext | nindent 12 }}
          {{- if .Values.httpServer.port }}
          ports:
            - name: http
              containerPort: {{ .Values.httpServer.port }}
              protocol: TCP
          {{- end }}
          env:
            - name: ENV
              value: {{ .Values.env }}
            - name: ENCRYPTION_KEY
              valueFrom:
                secretKeyRef:
                  name: {{ include "microservice.fullname" . }}
                  key: encryptionKey
            - name: MICROSERVICE_NAME
              value: {{ include "microservice.fullname" . }}
            - name: MICROSERVICE_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: MICROSERVICE_INSTANCE_ID
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: NODE_NAME
              valueFrom:
                fieldRef:
                  fieldPath: spec.nodeName
            - name: MYSQL_HOST
              value: {{ .Values.database.mySql.host }}
            - name: MYSQL_PORT
              value: "{{ .Values.database.mySql.port }}"
            - name: MYSQL_USER
              valueFrom:
                secretKeyRef:
```

```
                name: {{ include "microservice.fullname" . }}
                key: mySqlUser
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: {{ include "microservice.fullname" . }}
                  key: mySqlPassword
          livenessProbe:
            httpGet:
              path: /isAlive
              port: http
            failureThreshold: 3
            periodSeconds: 10
          readinessProbe:
            httpGet:
              path: /isReady
              port: http
            failureThreshold: 3
            periodSeconds: 5
          startupProbe:
            httpGet:
              path: /isStarted
              port: http
            failureThreshold: {{
.Values.deployment.pod.container.startupProbe.failureThr\
eshold }}
            periodSeconds: 10
          resources:
            {{- if eq .Values.env "development" }}
            {{- toYaml
.Values.deployment.pod.container.resources.development | nindent
1\
2 }}
            {{- else if eq .Values.env "integration"  }}
            {{- toYaml
.Values.deployment.pod.container.resources.integration | nindent
1\
2 }}
            {{- else }}
            {{- toYaml
.Values.deployment.pod.container.resources.production | nindent
12\
 }}
            {{- end}}
      {{- with .Values.deployment.pod.nodeSelector }}
      nodeSelector:
        {{- toYaml . | nindent 8 }}
      {{- end }}
      {{- with .Values.deployment.pod.affinity }}
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchLabels:
                  app.kubernetes.io/name: {{ include
"microservice.name" . }}
              topologyKey: "kubernetes.io/hostname"
        {{- toYaml . | nindent 8 }}
      {{- end }}
      {{- with .Values.deployment.pod.tolerations }}
      tolerations:
```

```
        {{- toYaml . | nindent 8 }}
      {{- end }}
```

The values (indicated by `.Values.<something>`) in the above template come from a *values.yaml* file. Below is an example *values.yaml* file to be used with the above Helm chart template.

Figure 12.4

```
imageRegistry: docker.io
imageRepository: pksilen2/backk-example-microservice
imageTag:
env: production
auth:
  # Authorization Server Issuer URL
  # For example
  #
http://keycloak.platform.svc.cluster.local:8080/auth/realms/<my-
realm>
  issuerUrl:

  # JWT path where for user's roles,
  # for example 'realm_access.roles'
  jwtRolesClaimPath:
secrets:
  encryptionKey:
database:
  mySql:
    # For example:
    # my-microservice-mysql.default.svc.cluster.local or
    # cloud database host
    host:
    port: 3306
    user:
    password: &mySqlPassword ""
mysql:
  auth:
    rootPassword: *mySqlPassword
deployment:
  pod:
    annotations: {}
    imagePullSecrets: []
    container:
      imagePullPolicy: Always
      securityContext:
        privileged: false
        capabilities:
          drop:
            - ALL
        readOnlyRootFilesystem: true
        runAsNonRoot: true
        runAsUser: 65532
        runAsGroup: 65532
        allowPrivilegeEscalation: false
      env:
      startupProbe:
        failureThreshold: 30
      resources:
```

```
      development:
        limits:
          cpu: '1'
          memory: 768Mi
        requests:
          cpu: '1'
          memory: 384Mi
      integration:
        limits:
          cpu: '1'
          memory: 768Mi
        requests:
          cpu: '1'
          memory: 384Mi
      production:
        limits:
          cpu: 1
          memory: 768Mi
        requests:
          cpu: 1
          memory: 384Mi
  nodeSelector: {}
  tolerations: []
  affinity: {}
```

Notice the `deployment.pod.container.securityContext` object in the above file. It is used to define the security context for a microservice container.

By default, the security context should be the following:

- Container should not be privileged
- All capabilities are dropped
- Container filesystem is read-only
- Only a non-root user is allowed to run inside the container
- Define the non-root user and group under which the container should run
- Disallow privilege escalation

You can remove things from the above list only if required for a microservice. For example, if the microservice must write to the filesystem for some valid reason, then the filesystem cannot be defined as read-only.

12.2.4.3: Example CI/CD Pipeline

Below is a GitHub Actions CI/CD workflow for a Python microservice. The declarative workflow is written in YAML. The workflow file should be in the microservice's source code repository in the *.github/workflows* directory. Steps in the workflow are described in more detail after the example.

Figure 12.5

```
name: CI/CD workflow
on:
  workflow_dispatch: {}
  push:
    branches:
      - main
    tags-ignore:
      - '**'
jobs:
  build:
    runs-on: ubuntu-latest
    name: Build with Python 3.11
    steps:
      - name: Checkout Git repo
        uses: actions/checkout@v4

      - name: Setup Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.11'
          cache: 'pip'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Lint source code
        run: pylint src

      - name: Run unit tests
        run: python -m coverage run -m unittest

      - name: Report unit test coverage
        run: python -m coverage xml

      - name: Setup integration testing environment
        run: docker-compose --env-file .env.ci up --build -d

      - name: Run integration tests
        run: scripts/run-integration-tests-in-ci.sh

      - name: OWASP ZAP API scan
        uses: zaproxy/action-api-scan@v0.5.0
        with:
          target: http://localhost:8080/openapi.json
          fail_action: true
          cmd_options: -I -z "-config
replacer.full_list(0).description=auth1
                             -config
replacer.full_list(0).enabled=true
                             -config
replacer.full_list(0).matchtype=REQ_HEADER
                             -config
```

```
replacer.full_list(0).matchstr=Authorization
                               -config
replacer.full_list(0).regex=false
                               -config
'replacer.full_list(0).replacement=Bearer ZXlK...aG\
JHZ='"

      - name: Tear down integration testing environment
        run: docker-compose --env-file .env.ci down -v

      - name: Static code analysis with SonarCloud scan
        uses: sonarsource/sonarcloud-github-action@master
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}

      - name: 3rd party software license compliance analysis
with FOSSA
        uses: fossas/fossa-action@main
        with:
          api-key: ${{ secrets.FOSSA_API_KEY }}
          run-tests: false

      - name: Lint Dockerfile
        uses: hadolint/hadolint-action@v3.1.0

      - name: Log in to Docker registry
        uses: docker/login-action@v3
        with:
          registry: docker.io
          username: ${{ secrets.DOCKER_HUB_USERNAME }}
          password: ${{ secrets.DOCKER_HUB_TOKEN }}

      - name: Extract latest Git tag
        uses: actions-ecosystem/action-get-latest-tag@v1
        id: extractLatestGitTag

      - name: Set up Docker Buildx
        id: setupBuildx
        uses: docker/setup-buildx-action@v3

      - name: Cache Docker layers
        uses: actions/cache@v3
        with:
          path: /tmp/.buildx-cache
          key: ${{ runner.os }}-buildx-${{ github.sha }}
          restore-keys: |
            ${{ runner.os }}-buildx-

      - name: Extract metadata for building and pushing Docker
image
        id: dockerImageMetadata
        uses: docker/metadata-action@v5
        with:
          images: ${{ secrets.DOCKER_REGISTRY_USERNAME
}}/example-microservice
          tags: |
            type=semver,pattern={{version}},value=${{
steps.extractLatestGitTag.outputs.v\
alue }}

      - name: Build and push Docker image
```

```
        id: dockerImageBuildAndPush
        uses: docker/build-push-action@v5
        with:
          context: .
          builder: ${{ steps.setupBuildx.outputs.name }}
          push: true
          cache-from: type=local,src=/tmp/.buildx-cache
          cache-to: type=local,dest=/tmp/.buildx-cache
          tags: ${{ steps.dockerImageMetadata.outputs.tags }}
          labels: ${{ steps.dockerImageMetadata.outputs.labels
}}

      - name: Docker image vulnerability scan with Anchore
        id: anchoreScan
        uses: anchore/scan-action@v3
        with:
          image: ${{ secrets.DOCKER_REGISTRY_USERNAME
}}/example-microservice:latest
          fail-build: false
          severity-cutoff: high

      - name: Upload Anchore scan SARIF report
        uses: github/codeql-action/upload-sarif@v1
        with:
          sarif_file: ${{ steps.anchoreScan.outputs.sarif }}

      - name: Install Helm
        uses: azure/setup-helm@v3
        with:
          version: v3.13.0

      - name: Extract microservice version from Git tag
        id: extractMicroserviceVersionFromGitTag
        run: |
          value="${{ steps.extractLatestGitTag.outputs.value }}"
          value=${value:1}
          echo "::set-output name=value::$value"

      - name: Update Helm chart versions in Chart.yaml
        run: |
          sed -i "s/^version:.*/version: ${{
steps.extractMicroserviceVersionFromGitTag.o\
utputs.value }}/g" helm/example-microservice/Chart.yaml
          sed -i "s/^appVersion:.*/appVersion: ${{
steps.extractMicroserviceVersionFromGi\
tTag.outputs.value }}/g" helm/example-microservice/Chart.yaml

      - name: Update Docker image tag in values.yaml
        run: |
          sed -i "s/^imageTag:.*/imageTag: {{
steps.extractMicroserviceVersionFromGitTag.\
outputs.value }}@${{
steps.dockerImageBuildAndPush.outputs.digest }}/g" helm/example-
micr
oservice/values.yaml

      - name: Lint Helm chart
        run: helm lint -f helm/values/values-minikube.yaml
helm/example-microservice

      - name: Static code analysis for Helm chart with Checkov
        uses: bridgecrewio/checkov-action@master
```

```
        with:
          directory: helm/example-microservice
          quiet: false
          framework: helm
          soft_fail: false

      - name: Upload Checkov SARIF report
        uses: github/codeql-action/upload-sarif@v1
        with:
          sarif_file: results.sarif
          category: checkov-iac-sca

      - name: Configure Git user
        run: |
          git config user.name "$GITHUB_ACTOR"
          git config user.email
"$GITHUB_ACTOR@users.noreply.github.com"

      - name: Package and publish Helm chart
        uses: helm/chart-releaser-action@v1.5.0
        with:
          charts_dir: helm
        env:
          CR_TOKEN: "${{ secrets.GITHUB_TOKEN }}"
```

---

1. Checkout the microservice's Git repository
2. Setup Python 3.11
3. Install dependencies
4. Lint source code
5. Execute unit tests (needs coverage library in requirements.txt)
6. Report coverage
7. Set up an integration testing environment using Docker's `docker-compose up` command. After executing the command, the microservice is built, and all the dependencies in separate containers are started. These dependencies can include other microservices and, for example, a database and a message broker, like Apache Kafka
8. Execute integration tests. This script will first wait until all dependencies are up and ready. This waiting is done running a container using the dokku/wait image.
9. Perform DAST with ZAP API scan. For the scan, we define the URL to the OpenAPI 3.0 specification against which the scan will be made. We also give command

options to set a valid Authorization header for the HTTP requests made by the scan

10. Tear down the integration testing environment
11. Perform static code analysis using SonarCloud You need to have the following file in the root of the source code repository:

Figure 12.6

```
sonar.projectKey=<sonar-project-key>
sonar.organization=<sonar-organization>

sonar.python.coverage.reportPaths=coverage.xml
```

12. Check 3rd party software license compliance using FOSSA
13. Lint Dockerfile
14. Log in to Docker Hub
15. Extract the latest Git tag for further use
16. Setup Docker Buildx and cache Docker layers
17. Extract metadata, like the tag and labels for building and pushing a Docker image
18. Build and push a Docker image
19. Perform a Docker image vulnerability scan with Anchore
20. Upload the Anchore scan report to the GitHub repository
21. Install Helm
22. Extract the microservice version from the Git tag (remove the ‘v’ letter before the version number)
23. Replace Helm chart versions in the Helm chart’s *Chart.yaml* file using the Linux *sed* command
24. Update the Docker image tag in the *values.yaml* file
25. Lint the Helm chart and perform static code analysis for it
26. Upload the static code analysis report to the GitHub repository and perform git user configuration for the next step

27. Package the Helm chart and publish it to GitHub Pages

Some of the above steps are parallelizable, but a GitHub Actions workflow does not currently support parallel steps in a job. In *Jenkins*, you can easily parallelize stages using a *parallel* block.

You could also execute the unit tests and linting when building a Docker image by using the following kind of *Dockerfile*:

Figure 12.7

```
FROM python:3.11 as builder

WORKDIR /microservice
COPY ./requirements.txt /microservice/requirements.txt
RUN pip install --no-cache-dir --upgrade -r
/microservice/requirements.txt
COPY ./app /microservice/app
RUN pylint src
RUN python -m coverage run -m unittest
RUN python -m coverage xml
# You must implement sending unit test coverage report
# to SonarQube/SonarCloud here

FROM gcr.io/distroless/python3.11 as final
COPY --from=builder /microservice /microservice
WORKDIR /microservice
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port",
"80"]
```

The problem with the above solution is that you don't get a clear indication of what failed in a build. You must examine the output of the Docker build command to see if linting or unit tests failed. Also, you cannot use the SonarCloud GitHub Action anymore. You must implement SonarCloud reporting in the *builder* stage of the *Dockerfile* (after completing the unit testing to report the unit test coverage to SonarCloud).

12.2.5: Deploy

In the *Deploy* phase, released software is deployed to an execution environment automatically. A software component can be automatically deployed after a successful CI/CD pipeline run. This is called continuous deployment (CD). Notice that both *continuous delivery* and *continuous deployment* are abbreviated as CD. This can cause unfortunate misunderstandings. Continuous delivery is about releasing

software automatically, and continuous deployment is about automatically deploying released software to one or more environments. These environments include, for example, a CI/CD environment, staging environment(s) and finally, production environment(s). There are different ways to automate software deployment. One modern and popular way is GitOps, which uses a Git repository or repositories to define automatic deployments to different environments using a declarative approach. GitOps can be configured to update an environment automatically when new software is released. This is typically done for the CI/CD environment, which should always be kept up-to-date and contain the latest software component versions. Notable GitOps solutions are, for example, Flux and Argo CD.

GitOps can also be configured to deploy automatically and regularly to a staging environment. A staging environment replicates a production environment. It is an environment where end-to-end functional and non-functional tests are executed before the software is deployed to production. You can use multiple staging environments to speed up the continuous deployment to production. It is vital that all needed testing is completed before deploying to production. Testing can take a couple of days to validate the stability of the software. If testing in a staging environment requires three days and you set up three staging environments, you can deploy to production every day. On the other hand, if testing in a staging environment takes one week and you have only one staging environment, you can deploy to production only once a week. (Assuming that all tests execute successfully) Deployment to a production environment can also be automated. Or it can be triggered manually after completing all testing in a staging environment.

12.2.6: Operate

Operate is the phase when the software runs in production. In this phase, it needs to be ensured that software updates (like security patches) are deployed timely. Also, the production

environment's infrastructure and platform should be kept up-to-date and secure.

12.2.7: Monitor

Monitor is the phase when a deployed software system is monitored to detect any possible problems. Monitoring should be automated as much as possible. It can be automated by defining rules for alerts triggered when the software system operation requires human intervention. These alerts are typically based on various metrics collected from the microservices, infrastructure, and platform. Prometheus is a popular system for collecting metrics and triggering alerts.

The basic monitoring workflow follows the path below:

1. Monitor alerts
2. If an alert is triggered, investigate metrics in the relevant dashboard(s)
3. Check logs for errors in relevant services
4. Distributed tracing can help to visualize if and how requests between different microservices are failing

The following needs to be implemented to make monitoring possible and easy:

- Logging to standard input
- Distributed tracing
- Metrics collection
- Metrics visualization
- Alerting

12.2.7.1: Logging to Standard Input

Each service must log to the standard output. If your microservice is using a 3rd party library that logs to the standard output, choose a library that allows you to configure

the log format or request the log format configurability as an enhancement to the library. Choose a standardized log format and use it in all microservices, e.g., use Syslog format or OpenTelemetry Log Data Model (defined in a later section). Collect logs from each microservice to a centralized location, like an ElasticSearch database, where they are easily queriable.

12.2.7.2: Distributed Tracing

Integrate microservices with a distributed tracing tool, like Jaeger. A distributed tracing tool collects information about network requests microservices make.

12.2.7.3: Metrics Collection

Define what metrics need to be collected from each microservice. Typically, metrics are either *counters* (e.g., number of requests handled or request errors) or *gauges* (e.g., current CPU/memory usage). Collect metrics to calculate the service level indicators (SLIs). Below are listed the five categories of SLIs and a few examples of SLIs in each category.

- Availability
  - Is the service down?
  - Is a dependent service down?
- Error rate
  - How many times a service has been restarted due to a crash or unresponsiveness
  - Message processing errors
  - Request errors
  - Other errors
  - Different errors can be monitored by setting a metric label. For example, if you have a *request_errors* counter and a request produces an internal server

error, you can increment the *request_errors* counter with the label *internal_server_error* by one.

- Latency
  - Message or request processing duration
- Throughput
  - Number of messages/requests handled
- Saturation
  - Resource usage, e.g., CPU/memory/disk usage vs. requested amount

Instrument your microservice with the necessary code to collect the metrics. This can be done using a metrics collection library, like Prometheus.

12.2.7.4: Metrics Visualization

Create a main dashboard for each microservice to present the SLIs. Additionally, you should present *service level objectives* (SLOs) as dashboard charts. An example of an SLO is "service error rate must be less than x percent". When all SLOs are met, the dashboard should show SLO charts in green, and if an SLO is not met, the corresponding chart should be shown in red. You can also use yellow and orange colors to indicate that an SLO is still met, but the SLI value is no longer optimal. Use a visualization tool that integrates with the metrics collection tool, like Grafana with Prometheus. You can usually deploy metric dashboards as part of the microservice deployment.

12.2.7.5: Alerting

To define alerting rules, first define the service level objectives (SLOs) and base the alerting rules on them. If an SLO cannot be met, an alert should be triggered, and when the SLO is met again, the alert should automatically cancel. If you are using Kubernetes and Prometheus, you can define alerts using the Prometheus Operator and PrometheusRule CRs.

12.2.8: Software System Alerts Dashboard Example

Below is an example of a Grafana dashboard to visualize active alerts in a software system.

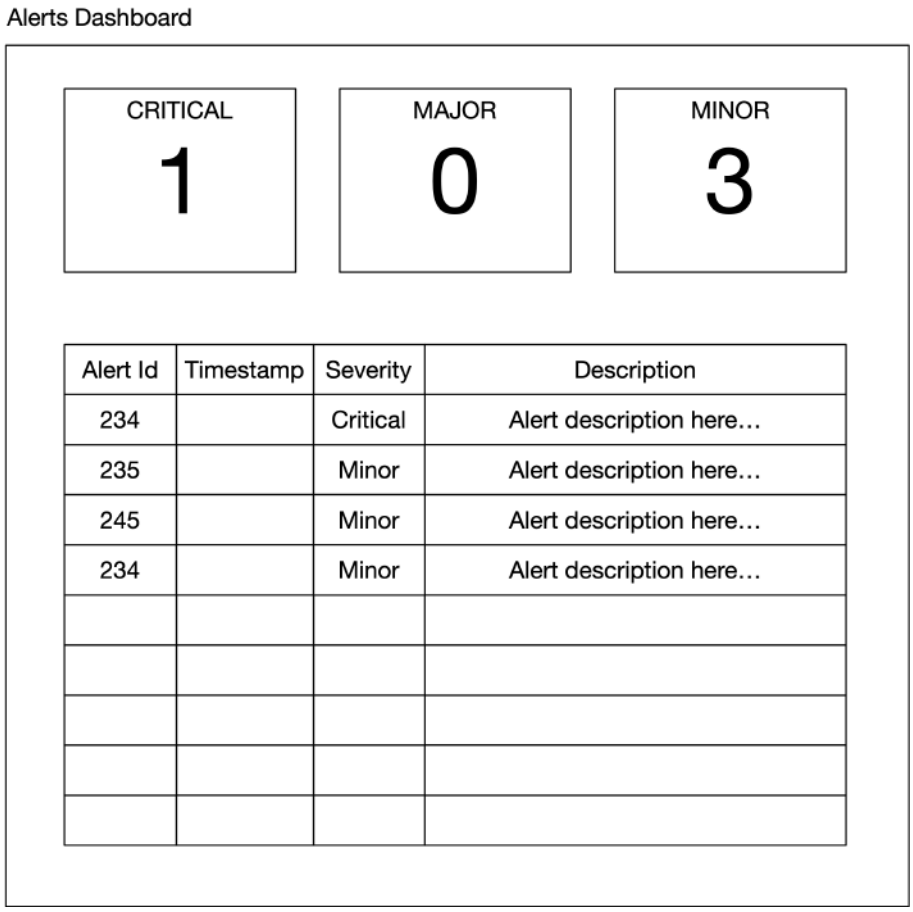

Figure 12.8. DevSecOps Diagram

12.2.9: Microservice Grafana Dashboard Example

Below is an example of a Grafana dashboard to visualize SLOs and SLIs for a single microservice. SLOs are presented as charts in the topmost section of the dashboard, and below them are five accordions, the first being opened to reveal the charts inside it. When opened, each accordion reveals charts for a particular SLI category. Each chart presents a specific SLI. In the figure below, the SLO 2 is shown in red

background, and it could indicate that the number of errors in the last hour is too high, for example.

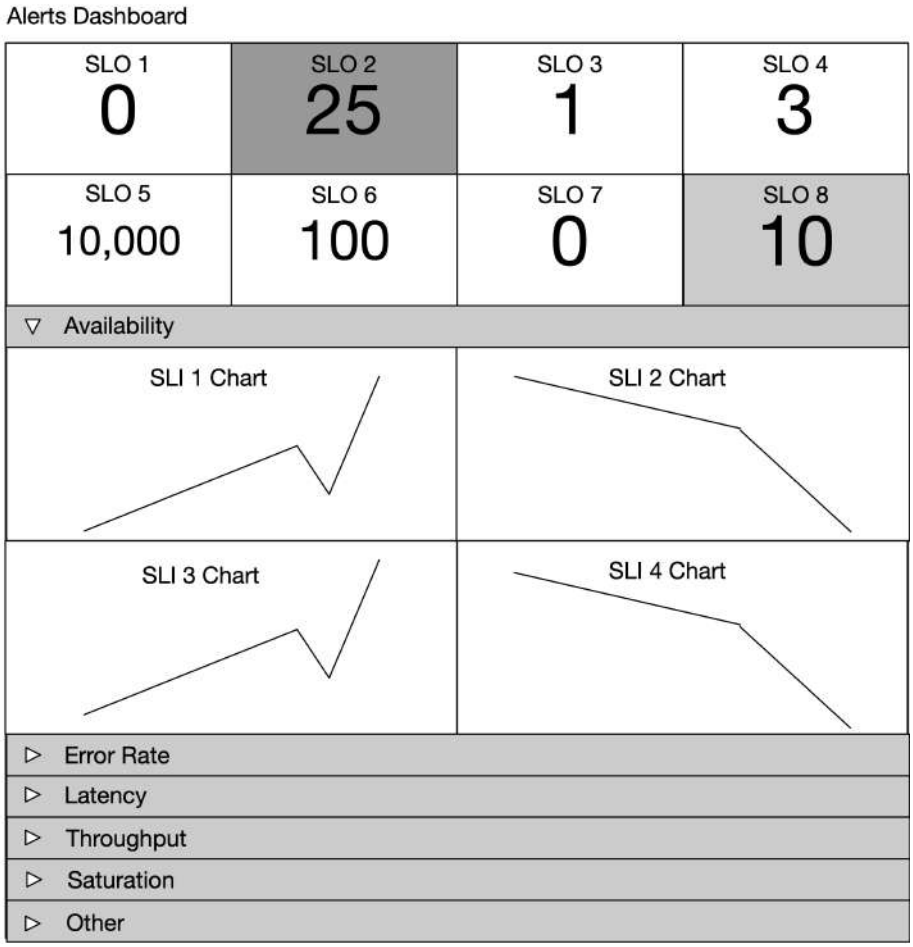

Figure 12.9. DevSecOps Diagram

Software operations staff connects back to the software development side of the DevOps lifecycle in the following ways:

- Ask for technical support
- File a bug report
- File an improvement idea

The first one will result in a solved case or bug report. The latter two will reach the *Plan* phase of the DevOps lifecycle. Bug reports usually enter the *Code* phase immediately, depending on the fault severity.

12.2.9.1: Logging

Implement logging in software components using the following logging severities:

- (CRITICAL/FATAL)
- ERROR
- WARNING
- INFO
- DEBUG
- TRACE

I don’t usually use the CRITICAL/FATAL severity at all. It might better to report all errors with the ERROR severity because then it is easy to query logs for errors using a single keyword only, for example:

Figure 12.10

```
kubectl logs <pod-name> | grep ERROR
```

You can add information about the criticality/fatality of an error to the log message itself. When you log an error for which there is a solution available, you should inform the user about the solution in the log message, e.g., provide a link to a troubleshooting guide or give an error code that can be used to search the troubleshooting guide.

Do not log too much information using the INFO severity because the logs become hard to read when there is too much noise. Consider carefully what should be logged with the INFO severity and what can be logged with the DEBUG severity instead. The default logging level of a microservice should preferably be WARNING (or INFO).

Use the TRACE severity to log only tracing information, e.g., detailed information about processing a single request, event, or message.

If you are implementing a 3rd party library, the library should allow customizing the logging if the library logs something.

There should be a way to set the logging level and allow the code that is using the library to customize the format in which log entries are written. Otherwise, 3rd party library log entries appear in the log in a different format than the log entries from the microservice itself, making the logs harder to read.

12.2.9.2: OpenTelemetry Log Data Model

This section describes the essence of the OpenTelemetry log data model version 1.12.0 (Please check OpenTelemetry Specification for possible updates).

A log entry is a JSON object containing the following properties:

Field Name	Description
Timestamp	Time when the event occurred. Nanoseconds since Unix epoch
TraceId	Request trace id
SpanId	Request span id
SeverityText	The severity text (also known as log level)
SeverityNumber	Numerical value of the severity
Body	The body of the log entry. You can include ISO 8601 timestamp and the severity/log level before the actual log message
Resource	A JSON object that describes the source of the log entry
Attributes	Additional information about the log event. This is a JSON object where custom attributes can be given

Below is an example log entry according to the above log data model.

Figure 12.11

```
{
  "Timestamp": "1586960586000000000",
  "TraceId": "f4dbb3edd765f620",
  "SpanId": "43222c2d51a7abe3",
  "SeverityText": "ERROR",
  "SeverityNumber": 9,
  "Body": "20200415T072306-0700 ERROR Error message comes here",
  "Resource": {
    "service.namespace": "default",
```

```
    "service.name": "my-microservice",
    "service.version": "1.1.1",
    "service.instance.id": "my-microservice-34fggd-56faae"
  },
  "Attributes": {
    "http.status_code": 500,
    "http.url": "http://example.com",
    "myCustomAttributeKey": "myCustomAttributeValue"
  }
}
```

The above JSON-format log entries might be hard to read as plain text on the console, for example, when viewing a pod's logs with the `kubectl logs` command in a Kubernetes cluster. You can create a small script that extracts only the `Body` property value from each log entry.

12.2.9.3: PrometheusRule Example

PrometheusRule custom resources (CRs) can be used to define rules for triggering alerts. In the below example, an *example-microservice-high-request-latency* alert will be triggered with a major severity when the median request latency in seconds is greater than one (request_latencies_in_seconds{quantile="0.5"} > 1).

Figure 12.12

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: example-microservice-rules
spec:
  groups:
  - name: example-microservice-rules
    rules:
    - alert: example-microservice-high-request-latency
      expr: request_latencies_in_seconds{quantile="0.5"} > 1
      for: 10m
      labels:
        application: example-microservice
        severity: major
        class: latency
      annotations:
        summary: "High request latency on {{ $labels.instance
}}"
        description: "{{ $labels.instance }} has a median
request latency above 1s (curre\
nt value: {{ $value }}s)"
```

13: Conclusion

This book has presented a lot of valuable principles and patterns. It can be challenging to grasp them all at once. For this reason, I suggest you prioritize learning. Topics I have found most important during many years of coding are the following:

- Code is read more often than written principle
    - For this reason, pay attention to the fact that your code is easily readable and understandable. To achieve this, you can use the uniform naming principle and avoid writing comments principle presented in this book
- Prefer object composition over inheritance principle
    - Consider always first if there is a *has-a* relationship between two objects, then use object composition. But if there is not a has-a relationship but a *is-a* relationship, then use inheritance
- Encapsulation principle
    - Don’t automatically implement attribute getters and setters in a class. They can break encapsulation. Only implement them when needed. If you have getters and setters for all or most attributes and little other behavior in your class, the class can be an anemic class. You might not be following the don’t ask, tell principle, which results in the feature envy code smell.
- Single responsibility principle
    - Whether it is a question about a software component or a class, try to make it have a single responsibility at a certain level of abstraction to keep the software component or class small enough. If the level of abstraction is high, the software component or class can become too large. What you can do is extract

functionality from the large class to smaller classes and make the original class use the smaller ones

- Program against interfaces principle
  - This is a necessity if you want to follow the open-closed principle
- Open-closed principle
  - When you put new functionality into new classes, instead of modifying existing classes and their methods, you usually cannot accidentally break any existing functionality
- TDD or USDD
  - When creating new functions, use TDD or USDD to make it less likely to forget to implement some failure scenarios or edge cases. Consider preferring TDD over USDD because of the following additional benefits:
    - Refactoring becomes a norm and helps you achieve a better design
    - Less stress and cognitive load
    - You are bound to write testable code
- Threat modeling
  - Helps you and your team to create more secure software
- Integration testing using BDD and ATDD
  - Define integration tests using BDD and formal specifications and make those integration tests acceptance tests of a feature. This formalized way of specifying a feature makes it less likely not to forget to write integration tests for the features of the software component.

Regarding design patterns, the following two patterns are the most valuable, and you can use them in almost every project. Other design patterns are not necessarily as widely used.

- (Abstract) Factory pattern
    - Use factories to instantiate different implementations of an interface
- Adapter pattern
    - You can create multiple implementations for a common interface using adapter classes that adapt another interface to the common interface

If you want to master more design patterns, I suggest to learn the following:

- Strategy pattern
    - Make the class behavior dynamic by depending on an abstract strategy with multiple implementations. When following this pattern, you implement different behaviors (strategies) in separate classes, making your code follow the single responsibility principle and open-closed principle
- Decorator pattern
    - Augment the behavior of a class or function without modifying the class or function. For functions, this pattern can be implemented using Python function decorators.
- Proxy pattern
    - Conditionally delegate to the wrapped class's or function's behavior. A good example of the proxy pattern is caching. The caching proxy class delegates to the wrapped class or function only when a result isn't available in the cache already. For functions, this pattern can be implemented using Python function decorators. For example, Python has a built-in `@cache` decorator that utilizes the proxy pattern.
- Command/Action pattern
    - Encapsulate functionality with parameters in a command/action object. This pattern allows you to

follow the open-closed principle because you can put new functionality in a new command/action class.

- State pattern
    - Don’t treat state as enums. Use objects that have attached behavior related to a particular state. Following this pattern allows you to replace conditionals with polymorphism.
- Template method pattern
    - Put a common algorithm that calls an abstract method to a base class in a class inheritance hierarchy. This abstract method call in the common algorithm makes the method a template method. The final behavior of the common algorithm is refined by the subclasses that implement the abstract method called from the template method.

To fully embrace a principle or pattern, you need to apply it in real-life projects and see the benefits yourself. When you see the benefits in practice, the value of a principle or pattern becomes more evident and no longer feels like a law imposed by some authority that you are forced to obey.

14: Appendix A

Figure 14.1

```
import os
import traceback

from pydantic import BaseModel


def is_pydantic(object: object):
    return type(object).__class__.__name__ == 'ModelMetaclass'


def to_entity_dict(dto: BaseModel):
    entity_dict = dict(dto)
    for key, value in entity_dict.items():
        try:
            if (
                isinstance(value, list)
                and len(value)
                and is_pydantic(value[0])
            ):
                entity_dict[key] = [
                    item.Meta.orm_model(**to_entity_dict(item))
                    for item in value
                ]
            elif is_pydantic(value):
                entity_dict[key] = value.Meta.orm_model(
                    **to_entity_dict(value)
                )
        except AttributeError:
            raise AttributeError(
                f'Found nested Pydantic model in {dto.__class__}
but Meta.orm_model was n\
ot specified.'
            )
    return entity_dict


def get_stack_trace(error: Exception | None):
    return (
        repr(traceback.format_exception(error))
        if error and os.environ.get('ENV') != 'production'
        else None
    )
```

Figure 14.2

```
from pydantic import BaseModel

from .OrderItem import OrderItem


class InputOrder(BaseModel):
    userId: str
    orderItems: list[OrderItem]
```

```
    class Config:
        orm_mode = True
```

Figure 14.3

```
from pydantic import BaseModel, PositiveInt

from ..entities.OrderItem import OrderItem as OrderItemEntity


class OrderItem(BaseModel):
    id: int
    salesItemId: str
    quantity: PositiveInt

    class Config:
        orm_mode = True

    class Meta:
        orm_model = OrderItemEntity
```

Figure 14.4

```
from pydantic.main import BaseModel

from .OrderItem import OrderItem


class OutputOrder(BaseModel):
    id: str
    userId: str
    orderItems: list[OrderItem]

    class Config:
        orm_mode = True
```

Figure 14.5

```
from sqlalchemy.orm import DeclarativeBase


class Base(DeclarativeBase):
    pass
```

Figure 14.6

```
from sqlalchemy import BigInteger
from sqlalchemy.orm import Mapped, mapped_column, relationship

from .Base import Base
from .OrderItem import OrderItem


class Order(Base):
    __tablename__ = 'orders'

    id: Mapped[int] = mapped_column(BigInteger(),
primary_key=True)
    userId: Mapped[int] = mapped_column(BigInteger(),
index=True)
```

```
    orderItems: Mapped[list[OrderItem]] =
relationship(lazy='joined')
```

Figure 14.7

```
from sqlalchemy import BigInteger, ForeignKey,
PrimaryKeyConstraint
from sqlalchemy.orm import Mapped, mapped_column

from .Base import Base


class OrderItem(Base):
    __tablename__ = 'orderitems'
    __table_args__ = (
        PrimaryKeyConstraint('orderId', 'id',
name='orderitems_pk'),
    )

    id: Mapped[int]
    salesItemId: Mapped[int] = mapped_column(BigInteger())
    quantity: Mapped[int]
    orderId: Mapped[int] =
mapped_column(ForeignKey('orders.id'))
```

Figure 14.8

```
from typing import Final


class OrderServiceError(Exception):
    def __init__(
        self,
        status_code: int,
        message: str,
        cause: Exception | None = None,
    ):
        self.__status_code: Final = status_code
        self.__message: Final = message
        self.__cause: Final = cause

    @property
    def status_code(self) -> int:
        return self.__status_code

    @property
    def message(self) -> str:
        return self.__message

    @property
    def cause(self) -> Exception | None:
        return self.__cause
```

Figure 14.9

```
from .OrderServiceError import OrderServiceError


class DatabaseError(OrderServiceError):
    def __init__(self, cause: Exception):
        super().__init__(500, 'Database error', cause)
```

Figure 14.10

```
from .OrderServiceError import OrderServiceError


class EntityNotFoundError(OrderServiceError):
    def __init__(self, entity_name: str, entity_id: int):
        super().__init__(
            404, f'{entity_name} with id {entity_id} not found'
        )
```

Figure 14.11

```
import strawberry

from ..dtos.InputOrder import InputOrder
from .InputOrderItem import InputOrderItem


@strawberry.experimental.pydantic.input(model=InputOrder)
class InputOrder:
    userId: strawberry.auto
    orderItems: list[InputOrderItem]
```

Figure 14.12

```
import strawberry

from ..dtos.OrderItem import OrderItem


@strawberry.experimental.pydantic.input(model=OrderItem,
all_fields=True)
class InputOrderItem:
    pass
```

Figure 14.13

```
import strawberry

from ..dtos.OutputOrder import OutputOrder
from .OutputOrderItem import OutputOrderItem


@strawberry.experimental.pydantic.type(model=OutputOrder)
class OutputOrder:
    id: strawberry.auto
    userId: strawberry.auto
    orderItems: list[OutputOrderItem]
```

Figure 14.14

```
import strawberry

from ..dtos.OrderItem import OrderItem


@strawberry.experimental.pydantic.type(model=OrderItem,
all_fields=True)
class OutputOrderItem:
    pass
```

15: Appendix B

Here is the source code for the `proto_to_dict` function:

Figure 15.1

```
# This is free and unencumbered software released into the
public domain
# by its author, Ben Hodgson <ben@benhodgson.com>.

# Anyone is free to copy, modify, publish, use, compile, sell,
or
# distribute this software, either in source code form or as a
compiled
# binary, for any purpose, commercial or non-commercial, and by
any
# means.

# In jurisdictions that recognise copyright laws, the author or
authors
# of this software dedicate any and all copyright interest in
the
# software to the public domain. We make this dedication for the
benefit
# of the public at large and to the detriment of our heirs and
# successors. We intend this dedication to be an overt act of
# relinquishment in perpetuity of all present and future rights
to this
# software under copyright law.

# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.
# IN NO EVENT SHALL THE AUTHORS BE LIABLE FOR ANY CLAIM, DAMAGES
OR
# OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE,
# ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
USE OR
# OTHER DEALINGS IN THE SOFTWARE.

# For more information, please refer to <http://unlicense.org/>

from google.protobuf.descriptor import FieldDescriptor

EXTENSION_CONTAINER = '___X'


TYPE_CALLABLE_MAP = {
    FieldDescriptor.TYPE_DOUBLE: float,
    FieldDescriptor.TYPE_FLOAT: float,
    FieldDescriptor.TYPE_INT32: int,
    FieldDescriptor.TYPE_INT64: int,
    FieldDescriptor.TYPE_UINT32: int,
```

```
    FieldDescriptor.TYPE_UINT64: int,
    FieldDescriptor.TYPE_SINT32: int,
    FieldDescriptor.TYPE_SINT64: int,
    FieldDescriptor.TYPE_FIXED32: int,
    FieldDescriptor.TYPE_FIXED64: int,
    FieldDescriptor.TYPE_SFIXED32: int,
    FieldDescriptor.TYPE_SFIXED64: int,
    FieldDescriptor.TYPE_BOOL: bool,
    FieldDescriptor.TYPE_STRING: str,
    FieldDescriptor.TYPE_BYTES: lambda b: b.encode('base64'),
    FieldDescriptor.TYPE_ENUM: int,
}


def repeated(type_callable):
    return lambda value_list: [
        type_callable(value) for value in value_list
    ]


def enum_label_name(field, value):
    return field.enum_type.values_by_number[int(value)].name


def proto_to_dict(
    pb, type_callable_map=TYPE_CALLABLE_MAP,
use_enum_labels=False
):
    result_dict = {}
    extensions = {}
    for field, value in pb.ListFields():
        type_callable = _get_field_value_adaptor(
            pb, field, type_callable_map, use_enum_labels
        )
        if field.label == FieldDescriptor.LABEL_REPEATED:
            type_callable = repeated(type_callable)

        if field.is_extension:
            extensions[str(field.number)] = type_callable(value)
            continue

        result_dict[field.name] = type_callable(value)

    if extensions:
        result_dict[EXTENSION_CONTAINER] = extensions
    return result_dict


def _get_field_value_adaptor(
    pb, field, type_callable_map=TYPE_CALLABLE_MAP,
use_enum_labels=False
):
    if field.type == FieldDescriptor.TYPE_MESSAGE:
        # recursively encode protobuf sub-message
        return lambda pb: proto_to_dict(
            pb,
            type_callable_map=type_callable_map,
            use_enum_labels=use_enum_labels,
        )

    if use_enum_labels and field.type ==
FieldDescriptor.TYPE_ENUM:
```

```
        return lambda value: enum_label_name(field, value)

    if field.type in type_callable_map:
        return type_callable_map[field.type]

    raise TypeError(
        'Field %s.%s has unrecognised type id %d'
        % (pb.__class__.__name__, field.name, field.type)
    )


def get_bytes(value):
    return value.decode('base64')


REVERSE_TYPE_CALLABLE_MAP = {
    FieldDescriptor.TYPE_BYTES: get_bytes,
}


def _get_field_mapping(pb, dict_value, strict):
    field_mapping = []
    for key, value in dict_value.items():
        if key == EXTENSION_CONTAINER:
            continue
        if key not in pb.DESCRIPTOR.fields_by_name:
            if strict:
                raise KeyError(
                    '%s does not have a field called %s' % (pb,
key)
                )
            continue
        field_mapping.append(
            (
                pb.DESCRIPTOR.fields_by_name[key],
                value,
                getattr(pb, key, None),
            )
        )

    for ext_num, ext_val in dict_value.get(
        EXTENSION_CONTAINER, {}
    ).items():
        try:
            ext_num = int(ext_num)
        except ValueError:
            raise ValueError('Extension keys must be integers.')
        if ext_num not in pb._extensions_by_number:
            if strict:
                raise KeyError(
                    '%s does not have a extension with number
%s. Perhaps you forgot to i\
mport it?'
                    % (pb, key)
                )
            continue
        ext_field = pb._extensions_by_number[ext_num]
        pb_val = None
        pb_val = pb.Extensions[ext_field]
        field_mapping.append((ext_field, ext_val, pb_val))

    return field_mapping
```

```
def _string_to_enum(field, input_value):
    enum_dict = field.enum_type.values_by_name
    try:
        input_value = enum_dict[input_value].number
    except KeyError:
        raise KeyError(
            '`%s` is not a valid value for field `%s`'
            % (input_value, field.name)
        )
    return input_value
```