MongoDB for Jobseekers

Reach new heights in your career with MongoDB

Justin Jenkins

bpb

MongoDB for Jobseekers

Reach new heights in your career with MongoDB

Justin Jenkins

www.bpbonline.com

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-255

www.bpbonline.com

Dedicated to

*My wife **Jessica***

*And my sons **Caderyn** & **Evan***

About the Author

Justin Jenkins has nearly two decades of experience in the tech industry, having held various roles. Early on, he found a passion for databases. An early adopter of MongoDB after years of working with SQL, he has made contributions to startups, well known tech companies, and non-profit organizations. Recognized by MongoDB as an Enthusiast, he shares his expertise through resources like this book, online courses which you can find on LinkedIn Learning, and speaking events. Having lived on both coasts of the US, he currently resides in Colorado Springs, Colorado with his wife and two incredibly fun, and incredibly exhausting boys. When not coding, he enjoys cooking, supporting soccer teams like the Seattle Sounders and Manchester United, and exploring the outdoors. He constantly mediates on the words of Mark 9:23-24 and hopes to for the rest of his life.

About the Reviewers

- **Darshan Jayarama** is a seasoned professional in the field of database administration. With a Bachelor of Engineering degree and over a decade of experience, he has excelled as a MongoDB DBA. Currently serving as a Senior Technical Services Engineer at MongoDB for the past four years, he brings extensive expertise in MongoDB, MySQL, and Sybase. His in-depth knowledge, problem-solving abilities, and commitment to delivering exceptional technical solutions have made him a trusted resource in the industry.

- **Vinicius Grippa** is a Percona Senior Database Engineer, Oracle Ace, and author of the book Learning MySQL. Vinicius has a Bachelor's degree in Computer Science and has worked with databases for over 15 years. He has experience designing databases for mission-critical applications and has become a specialist in MySQL and MongoDB ecosystems.

Acknowledgement

I am deeply grateful to my family for their unwavering patience and support throughout the process of writing this book.

To my parents, David and Giselle, I extend a special thank you for tolerating that chaotic pile of computer parts in your basement during my formative years. Your belief in my abilities and continuous encouragement have shaped my career in more ways than I can express.

I would like to extend my gratitude to my employers over the years for providing me with opportunities to explore and incorporate MongoDB in various capacities. Their trust in my skills has allowed me to grow both personally and professionally.

A heartfelt appreciation goes out to the team at BPB Publishing for their support and for providing me with the invaluable opportunity to write this book.

It almost goes without saying, but I would also like to aknowlege MongoDB for developing such an exceptional product and I am grateful for the opportunities it has afforded me to share my experiences.

Lastly, I would like to acknowledge the teachers and specialists who supported me during my early years of education, when I faced challenges with reading and writing. Your dedication and belief in my potential have inspired me to overcome obstacles and your impact on my life will always be cherished. I hope that my journey serves as a testament to how much giving a "little extra help" can mean.

Preface

Within these pages, we invite you on an immersive journey to discover the remarkable capabilities of MongoDB and gain the expertise to harness its full potential. Whether you are an experienced professional seeking to deepen your knowledge or an enthusiastic beginner eager to embrace this cutting-edge technology, this comprehensive guide will be your trusted companion.

As we explore MongoDB, you will uncover its unique qualities that sets it apart from traditional databases. MongoDB's Document Model revolutionizes data organization and retrieval, propelling it to the forefront of the industry.

We will guide you through practical skills, including installation on various operating systems and essential tools such as MongoDB Shell and Compass. Engaging examples will demonstrate how MongoDB's document-based approach efficiently stores and organizes complex data structures.

You will learn to master fundamental operations such as creating, updating, and deleting documents within MongoDB. We will equip you with the skills to work with complex data types as well, using MongoDB's powerful array and embedded document features.

We will delve into crucial concepts such as indexing and collection management, as well as how to enhance query performance and effectively handle data scaling challenges. From Data migration, security, backup strategies, and encryption, we will provide comprehensive guidance to safeguard your data.

Programming with MongoDB is easy and powerful. To show this, we will will explore code examples in Python, Node.JS, and PHP, enabling you to harness MongoDB's capabilities within your applications. We will also discover MongoDB's cloud services, provided by MongoDB Atlas, which offers cloud hosting, database tools, and effortless application development and maintenance.

Finally, we will provide valuable interview preparation resources, bolstering your MongoDB expertise and increasing your chances of success in the job market.

Prepare for an adventure as we unravel the core concepts and features of MongoDB. Whether you aspire to become a MongoDB professional, strengthen your database management skills, or satiate your curiosity, this book is your gateway to unlocking the true potential of MongoDB and your future career opertunties.

Here is a brief overview of the chapters:

Chapter 1: Why MongoDB? – This chapter will start off with a brief history of databases, and how MongoDB and its Document Modal are different, as well as how these differences make it so powerful and popular.

Chapter 2: MongoDB Jobs and Roles – This chapter is dedicated to providing some context and explaination of different job roles available to those with MongoDB experience as well as what sections of this book you might want to pay particular attention to, based on each particular job role.

Chapter 3: Getting Started – In this chapter, we will cover installing MongoDB Server on various different operating systems, as well as Docker and MongoDB's Cloud service Atlas. Additionally, we will walk through installing the MongoDB Shell, Tools and the official GUI MongoDB Compass.

Chapter 4: A Better Way to Store Data – Documents – In this chapter, we will cover some key details about Documents, rules you should know, and most importantly, some examples of real-life data put into Documents that will help you understand how you might put your own data into Documents.

Chapter 5: Let's Do It - Create, Update and Delete Documents – By the end of this chapter, you will have a basic idea of how to create, update and delete documents within MongoDB. We will also discuss how to do each of these actions to multiple documents at one time.

Chapter 6: Getting What You Want – Querying – By the end of this chapter, you should be comfortable with querying MongoDB using various query operators, have an understanding of the core concepts of querying with MongoDB, as well as tools that you can use to deal with unique challenges such as case sensitively in MongoDB.

Chapter 7: Complex Data, Made Simple – MongoDB's document model allows us to store much more complex data than legacy databases, by using arrays and embedded documents. These data types give us a lot of flexibility, but with flexibility can come complexity. Fortunately, the MongoDB Query API provides robust tools for dealing with these complex types. By the end of this chapter, you should have a solid understanding of how to perform typical queries to find and modify arrays as well as embedded objects. Additionally, you will have a high-level understanding of the many MongoDB operators available for these data types, and how to use them.

Chapter 8: The MongoDB Aggregation Framework – For more complex cases, MongoDB offers what is called the Aggregation Framework, which allows a

structured way to formulate a series of steps, called a "pipeline", to get back just the data you need. In this chapter, we will discuss this framework and dive into some examples of its use. Covering the Aggregation Framework in-depth is beyond the scope of this book. However, by the end of this chapter, you should have a solid idea of how the framework works and how you can use it to fit your needs.

Chapter 9: Planning for Performance - Collections and Indexes – In this chapter, we will explain what an index is, how MongoDB uses indexes, different index types, different collection types, and how to create, configure and delete indexes and collections. By the end of this chapter, you should have a solid foundation of indexing and collection options in MongoDB and lots of areas you can look further into, if you want to learn more.

Chapter 10: Getting In and Getting Out - Data Migration – By the end of this chapter, you should feel comfortable with importing and exporting in MongoDB using MongoDB Compass, MongoDB Database Tools as well as scripting methods. Additionally, you will have a good idea of how to transfer data between collections and databases.

Chapter 11: Make It Great - Configuration and Monitoring – In this chapter, we will consider how the server's start, stop and restart processes work, as well as how to configure your server. We will focus on important settings you will need to know, to properly administrate your server, as well as various monitoring tools and tips for troubleshooting.

Chapter 12: Seamless Scaling – Replication and Sharding – In this chapter, we will cover the core concepts of replication in MongoDB, discussing how to setup a replica set and how to leverage the concept of "sharding" for horizontal scaling using MongoDB. By the end of this chapter, you should be comfortable setting up a basic replica set and have a knowledge of how to administrate that replica set. You will also have a solid idea of how sharding works in MongoDB and how and why you would use it for your application.

Chapter 13: Being Proactive – Security and Backups – In this chapter, we will begin by discussing how to protect your database via authentication, authorization and roles. Then we will transition to how to effectively backup and restore your databases, wrapping up with a brief discussion about database encryption.

Chapter 14: Making Stuff – Programming with MongoDB – By the end of this chapter, you should have a solid idea of how to connect and perform basic queries against MongoDB, using code written in Python, Node.JS (JavaScript) and PHP.

Make sure to see the section about the book's free GitHub Codespace, where you can try some of this code out for yourself, without installing anything locally.

Chapter 15: Tools for Success – MongoDB Shell and Compass UI – In this chapter, we will dig a bit deeper into some of the most useful and common tools you will use with MongoDB. We will learn how to configure and personalize the MongoDB Shell, mongosh, as well as how to create useful custom functions you can use within the shell. Then, we will explore the MongoDB Visual Studio Code extension, and MongoDB Playgrounds. Lastly, we will do a bit of a review, as well as a more expansive investigation of MongoDB Compass, the official GUI for MongoDB. By the end of this chapter, you should be empowered to take your use of all these tools to the next level.

Chapter 16: Cloud Services – MongoDB Atlas – This chapter will only briefly cover a couple of the key aspects of the cloud services offered by MongoDB Atlas, as indeed, a whole book in itself could be written about Atlas. With that said, after reading this chapter, you should have a solid understanding of what Atlas has to offer and how you can use it to leverage the power of MongoDB even further. Atlas has essentially three main categories of features, two of which we will talk about in this chapter: cloud hosting services and database tools, such as Search and Charts. In the next chapter, we will discuss the third category, Atlas Application Services.

Chapter 17: MongoDB Atlas – Application Services – We will be building a React app using Atlas App Services which allows us to build on top of MongoDB, without actually having to run or maintain our own Replica Set, or server. Rather, we will rely on a shared free MongoDB Cluster, Atlas Functions, the Realm SDK for Web and the Atlas' Data API.

Chapter 18: Jobseeker – Interview Prep – This chapter will present fifty different interview-like questions, at various levels of difficulty, as well as a response. These may not be the exact questions you would be asked in an interview, for which MongoDB skills are required, but you can expect to be asked some variation of them. For each question, a response is offered, as well as a reference to the chapter we discussed the topic in general, or directly. We will discuss how to use this chapter's questions in the next section.

Chapter 19: Conclusion – In this final chapter, we will briefly touch on a few more complex topics such as Change Streams, Transactions in MongoDB, GridFS for storing large files and some comparisons between SQL queries and their MongoDB equivalents.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/9kqvjnp

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/MongoDB-For-Jobseekers**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. Why MongoDB? 1

Introduction 1

Structure 2

Objectives 2

Recipes as Data 2

Find me a recipe query 4

The history of data 4

Databases of clay 4

Computer Databases 6

Relational databases 7

Relational Databases vs Document Databases 7

Bringing separate data together 10

Data that goes together, can live together 11

Team document breakdown 13

Thinking of data differently 15

This is why MongoDB 15

Conclusion 16

Challenge – Storing data in a document 16

2. MongoDB Jobs and Roles 19

Introduction 19

Structure 19

Objectives 20

Interest in MongoDB 20

Jobs and Career Paths 21

Job Roles 21

Full-stack Developer 22

Data Engineer 22

Database Administrator 23
DevOps Engineer 23
Business Intelligence Analyst 23
Data Scientist 23
Technical Consultant 24
Technical Writer 24
Future MongoDB Jobs 24
Example Interview Questions 24
Questions 25
Conclusion 25

3. Getting Started 27
Introduction 27
Structure 27
Objectives 28
Prerequisites 28
Installing MongoDB 28
Installing MongoDB on Windows 29
Installing MongoDB server, compass, and tools 30
Connecting to MongoDB Server 32
Installing MongoDB on macOS 33
Installing MongoDB Server, Shell and Tools 33
Running MongoDB Server 34
Connecting to MongoDB Server via Compass 35
Installing MongoDB on Docker 36
Running MongoDB server on demand 36
Persisting database data files 37
Connecting to MongoDB on Docker 37
Connect via the MongoDB Shell mongosh 37
Connect via MongoDB Compass 38
Running MongoDB Server via Docker Compose 38

Setting up MongoDB on MongoDB Atlas Cloud 39
Conclusion 42

4. A Better Way to Store Data – Documents 43
Introduction 43
Structure 43
Objectives 44
Importing example documents 44
Importing with MongoDB Compass 44
Importing on the Command Line with mongoimport 47
What is a Document? 47
Other Considerations 48
Document Structure 49
MongoDB Shell Commands 50
More about types 50
String 50
Numbers 50
Dates 50
Epoch dates 51
Arrays and objects 52
Types when importing/exporting 52
Examples of documents 52
Stock data 53
User profile 54
Recipe 55
Home sale listing 57
Conclusion 58

5. Let's Do It – Create, Update and Delete Documents 59
Introduction 59
Structure 59
Objectives 60

Creating Documents 60
Using the MongoDB Shell *60*
Inserting a Document *62*
View Our New Document *62*
Other ways to query *63*
Find By ObjectId *63*
Find By Document Field *63*
Find one document *63*
Creating more complex documents *64*
Using the MongoDB Shell as a JavaScript Shell *64*
Inserting multiple documents *65*
Insert using MongoDB Compass *66*
Updating Documents 67
Adding new fields *67*
Removing fields *68*
Updating multiple documents *68*
"Upsert" a Document *69*
Updating Using MongoDB Compass *70*
Deleting Documents 71
Conclusion 72

6. Getting What You Want – Querying 73
Introduction 73
Structure 73
Objectives 74
Importing Example Documents 74
MongoDB Shell vs MongoDB Compass 74
MongoDB Shell Queries *74*
MongoDB compass queries *75*
Querying MongoDB 76
Why cursors? *76*

The MongoDB Query API 76
Using Filter to match documents 77
Using Projection to Control Output 77
Using sort() to Order Output 79
Using Variables in Queries 81
Using count() and limit() and skip() 82
MongoDB Query Operators 83
Comparison Operators 83
Using Operators in update queries 85
Field Update Operators 85
Atomic Operations 87
Logical Operators 87
Element Operators 89
Objects and arrays 90
Query Case Sensitivity 91
Using regex queries 91
Options for Dealing with Casing Issues 91
Maintaining Shadow Fields 92
Storing Shadow Fields as Objects 93
Conclusion 94

7. Complex Data, Made Simple 95
Introduction 95
Structure 96
Objectives 96
Arrays and embedded documents 96
A Note on quotes 96
Example Documents 97
Querying arrays 99
Array Order Matters 99
Matching multiple array values 100

Mixed data type arrays *100*
Arrays and query operators *101*
Querying embedded documents 102
Exact matches *102*
Matching inside embedded documents *103*
Array update operators 104
Optional example documents *104*
Adding array items *105*
Appending multiple items *105*
Sorting array items *106*
Removing array items *107*
Conclusion 110

8. The MongoDB Aggregation Framework 111
Introduction 111
Structure 111
Objectives 112
Typical aggregation pipelines 112
Pipeline stages *112*
Building a pipeline *112*
Projecting aggregated fields *113*
Aggregations in MongoDB Compass *114*
MongoDB compass pipeline features *116*
Combining operators *117*
Filtering matching documents *119*
Using stages multiple times *121*
Making large pipelines more readable *122*
Grouping and sorting stages *123*
Complex pipelines 125
Create a complex pipeline *126*
Complex Pipeline Example *127*

Stages of a complex pipeline explained *129*
Stage one *129*
Stage two *129*
Stage three *129*
Stage four *130*
Stage five *130*
Additional uses 131
Case insensitive searching and sorting *131*
Using a pipeline for better searches *132*
Stage one *132*
Stage two *133*
Stage three *133*
Stage four *133*
Stage five *134*
Updating documents *136*
Conclusion 138

9. Planning for Performance – Collections and Indexes 139
Introduction 139
Structure 139
Objectives 140
Indexing collections 140
Basic Indexes *141*
Creating an index *141*
Creating an Index in MongoDB compass *142*
Naming an index *142*
Indexes on Arrays *143*
Compound indexes *143*
Unique indexes *143*
Query plans *144*
Using explain() With an Index *145*

Special index types *147*
Case insensitive indexes *147*
Wildcard indexes *148*
Time to live indexes *149*
Geospatial indexing *150*
Maintaining indexes 150
Hiding Indexes *151*
Deleting indexes *152*
Modifying indexes *152*
Collection settings and types 152
Capped Collections *153*
Time-Series collections *154*
Storing files with GridFS *155*
Document schema validation 156
Basic schema validation *156*
Validation in MongoDB compass *158*
Collection maintenance 159
Collection statistics *159*
Deleting collections *161*
Conclusion 161

10. Getting In and Getting Out – Data Migration 163
Introduction 163
Structure 163
Objectives 164
Importing data 164
Importing via MongoDB Compass *164*
Using database import tools *166*
MongoDB database tools *166*
Using the mongoimport Command 167
Importing Into Non-empty Collections *169*

Importing JSON from an API ... 169
Bulk Inserts ... 172
Exporting data ... 172
Exporting via MongoDB Compass ... 172
Using database export tools ... 173
Using the mongoexport Command ... 173
Using the mongodump Command ... 174
Using the mongorestore command ... 174
Transferring data ... 175
Transferring via the Aggregation Framework ... 175
Archiving documents ... 175
Conclusion ... 176

11. Make It Great – Configuration and Monitoring ... 177
Introduction ... 177
Structure ... 178
Objectives ... 178
MongoDB server operations ... 178
Starting the MongoDB server ... 178
Stopping the MongoDB Server ... 179
MongoDB Server binary ... 179
The importance of ports ... 179
Other MongoDB binaries ... 180
Configuration ... 180
Server binary and data ... 180
MongoDB Data Files ... 181
Command line ... 181
Configuration file ... 182
File Format ... 182
Server defaults ... 182
Common Options ... 183

Externally sourced config 188
Monitoring MongoDB 188
Database statistics 189
The dbStats command 189
The serverStatus command 189
Command line tools 192
The mongotop command 192
The mongostats command 193
Monitoring software 194
Conclusion 195

12. Seamless Scaling – Replication and Sharding 197
Introduction 197
Structure 198
Objectives 198
Reducing risk with replication 198
Replica sets 199
Primaries, secondaries and elections 199
Automatic failover 199
Initiation and configuration 201
Replica set configuration 201
Initiating a replica set 204
Connecting to a replica set 204
The local database 207
The oplog collection 208
Changing a primary to a secondary 209
Member roles and types 210
Priority 210
Voting nodes 211
Hidden nodes 211
Delayed nodes 212

Arbiters 212
Administration 213
Changing an existing replica set 213
Fail-safes 213
Maintenance and disaster recovery 215
Monitoring 216
Scaling with Sharding 216
Sharding data and Shard keys 217
Config Servers and the mongos process 217
Replica Set Considerations 218
Sharding configuration 219
Conclusion 219

13. Being Proactive – Security and Backups 221
Introduction 221
Structure 221
Objectives 222
Authentication – Proving who you are 222
Enabling access control 223
Localhost exception 223
Authorization on Docker 224
Creating Users 224
Logging in With a User 226
Types of authentication 227
Users on different databases 227
Authentication and replica sets 228
Authorization: What you can do 228
Privileges 229
Roles 229
User-defined roles 230
Backup strategies 232

Filesystem Backups 232
MongoDB Database Tools 233
Example backup script 233
MongoDB services 235
Restoring backups 235
Restoring via MongoDB data files 235
MongoDB Database tools 235
Example Restore script 236
Database encryption 237
Network encryption 237
Application-level encryption 237
Conclusion 238

14. Making Stuff – Programming with MongoDB 239
Introduction 239
Structure 239
Objectives 240
Programming with MongoDB 240
Code examples 240
Book GitHub Codespace 240
Python and MongoDB 241
Installing the PyMongo library 241
Connecting to a Database with Python 242
Performing queries with python 243
Query options with python 243
Inserting a document with Python 244
Aggregation with Python 246
Node.JS and MongoDB 247
Installing the MongoDB Driver 247
Connecting to a Database with Node.JS 247
Query Options with Node.JS 249

Inserting a Document with Node.JS 251
Aggregation with Node.JS 252
Using MongoDB with Mongoose 254
PHP and MongoDB 256
Installing the MongoDB Driver and Library 256
Autoloading the PHP MongoDB library 256
Connecting to a database with PHP 257
Query Options with PHP 258
Inserting a document with PHP 259
Aggregation with PHP 260
Using MongoDB with Laravel 262
Other language examples 263
Go 263
C# 265
Java 266
Kotlin 267
Rust 268
C++ 268
Ruby 270
Swift 270
Perl 271
Scala 272
Bash 272
Conclusion 273

15. Tools for Success – MongoDB Shell and Compass UI 275
Introduction 275
Structure 275
Objectives 276
MongoDB Shell 276
Configuration 276
Editor mode 276

Node.JS Scripting 277
Snippets 282
Visual Studio Code 283
MongoDB extension 283
MongoDB Playgrounds 284
MongoDB compass 285
Advanced connection options 285
Creating Collections 287
My Queries 287
Exporting Queries 288
Aggregation Framework and Pipelines 288
Schema 290
Explain plan 291
Performance monitoring 292
Conclusion 293

16. Cloud Services – MongoDB Atlas 295
Introduction 295
Structure 295
Objectives 296
Database Services 296
Multi-Cloud Database Services 296
Clusters 296
Serverless 298
Viewing and Editing Data 298
Users 299
Network Access 299
Connecting to Atlas Cluster 301
Data API 301
Backups 303
Cloud Tools 304

Search 304
Triggers 306
Device Sync 307
Data Lake 307
Atlas CLI 308
Charts 308
Create Charts 309
Sharing and Embedding 312
Conclusion 313

17. MongoDB Atlas – Application Services 315
Introduction 315
Structure 315
Objectives 316
Atlas Database and App 316
Create Atlas App 318
Database User Access 320
Anonymous Realm Users 320
User Create Function 321
React App 325
Create React App 325
Dependencies 326
Using the Realm Web SDK 326
Coding the App 327
Core Components 327
Realm Context 332
Context Hooks 334
User Context 335
useUser Hook 337
useAggregate Hook 340
Question Context 342

Displaying Questions *345*
Question Component 346
Question Tabs Component 349
Question Navigation Component *356*
Wrap Up *358*
Conclusion 358

18. Jobseeker – Interview Prep 359
Introduction 359
Structure 360
Objectives 360
MongoDB Questions and Response 360
How to use these questions *361*
MongoQuest *361*
Questions round 1 361
Questions round 2 362
Questions round 3 364
Questions round 4 365
Questions round 5 367
Questions round 6 368
Questions round 7 370
Questions round 8 371
Questions round 9 372
Questions round 10 374
Conclusion 375

19. Conclusion 377
Introduction 377
Structure 377
Objectives 378
Change Streams 378

Subscribing to Database Changes *378*
Triggering Actions with Change Streams *379*
Running Change Streams *381*
Transactions 383
Transactions in mongosh *383*
Transactions in Code *384*
Storing Files with GridFS 386
Using mongofiles *387*
Using Code *388*
SQL to MongoDB 389
Conclusion 390

Index 391-399

Chapter 1

Why MongoDB?

"There is a great deal of difference between the eager man who wants to read a book, and the tired man who wants a book to read."

-G. K. Chesterton

Introduction

Many people love to cook. But do not worry; this is not a book about cooking. We are bringing up food to whet your appetite a little and maybe even illustrate how you are probably using a lot of the concepts we will learn about here, in your everyday lives.

If you are picking up this book, you probably have some idea of what MongoDB is or you might have heard that learning it will help you along in your career. Maybe you just really want to know more: Why should I learn MongoDB; you might be asking?

MongoDB is a powerful, fast, and modern database that was designed for modern applications. From its flexible Document Model (for data), JavaScript based query language, blazing speed, advanced replication and scaling, simple redundancy, advanced aggregation framework, and hooks into the wider MongoDB Atlas Cloud ecosystem, it just might be a clear choice for your next project (or even an existing one)!

Through real life examples and analogies (like cooking), we are going to learn all about MongoDB and its concepts, its associated technologies and how those concepts might apply to the next step in your career.

Hopefully, you are hungry to learn some more. Let us get started.

Structure

In this chapter, we will discuss the following topics:

- Recipes as data
- The history of data
    - Databases of Data
    - Computer Databases
    - Relational Databases
    - Bringing Separate Data Together
- Data that goes together, can live together
- Thinking of data differently
- This is why MongoDB…

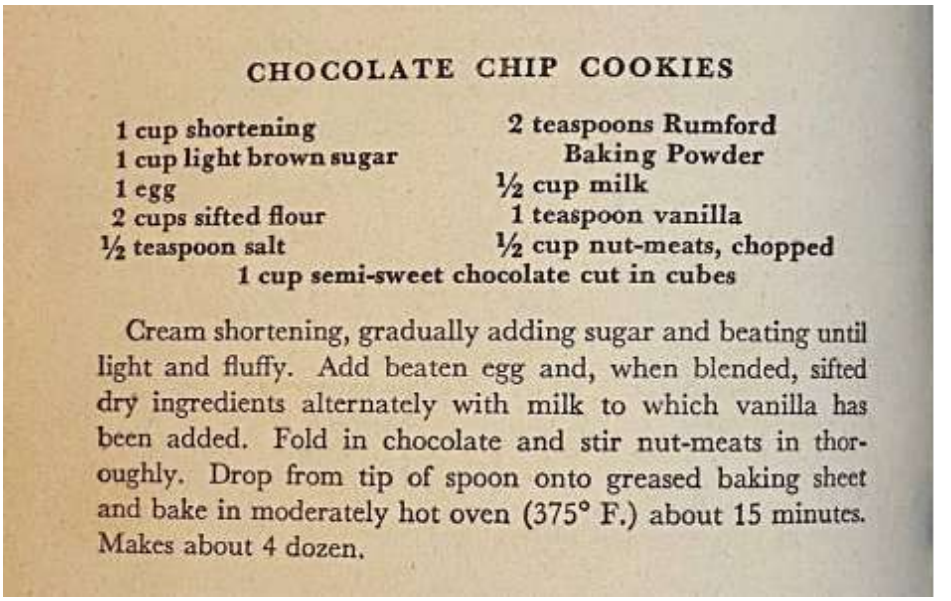
Objectives

By the end of this chapter, you will have a basic idea of the history of data and databases, how different types of databases work and how MongoDB differs with a newer, modern way to store and interact with data.

Recipes as Data

We all need food and *good* food that is cooked well, is appreciated throughout the world and across cultures. Food (and cooking it) can sometimes be subjective and complex, yet somehow simple (at its best), at the same time.

What your favourite meals do have in common is some form of structure: there are important details about what ingredients you need (their measurements, units, and so on). There are precise steps to follow, specific temperatures as well as cooking techniques needed, to successfully make the dish. *Figure 1.1* is an example of an old recipe:



CHOCOLATE CHIP COOKIES

1 cup shortening
1 cup light brown sugar
1 egg
2 cups sifted flour
½ teaspoon salt
2 teaspoons Rumford Baking Powder
½ cup milk
1 teaspoon vanilla
½ cup nut-meats, chopped
1 cup semi-sweet chocolate cut in cubes

Cream shortening, gradually adding sugar and beating until light and fluffy. Add beaten egg and, when blended, sifted dry ingredients alternately with milk to which vanilla has been added. Fold in chocolate and stir nut-meats in thoroughly. Drop from tip of spoon onto greased baking sheet and bake in moderately hot oven (375° F.) about 15 minutes. Makes about 4 dozen.

Figure 1.1: *Century Old Chocolate Chip Cookie Recipe (Public Domain)*

In essence, these instructions are an orderly set of data which we simply call a "recipe." We compile it into a document, which in turn is sometimes gathered into a collection of documents. In simple English, it is called a "Cookbook" or a "Recipe Book".

Switching to a more modern example; what if you wanted a recipe for dinner? You have about half an hour and all you have in your refrigerator, is a chicken. So, you ask the voice assistant on your phone:

"Find me recipes for chicken that I can cook in 30 minutes or less."

That might seem like a simple statement but (going with our theme here) there is a lot of information packed into this short sentence. Let us break it out a bit, and see how we might turn this spoken request into a database query:

"Find me recipes for chicken"

Starting off, this has the makings of a good query. The user wants us to "find" a particular thing: a "recipe". To make this simple, we will query for recipes literally titled "Chicken". Now, in reality, we would probably do some sort of wildcard search for recipes with the word "chicken" in the title, or more likely also as one of the ingredients.

"… that I can cook in 30 minutes"

This is a little more complicated. Do they mean the actual time to cook the recipe is 30 minutes? Or do they mean cooking time plus the preparation time? Given the context here, we will assume that they mean the *combination* of both. Also, this would mean 30 minutes or less, as in less than or equal to.

Find me a recipe query

Here is how we might construct this as a query in MongoDB:

```
db.recipes.find( { "title" : "Chicken", "total_time" : { $lte: 30 } } );
```

We will gain more information about how querying works and how to compose queries in future chapters. For now, you can see we are looking for a match on the **`title`** field and doing our less than or equal to on a **`total_time`** field. It might look a little funny right now, but the **`$lte`** means "less than or equal to" **`30`** and it is called a Query Operator (they start with a **`$`**). We will learn more about them soon.

Put back into English, it translates to: "Find me recipes where the title is Chicken, and the total time to make it is less than or equal to 30 minutes."

With MongoDB**,** we will search our recipes, which are stored in **Documents** (more on that in *Chapter 4, A Better Way to Store Data – Documents*) within a **Collection** here called "recipes" (more on that in *Chapter 9, Planning for Performance – Collections and Indexes*) by using the **`find`** method. Given this method of storing data, MongoDB is commonly referred to as a **Document Database**.

Before we go much further, let us step back for a moment and delve a little into the history of data and databases.

The history of data

"As a general rule, the most successful man in life is the man who has the best information."

-Benjamin Disraeli

"Information is the oxygen of the modern age. It seeps through the walls topped by barbed wire; it wafts across the electrified borders."

- Ronald Reagan

There are many types of databases, and the concept of a computer database has been around for almost as long as (electronic) computers themselves. However, data is not a "new" or even "modern" concept. Humans may have only had computers for a couple decades, but data (and even forms of databases) have been around for millennia.

Databases of clay

The first computer database is generally credited to Charles Bachman in the 1960s while working for IBM. However, the general concept itself goes back thousands

of years. Data has always been a powerful asset and being able to store it reliably and easily (for later use) has been a major factor in the success of various successful individuals, companies, cities, and empires.

Across time, empires and countries have used data to help them maintain and manage their control from a high level, down to a personal level. Unfortunately, since these records were often stored on various forms of paper, most of them were lost over time.

It is difficult to know how far back the concept of "data" goes, but amazingly, we have actual records of data going back for more than a thousand years, thanks to an ancient form of database technology. By writing on tablets of soft clay (which could then be dried in the sun or hardened in a furnace), ancient peoples across the East were able store all sorts of important (or even trivial) information. *Figure 1.2* is an example of such a clay tablet:

Figure 1.2: *Clay Tablet with Cuneiform Writing*

Archeologists have found thousands and thousands of these tablets buried in the sand all over the Middle East and Asia; apparently, ancient people loved their data! Interestingly, after paleographers were able to decode and translate these ancient tablets (which were mostly written in script called cuneiform that can compose multiple different languages), they found many of these documents contained data we might keep in a database today.

These tablets stored things like the records of a household's property, and how it was to be distributed in the event of the head of the household dying (what we might refer to as a "last will and testament") or tallies of food stores: *"Ikup-pi-Adad has 5 bushels of grain, three oxen and 8 jars of fat."*

There were also tax bills, financial exchanges, letters, contracts, military dispatches, historical documents, poetry, and vast inventories of "store cities," containing anything from food to textiles, precious metals, chariots, and horses which the king, queen and their advisors used to maintain their kingdom with. And of course, they also had recipes. All sorts of recipes have been found; 4,000-year-old instructions pressed into tablets for preparing meals not unlike we might eat today, and even recipes for beer!

Computer Databases

With the advent of modern computers, data once again became something we needed a way to manage, electronically this time.

The earliest databases stored their data in a hierarchical fashion, meaning every piece of data had a "parent", starting with a "root" record. This might be thought of like a "family tree", or a "organizational chart" for a company. It starts with a "root" record that has multiple child nodes, which have one or more nodes themselves and so on, to build out a hierarchy. You can find the data you need by navigating along the tree until you land on the data you need.

For example, if you needed to find the **Database Administrator** at a company, you would start at the top with the CEO (or perhaps Founder) at the "root" and then travel from record to record via the tree: CEO | VP of Engineering | Director of Engineering | Technical Manager | Database Administrator

Each time you want to find the **Database Administrator,** you would need to follow this path of parent/child links (or perhaps another one, depending on how complicated the structure was, this is just a simple example). *Figure 1.3* is an example of a hierarchical database:

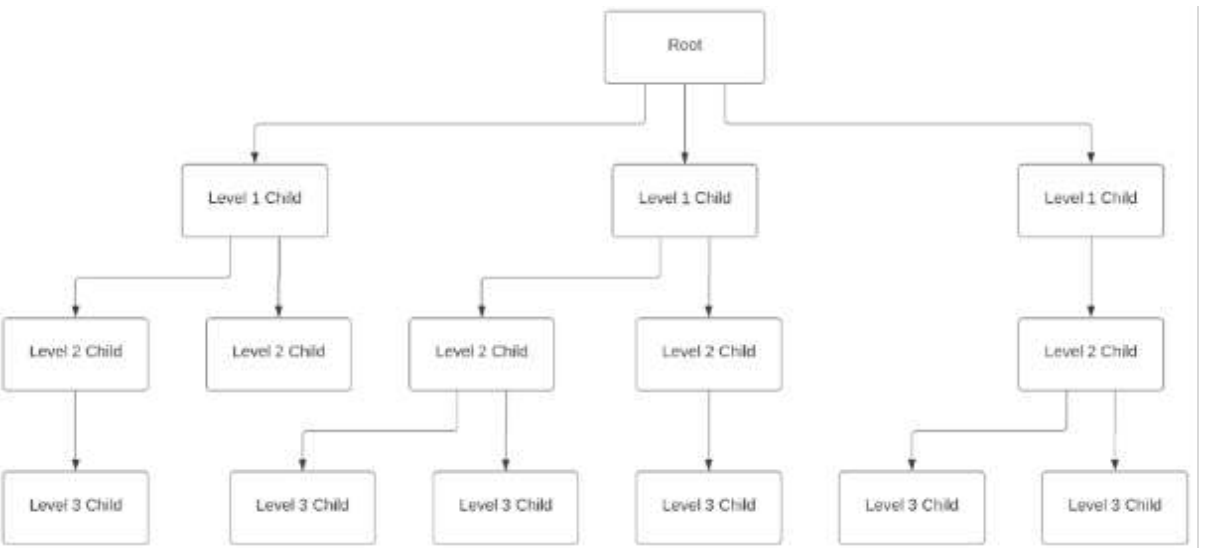


***Figure 1.3**: Example of Hierarchical Database*

Pretty soon, database architects found that this method of storing data had a lot of drawbacks. For one, making a single change to your database structure may require you to change many or all your records and how they were associated to each other.

Additionally, given the hierarchical relationships, it is nearly impossible to avoid "duplicate" data and indexing is also rather difficult, leading to generally bad performance.

Relational databases

To solve many of these issues, the concept of a "relational database" was proposed by E. F. Codd in the 1970s. In short, this type of database is concerned with how the data records relate to each other, instead of a single parent/child type relationship. Generally, these records are stored in tables which have "rows" and "columns" like a spreadsheet. And "tables" which group together records that have the structure or "schema".

Since a Relational Databases' data is no longer hierarchical, there is no simple way to "navigate" to the data you might need. To solve this, a structured language to "query" the data was formed, called (appropriately enough) the **Structured Query Language** (**SQL**).

This querying language was later standardized in the 1980s and is still used in many databases to this day, such as Oracle, Microsoft SQL Server, MySQL, and its variants. In fact, Relational Databases and SQL have gone on to have a major impact on the modern history of computing, and the internet in general.

Relational Databases vs Document Databases

To illustrate the basic differences between **a Relational Database** and a **Document Database**, let us use the example of a sports league, as in football (or soccer, if you will), baseball, basketball, or cricket. A typical sports league will have players as well as teams, stadiums, games (or matches), and a schedule.

In a Relational Database, we could store each of these in their own tables, for example, take the players and teams, as shown in *Table 1.1* and *Table 1.2*:

id	number	first_name	last_name	team_id	goals	updated
1	1	David	De Gea	6	0	2023-02-02
2	10	Mohamed	Salah	56	48	2022-11-11
3	3	Giorgio	Chiellini	4	8	2020-03-06
4	10	Sadio	Mané	46	33	2022-10-01
5	2	Clint	Dempsy	6	57	2018-29-08
6	10	Christian	Pulisic	6	32	2022-12-12

Table 1.1: *Example of Players Table*

Refer to *Table 1.2*:

id	rank	team_name	country_code	stadium_id	updated
1	1	Brazil	BR	45	2023-02-02
2	5	England	NULL	2	2022-11-11
3	3	Argentina	AR	34	2020-03-06
4	20	Italy	IT	22	2022-12-12
5	4	France	FR	12	2018-29-08
6	10	USA	US	NULL	2022-10-01

Table 1.2: *Example of Teams Table*

The "relations" come in when we associate or "link" two distinct types of data. Here, each team will have many players, but a player will only have a single team. *Figure 1.4* shows an example of the relationships between players and teams:

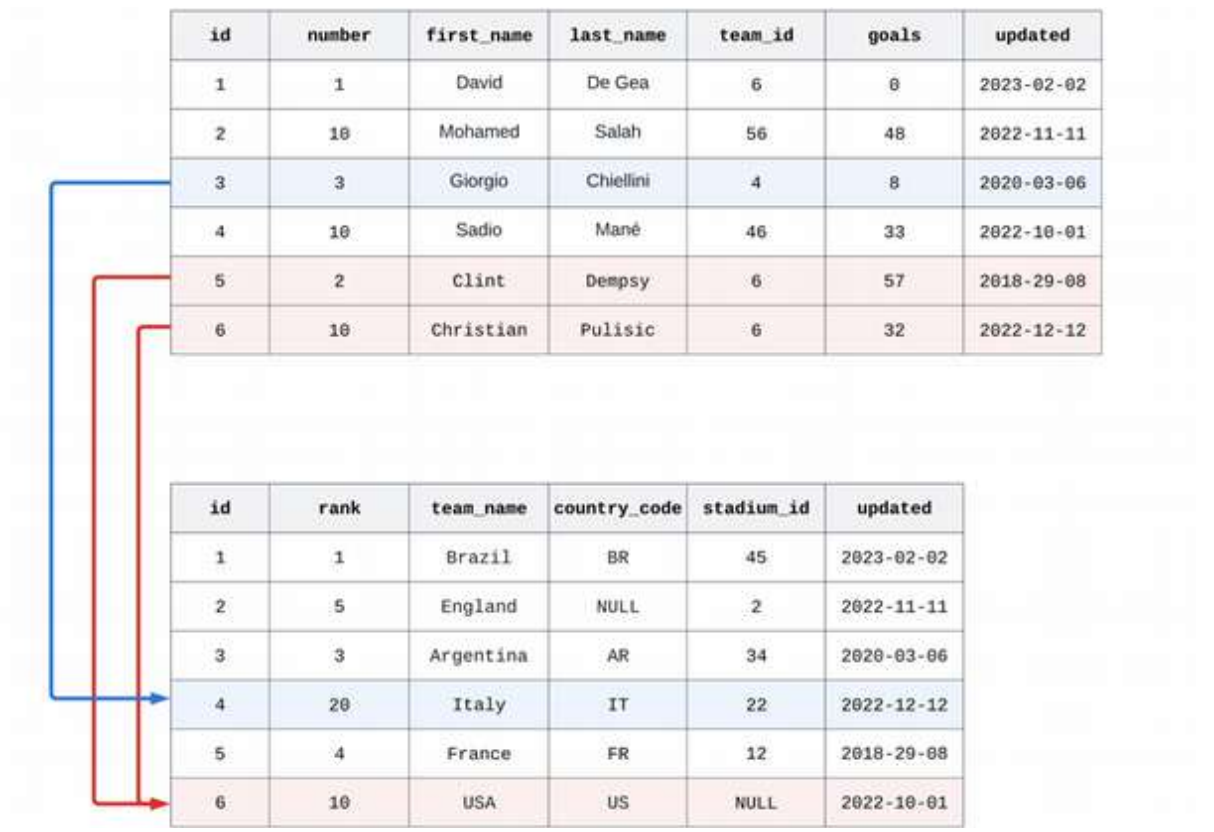


Figure 1.4: *Example of Relationships Between Players and Teams*

There are other relationships too. The teams have games which link two teams together; those games are stored in a schedule and played in different stadiums.

To query this relational data, we will use a SQL query. For example, to get the names of all the players we might write a query like:

```
SELECT first_name, last_name FROM players;
```

If we wanted to get all the pieces of data for a player record (or all the "columns"), we can use the asterisk or * character:

```
SELECT * FROM players WHERE last_name = 'Dempsy';
```

This might return the set of data shown in *Table 1.3*:

id	number	first_name	last_name	team_id	goals	updated
5	2	Clint	Dempsy	6	57	2018-29-08

***Table 1.3**: Example of Query for a Player*

You can do similar queries for each table to get the information you need. All these pieces of data need to be able to have their own relationships which are separate, but also related to each other. An example of table relationships can be seen in *Figure 1.5*:

***Figure 1.5**: Example of Table Relationships*

Bringing separate data together

What about when you need some of the related data returned together? What if we wanted a list of all the players, from both teams that are playing a certain game? In a relational database, we can accomplish these "links" via what is called "join," and so we will add those into our SQL query:

```
SELECT p.number, p.last_name, t.team_name, g.date, s.stadium

FROM players p

JOIN teams t
ON t.id = p.team_id

JOIN stadiums s
ON s.id = t. stadium_id

JOIN games g
ON g.id = t.team_id

WHERE g.game_id = 12
```

A query like this will return a result made up of columns and rows for the players playing in game with the id `12`, as shown in *Table 1.4*:

number	last_name	team_name	date	stadium
3	Maguire	England	2022-11-25	Lusail
10	Pulic	USA	2022-11-25	Lusail
3	Shaw	England	2022-11-25	Lusail
1	Sterling	England	2022-11-25	Lusail
2	Reyna	USA	2022-11-25	Lusail
10	McKennie	USA	2022-11-25	Lusail

***Table 1.4**: Example of Game Players Query Result*

You will notice that there is a lot of repetition of data here, and it is not ordered (there are ways to **`ORDER BY`** in SQL, although not shown here), but it is generally the information we wanted.

There are various ways to "join" tables. Here, we joined the players table to the teams table based on shared ids. We then joined teams and games tables, and finally filtered by "where" the game has an id of `12`.

If you do not understand fully what is going on here, do not worry! This book is not about SQL or relational databases! The point here is to give a general example for means of comparison. If you are already familiar with Relational Databases and SQL, hopefully you are following along… so how is MongoDB different?

Data that goes together, can live together

With a Document Database, we can take these relations and use them to help us organize our data. We will be covering this in greater detail in the rest of the book, but suffice it to say that we can use the principal of "data that goes together, lives together".

Using our sports league example, we could have a couple of Collections in which we can store our teams, players, games, and so on. However, we can also take advantage of the flexibilities that using Documents gives us.

So, we could create a document in our teams Collection that looks like this:

```
{
  "_id": {
    "$oid": "6323a3975c80e800b1c46d07"
  },
  "team": "England",
  "rank": 5,
  "stadium": {
    "name": "Wembley Stadium",
    "capacity": 90000
  },
  "players": [
    {
      "number": 7,
      "last_name": "Saka",
      "first_name": "Bukayo",
      "goals": 12
    },
    {
      "number": 10,
      "last_name": "Kane",
```

```
      "first_name": "Harry",
      "goals": 41
    }
  ],
  "schedule": [
    {
      "type": "game",
      "date": {
        "$date": "2023-05-18T16:00:00Z"
      },
      "opponent": "Faroe Islands",
      "location": "Tórsvøllur",
      "score": {
        "for": 8,
        "against": 1
      }
    },
    {
      "type": "scrimmage",
      "date": {
        "$date": "2023-05-24T16:00:00Z"
      },
      "opponent": "Team B",
      "location": "Old Trafford"
    }
  ]
}
```

If you are feeling overwhelmed, do not worry. We are going to explain each part of this Document below, as well as in the coming chapters. However, the main take away is that we have stored a lot of the data about the team *together* in a single team Document.

We can do this because Documents are a lot more flexible when it comes to their data (and types of data) than tables are. In fact, we can get a lot more complex than this if we want. In this example, we have demonstrated that we can store strings, numbers,

arrays, and objects. If you are familiar with JSON, you might be thinking that this looks a lot like JSON. In fact, you are right. Documents are stored in a binary form of JSON called BSON (binary JSON).

Team document breakdown

To understand things a little bit better, let us break this example Document out field by field:

Team

```
"team": "England",
"rank": 5,
```

Here we have two fields: one is a simple **`string`** containing the team's **`name`**, and the other a **`rank`** field with a **`number`**; pretty much like any other database can do.

Stadium

```
"stadium": {
  "name": "Wembley Stadium",
  "capacity": 90000
},
```

The **`stadium`** field is a little more complex. Here, we have an "object" which is denoted by the open and closing brackets { … }. Inside this **`object`**, we can store pretty much anything we can in our parent Document (more on that later). In fact, this is sometimes referred to as a "Subdocument" or more simply, a Document inside a Document.

For this example, we have a **`string`** field with the stadium's name, and a **`number`** field with the stadium's capacity.

Players

```
"players": [
  {
    "number": 7,
    "last_name": "Saka",
    "first_name": "Bukayo",
    "goals": 12
  },
  {
    "number": 10,
```

```
    "last_name": "Kane",
    "first_name": "Harry",
    "goals": 41
  }
],
```

Now we are really starting to see the concept of data that "goes together, lives together". Instead of storing the players for our team in a separate table (or Document), we are storing the players and their individual information directly within the team Document.

We can do this in an orderly way by using an **`array`**, denoted by the **`[`** and **`]`**. Inside this array, we can store **`numbers`**, **`strings`**, **`objects`**, or even more **`arrays`**. Here, we are storing each player as an **`object`** that lives within the **`players`** array.

Schedule

```
"schedule": [
  {
    "type": "game",
    "date": {
      "$date": "2023-05-18T16:00:00Z"
    },
    "opponent": "Faroe Islands",
    "location": "Tórsvøllur",
    "score": {
      "for": 8,
      "against": 1
    }
  },
  {
    "type": "scrimmage",
    "date": {
      "$date": "2023-05-24T16:00:00Z"
    },
    "opponent": "Team B",
    "location": "Old Trafford"
  }
]
```

Again, we are using an array, but also showing that we can store objects with objects inside them. This is getting a little more complex, so do not worry if you are not quite following yet. We will cover all this in greater detail, in the following chapters.

Thinking of data differently

> **"We're entering a new world in which data may be more important than software."**
>
> *- Tim O'Reilly*

The Team Document we constructed is just an example; it may or may not make sense for your data (you might want to store each Player in their own Document, with an id for the Team for example; it depends on how you plan to use your data). In fact, one of the even more flexible features of this model is that each Document does not need to have the same fields or "schema"; you can mix and match and have more, less, or different fields for each Document in a Collection.

Having this flexibility can have a really big impact on how you store and retrieve your data. For example, with this Document, you may be able to code a "Team Information" page on your web site (that shows team details, stadium information, player rosters, upcoming schedules, and past results) by returning a single Document. That same page may need to make many different database queries (and multiple joins) to return the same data.

This opens a lot of possibilities for storing data differently, more flexibly and more importantly in line with what your application needs. At the same time, it does mean we might need to think a lot differently about how we store our data. We will dig into this more when we cover how to best structure your data in Documents (*Chapter 4, A Better Way to Store Data: Documents*); for now, it might help to try and visualize some data you already know about and how you might store that in a Document (see *Challenge* at the end of this chapter).

This is why MongoDB

With the Document model, you can store your data in ways you might have never imagined, or in ways that you would have had to have totally reorganized (or cobbled together) in code, every time it is queried. With MongoDB, we can often store the data *exactly* how the application needs it.

Beyond this fundamental difference in data architecture, MongoDB is also fast. Depending on your usage, it may be able to handle vastly more queries than a SQL based database. Additionally, it has many unique and powerful ways to query data and perform complex aggregations. When you do need to scale, you can do so easily

with built-in replication and data sharding (sharding is the concept of automatically breaking up your data across databases, more on that in *Chapter 12, Seamless Scaling: Replication and Sharding)*.

Conclusion

In this chapter, we learned about the history of data, general types of databases over time and how MongoDB differs from some of these legacy databases. For most of the rest of this book we will be discussing MongoDB specific topics. However, it is important to understand that MongoDB may not be the solution to every situation or database need. In fact, you might even find you will get the best results by mixing types of databases!

All that said, since this is a book aimed at helping jobseekers in the next chapter, we will discuss some of the jobs and roles that learning MongoDB might help you land, or excel in your current role. After that, we will move on to getting started by setting up MongoDB on your local machine, or on MongoDB Atlas Cloud.

This book will also cover topics such as how to create, read, update, and delete Documents, querying, Document structures and schemas, importing/exporting data, programming with MongoDB, Mongo Shell and Compass UI, MongoDB Atlas, serverless programming with Stitch, career paths, interview prep and more!

Challenge – Storing data in a document

How might you store some data you already know about in a Document? Would you use strings, numbers, arrays, objects? Could you have Documents inside your Document? Take a moment to sketch out how you might take advantage of the ability to use Documents! It is okay if it is not perfect, we will cover Documents in a lot more detail in the coming chapters.

Here are some ideas to get you started:

Your Personal Résumé or CV

Think about how you would store different jobs and experience you have had, along with the associated dates, company names, and so on.

Web Site User Profile

How would you store a user's information in a single Document?

Think about their name, a complete address (not just stored as single string), other contact information, user preferences, and so on.

A Recipe

Take a recipe you know or have and break it up into a document.

Account for all the ingredients, their amounts and units, preparation steps, and so on.

For more thoughts and ideas about this challenge see: **https://learnmongo.com/book/data-in-document/**

If you feel like you have no clue, do not worry … just keep reading! It will become clearer as we go along.

Chapter 2
MongoDB Jobs and Roles

"Life depends on change, and renewal."

— The (Second) Doctor

"We keep moving forward, opening new doors, and doing new things, because we're curious and curiosity keeps leading us down new paths."

— Walt Disney

Introduction

This chapter is for those who are interested in knowing what sort of jobs and careers learning MongoDB might benefit, both from an enrichment point of view and as a requirement. If you already have a position that you are happy with and just wanted to learn MongoDB, feel free to skip to the next chapter; but if this is of interest to you, keep reading!

Structure

We will discuss the following topics:

- Interest in MongoDB
- Jobs and Career Paths
- Example Interview Questions

Objectives

As this book is designed for people at different points in their career, all it requires is curiosity. Jobs and career paths however do require a bit more planning, assessment and sometimes experience. Since you, dear reader, may be at various points in your career, we will try to discuss the topic of jobs and career paths from a rather high level.

Interest in MongoDB

If you are interested in learning more about MongoDB, you are not alone. For the past six years, MongoDB has been the top, or in the top two "most wanted" tech in the database category of the *Stack Overflow* developer survey. This represents the top database technology that developers want to work with, but do not currently work with, or do not yet have the skills to work with. According to the 2022 Stack Overflow survey: *"MongoDB is used by a similar percentage of both Professional Developers and those learning to code and it is the second most popular database for those learning to code (behind MySQL). This makes sense since it supports a large number of languages and application development platforms."*

Figure 2.1 shows the Stack overflow 2022 developer survey graph:

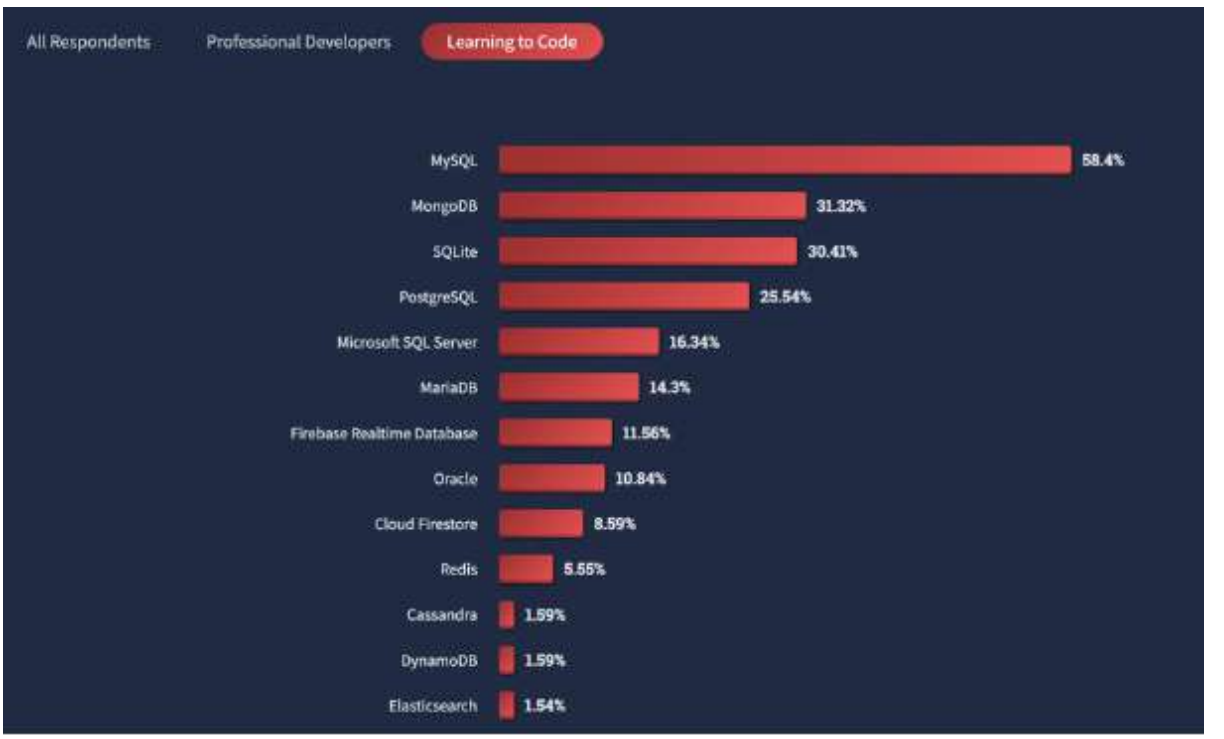


Figure 2.1: *Stack Overflow 2022 Developer Survey*

Additionally, "those learning to code that, currently use MySQL, are more likely to want to start or continue to use MongoDB over any other database" according to the survey.

To top things off, of those who have used MongoDB in "extensive development work in over the past year" MongoDB consistently scores over a 60% "loved" score and is in the top three overall loved databases, meaning not only do many developers want to learn MongoDB, but the majority of developers that use it, love it.

In another 2022 survey of developers, performed by *JetBrains*, who make development tools and IDEs, MongoDB is roughly tied for the third most "used" database by developers in the last 12 months with 27% of developers. It also leads the category for developers that have been working with databases for 1-3 years, with 34% of respondents saying they are using MongoDB.

Lastly, according to *Statista*, MongoDB is the fourth most popular database technology, worldwide in 2022 with over 28% overall.

Given these numbers and trends, learning MongoDB puts you in "good company" and might just help you find a good job at a company!

Jobs and Career Paths

Having knowledge and experience with MongoDB can be helpful in a wide variety of job roles. While most of these roles are, unsurprisingly, in the Technology field, there are some other fields that might also benefit from this skill such as research or academia.

There are three general job types that will most greatly benefit from learning MongoDB and may even be outright required. These would be a Software Developer, often in "Full Stack" development, Database Administrators, and Data Analysts. Each of these roles may have some crossover with various features of MongoDB, but they very likely have an area of focus within those features.

In this chapter, we will break out some of these job roles, discuss some of the key job skills, and also discuss which features you probably will want to make sure you aim to master.

Job Roles

Based on research of current opening, and future trends, there are a number of job roles for someone who knows MongoDB. Following is a list of roles, and a standard description of that role along with how MongoDB might apply to that role.

Full-stack Developer

One of the most common job roles where MongoDB knowledge will come in handy is within Software Development, or more specifically Full-stack development. These jobs where the developer is responsible for the entire "stack" from the frontend to the backend, to the database. Since MongoDB has a modern, easily scaled, and very flexible model for its database, it is often used as the backend database for modern web applications. In fact, in *Chapter 17, MongoDB Atlas – Application Services*, we will do just that, by building a fully functional web app using MongoDB, with full code examples as well as a Git repo you can try.

Full-stack developers who know MongoDB can build scalable, performant, and secure web applications that leverage the benefits of MongoDB's flexible data model. You can read more about Full-stack development and MongoDB here:

https://www.mongodb.com/languages/full-stack-development

If this role interests you, nearly this whole book is designed for you! *Chapter 4, A Better Way to Store Data – Documents*, through *Chapter 9, Planning for Performance – Collections and Indexes*, cover all aspects of how MongoDB works from how it stores data, to querying, to performance and indexing. Additionally, *Chapter 14: Making Stuff – Programming with MongoDB* along with *Chapter 16, Cloud Services – MongoDB Atlas* and *Chapter 17, MongoDB Atlas – Application Services* where we build an app, as we just mentioned.

Data Engineer

In a typical organization, data engineers are responsible for building and maintaining data infrastructure. These may include so called "data pipelines" or processes, along with data warehousing solutions or "data lakes". Sometimes these processes are known as ETL, or Extract, Transform, and Load. MongoDB is often used as part of these data infrastructures given its ability to scale, handle mass loads of reads and writes, and even more so for the flexibility of its data model.

Data engineers with knowledge of MongoDB can help design and build scalable and performant data pipelines. While we will not have a focus on this type of role throughout the book, many of the skills we will cover are extremely important for these types of roles to understand such querying, scaling and using powerful tools like MongoDB's Aggregation Framework.

If this role interests you, make sure to pay close attention to the chapters on the document model, how MongoDB stores data in *Chapter 3, Getting Started* and *Chapter 7, Complex Data, Made Simple*, as well as the chapters which explain how to query MongoDB, that is, *Chapter 5, Let's Do It – Create, Update and Delete Documents*, to *Chapter 8, The MongoDB Aggregation Framework*. Additionally, *Chapter 10: Getting In and Getting Out: Data Migration* will be of particular interest.

Database Administrator

As with other database types, a MongoDB database administrator is responsible for managing the MongoDB database as well as ensuring its availability, scalability, and performance. This role is also responsible for security, backups, and disaster recovery.

Understanding the various ways to scale MongoDB and which will be best for each use case, along with a solid understanding of querying and indexing strategies are all key for this role, as well as knowing the various tools available for monitoring MongoDB.

For this role, *Chapter 11, Make It Great – Configuration and Monitoring*; *Chapter 12, Seamless Scaling – Replication and Sharding*, and *Chapter 13, Being Proactive – Security and Backups* will be of particular interest.

DevOps Engineer

In many modern organizations, there is a separate role, known as a *DevOps* engineer, or *Site Reliability* engineer. This role is responsible for the deployment and management of applications in a production environment. Given that many modern applications and microservices use MongoDB, understanding how to ensure highly available and scalable MongoDB deployments may be very important to these types of roles.

Business Intelligence Analyst

A business intelligence analyst is responsible for analyzing data to support business decisions. MongoDB's flexible data model can help analysts work with complex data structures and perform complex queries to extract insights.

Various tools, such as Microsoft Power BI, or MongoDB's own Atlas tools such as Charts support MongoDB in many powerful ways. Any BI analysts with expertise in MongoDB can greatly help their organizations make the best, data-driven, decisions. For this role, an understanding of querying and the MongoDB Aggregation framework will be a big boost.

Data Scientist

Using similar skills as a BI analyst, data scientists are responsible for using data to derive insights and inform decisions. These decisions may go beyond business into the realms of science such as observational or experimental data. Since MongoDB can be used to store and process large amounts of data, data scientists with MongoDB expertise can leverage its features to analyze complex data sets in robust ways.

Technical Consultant

A technical consultant provides technical guidance and support to customers who use MongoDB. Technical consultants with MongoDB expertise can help customers optimize their MongoDB deployments, troubleshoot issues, and design solutions that meet their specific needs.

Having a well-rounded knowledge of MongoDB will be critical in this role, encompassing aspects of the various roles we have already mentioned, as well essentially the rest of the topics in this book. If you are looking to grow in your role as a technical consultant or move into this role, pay close attention to each of the topics in this book and think about how they might add value for a potential client.

Technical Writer

Technical writers are responsible for creating documentation for software products. While MongoDB's data model and query language is very flexible, it can be challenging to understand for developers, at least at first. Technical writers with expertise in MongoDB can help create clear and concise documentation that helps users get the most out of MongoDB.

Future MongoDB Jobs

According to many analysts, as more organizations adopt modern web applications and microservices architecture, it is likely that the demand for MongoDB skills will continue to grow. This could lead to additional job opportunities and career paths for professionals with MongoDB expertise.

Knowledge of MongoDB can help in a variety of career paths, from database administration to full-stack development to data science, there are many roles that involve working with MongoDB in some capacity.

Example Interview Questions

Towards the end of this book, in *Chapter 18: Jobseeker – Interview Prep*, we will present dozens of interview type questions, and possible responses, that can help you prepare for an interview and help test your comprehension of the contents of this book in general.

Here we have presented a couple of these questions, to give you both a flavor of some of the topics we will follow in this book, as well as let you contemplate how you might answer these questions with your current knowledge.

Questions

1. Can you explain the difference between a document-oriented database and a relational database?
2. How do you handle schema design with MongoDB?
3. What is the difference between a cursor and a result set?
4. How do you optimize MongoDB performance?
5. What is a MongoDB Replica Set?
6. How do you monitor MongoDB performance?
7. How do you backup and restore MongoDB data?
8. Can you explain the concept of document validation in MongoDB?
9. How would you handle failover in MongoDB?

Conclusion

Thus far, we have introduced the general concept of MongoDB, and how it differs from more traditional, or some might say legacy databases as well as the many existing and future job opportunities that knowledge of MongoDB may offer you.

In the next chapter, we will discuss how to get up and running with MongoDB, by installing and setting up your own MongoDB database. Let us start on our journey.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 3
Getting Started

"The beginning is the most important part of the work."

— Plato, The New Republic

"The secret to getting ahead is getting started."

— Mark Twain

Introduction

In this chapter, we will cover installing MongoDB Server on various Operating Systems as well as Docker and MongoDB's Cloud service Atlas. Additionally, we will walk through installing the MongoDB Shell, Tools, and the official GUI MongoDB Compass.

Structure

In this chapter, we will discuss the following topics:

- Installing MongoDB
- Installing MongoDB on Windows
    - Installing MongoDB Server, Compass and Tools

    - Connecting to MongoDB Server
- Installing MongoDB on macOS
    - Installing MongoDB Server, Shell and Tools
    - Running MongoDB Server
    - Connecting to MongoDB Server via Compass
- Installing MongoDB on Docker
    - Running MongoDB Server on Demand
    - Connecting to MongoDB on Docker
    - Running MongoDB Server via Docker Compose
- Setting up MongoDB on MongoDB Atlas Cloud

Objectives

By the end of this chapter, you should be comfortable with installing and setting up MongoDB Server and Tools on your local system, Docker or via MongoDB Atlas Cloud.

Prerequisites

Before we install MongoDB Server and its Shell, you will want to clone this book's git repository to your computer. Although it is not required, it will make a lot of things easier while reading this book. There is a folder for each chapter with a README file and for some chapters, this folder will also contain example files and odds and ends.

If you already have `git` installed, simply clone the repo to a suitable location on your system.

If you do not have Git setup, you can head on over to the official GitHub site and follow the instructions on how to clone the repository.

Installing MongoDB

Once you have the repository downloaded, we can move on to installing the MongoDB software.

At the time of writing this chapter, the major version of MongoDB Server is 6.x and thus these instructions will reflect that version. It is possible these steps will change with future versions of the software, so if you are stuck, make sure to check out the

MongoDB website. For the most part, these instructions are written such that the version should not matter.

For clarity, this chapter will be broken up into sections by Operating System. Feel free to skip to the section about your Operating System of choice. If you are curious about (or need to know) how to install and setup MongoDB on each type of Operating System go ahead and read each section!

Note: This chapter is designed to help you set up and run with MongoDB Server for purposes of learning the software. These instructions are not meant to be a guide for a "production" MongoDB environment! In later chapters, we will discuss important additional steps and configurations you will want to make for production use.

Installing MongoDB on Windows

If you are running the Windows Operating System, the easiest way to get MongoDB installed is to download the free Community Server `.msi` file from the MongoDB website:

https://www.mongodb.com/try/download/community

The website design will likely change over time, but you should be able to find this on **https://mongodb.com** under **Products** | **Community Edition** | **Community Server**, or simply use a search engine and search for *MongoDB Community Server*.

You should see a page that looks something like *Figure 3.1*:

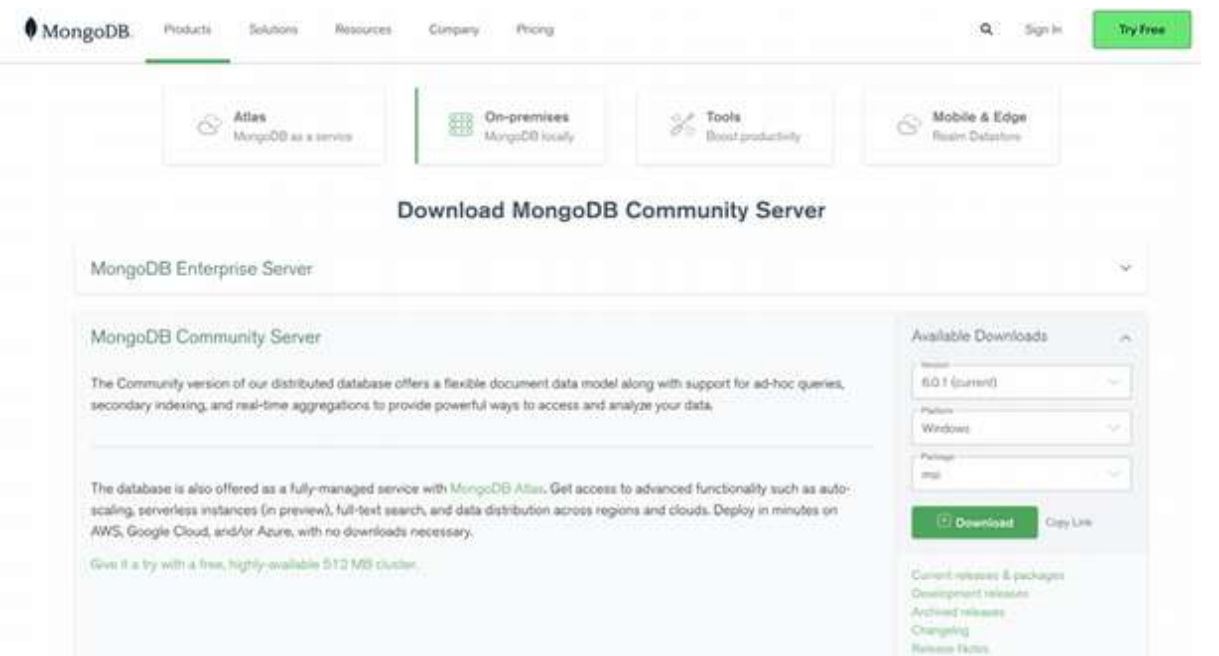


Figure 3.1: *Download from MongoDB Website*

Installing MongoDB server, compass, and tools

Choose the **`.msi`** file, download it and open the file. This will begin the MongoDB Server setup wizard. We will opt to make it a service, and install MongoDB Compass, which is the default GUI for MongoDB. Follow the given steps:

1. When you open the install wizard, it will look something like *Figure 3.2*:

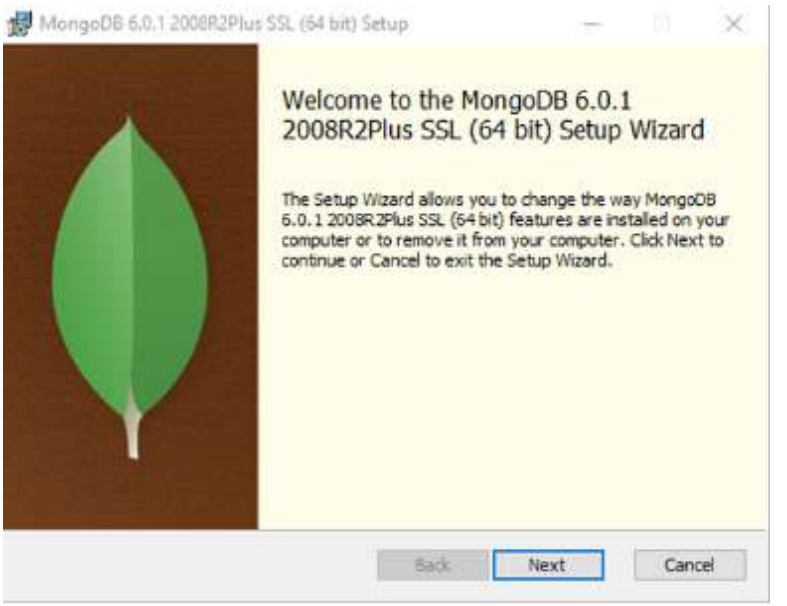


Figure 3.2: *MongoDB Install Wizard*

2. Choose the **Complete** setup type, as shown in *Figure 3.3*:

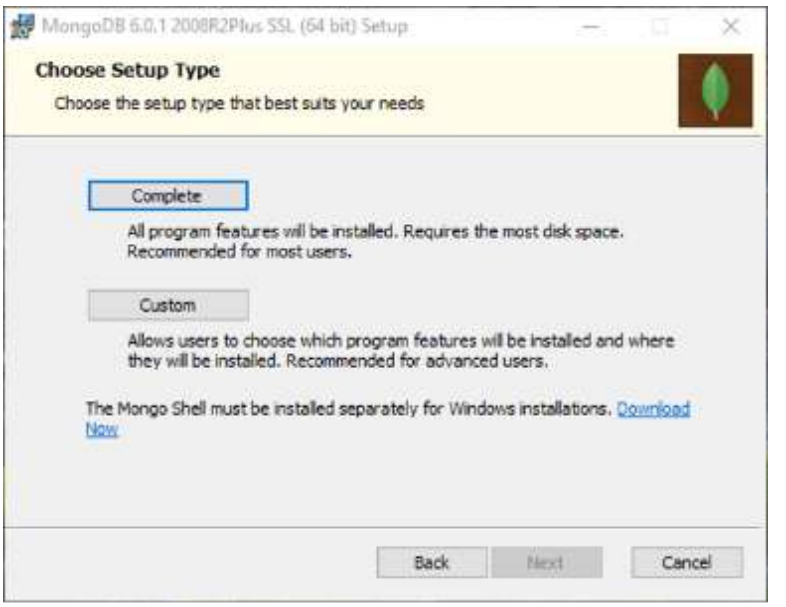


Figure 3.3: *Choose Setup Type*

3. Continue with the default options to **Install MongoDB as a Service**. This will make sure MongoDB automatically starts up for you, so that you do not need to start it every time you need to use it. Refer to *Figure 3.4*:

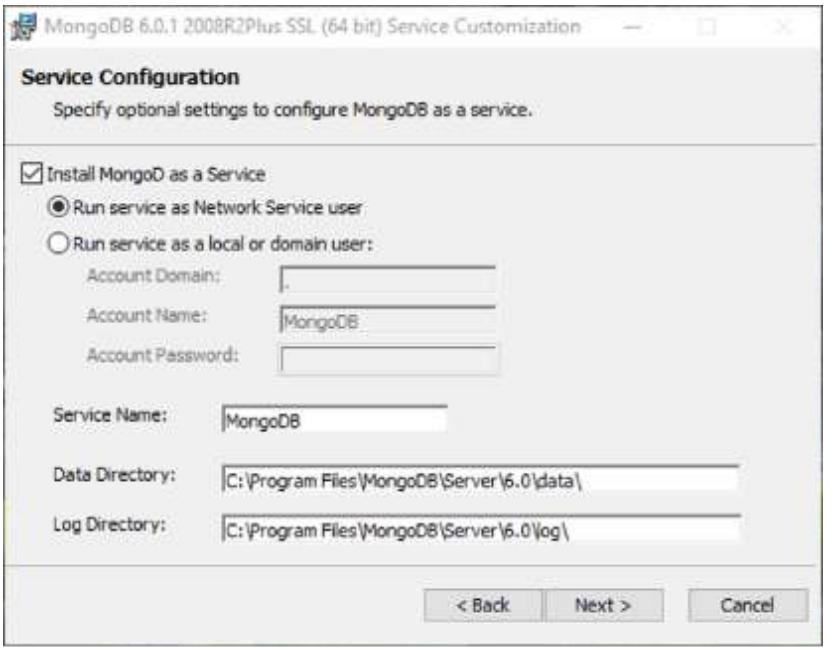


Figure 3.4: *Install as a Service*

4. Lastly install MongoDB Compass. It is the official GUI for MongoDB. Refer to *Figure 3.5*:

Figure 3.5: *Install MongoDB Compass*

5. When the installation completes, you may need to reboot your machine. After that, you will want to make sure to download the Database Tools from the same website (choose the **`.msi`** file option):

   https://www.mongodb.com/try/download/database-tools

6. Run the .msi file and choose the defaults to install the Tools.

Connecting to MongoDB Server

By default, MongoDB Server will startup automatically and run via port `27017` (you may need to reboot, but the Service should start automatically). We can test connecting to it by using MongoDB Compass. Here is what MongoDB Compass will look like (by default) when opened; follow the given steps.

1. Press the green **Connect** button to test, as shown in *Figure 3.6*:

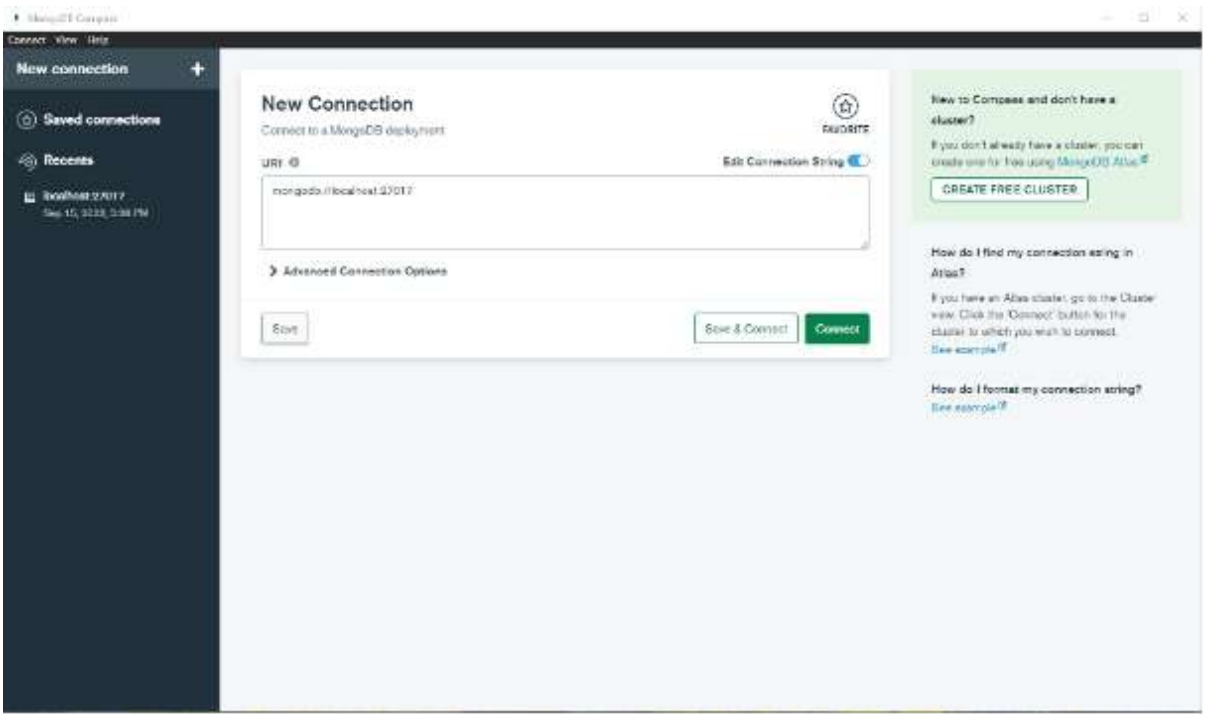


Figure 3.6: *Test Connecting in MongoDB Compass*

2. After you connect, you should see **`>_MONGOSH`** at the bottom left. If you click on that, it will open a Mongo Shell prompt, which we will use at different times in this book. In *Figure 3.7*, we ran the shell command **`show dbs;`** from MongoDB Compass:

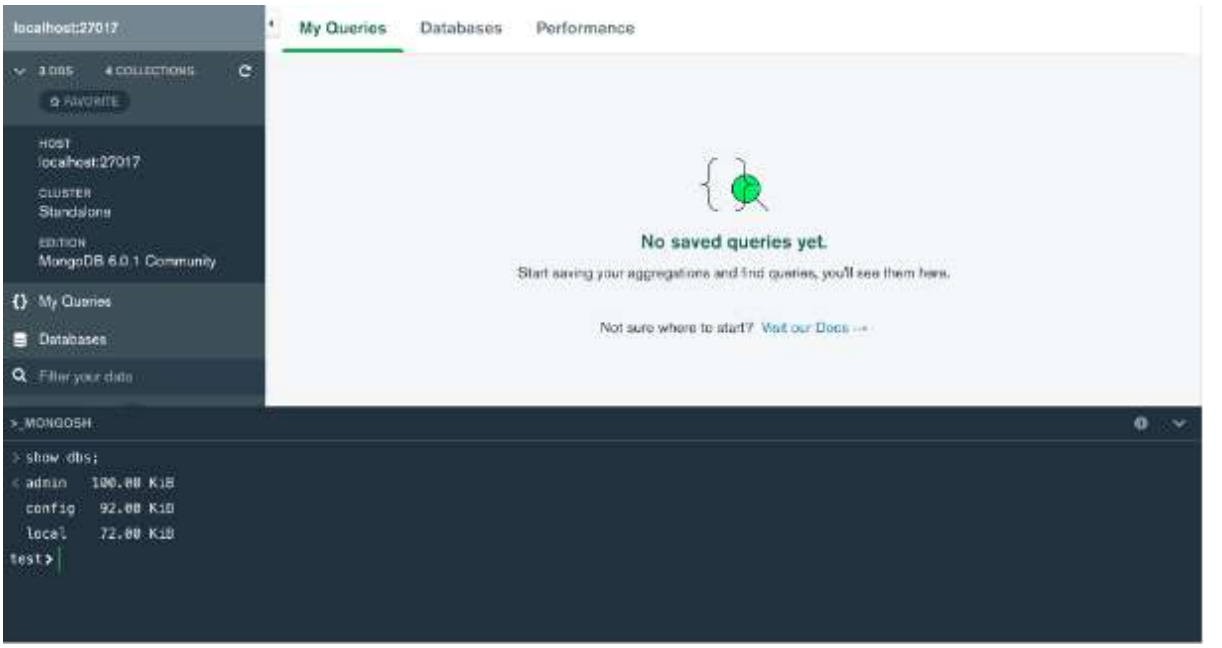


Figure 3.7: *Using the MongoDB Shell from Compass*

3. You can also download and then run the **`mongosh.exe`** program directly, instead of MongoDB Compass, if you prefer the command line. See the instructions on the MongoDB website.

Installing MongoDB on macOS

For macOS, the recommended install method is using the Homebrew "brew" package manager. If you do not already have Homebrew installed, refer to **https://brew.sh/** to set it up. It is free and easy to use!

Note: You will need the Xcode Command-Line Tools to install Homebrew. You can do so by running the following command from the Terminal (macOS command line):

```
xcode-select –install
```

After making sure that it is set up, you should have access to the **`brew`** command line program.

Installing MongoDB Server, Shell and Tools

Using **`brew`**, you can install MongoDB Community Server by using these commands via your terminal:

```
brew tap mongodb/brew
```

This will add the MongoDB "tap" and you will need to update Homebrew after you do this:

```
brew update
```

Now you can install MongoDB Community Edition using:

```
brew install mongodb-community@6.0
```

As noted at the beginning of this chapter, we will be using version 6.x of MongoDB and by using the `@6.0`, Homebrew will automatically install the latest 6.x version of the MongoDB Server as well as the Database Tools.

The default installation location depends on your Apple hardware:

Intel Processor

- **Data Directory**: `/usr/local/var/mongodb`
- **Configuration File**: `/usr/local/etc/mongod.conf`

Apple Silicon

- **Data Directory**: `/opt/homebrew/var/mongodb`
- **Configuration File**: `/opt/homebrew/etc/mongod.conf`

This will install the **`mongod`** binary (the MongoDB Server) as well as **`mongosh`** binary (the MongoDB Shell), the Database Tools and MongoDB Compass GUI.

Running MongoDB Server

To start or stop the MongoDB Server (again from the command line), run the following:

```
brew services start mongodb-community@6.0
```

If you need to stop it, you can use this similar command:

```
brew services stop mongodb-community@6.0
```

At this point, you do not need to run any other command or do any other setup. You should be good to go! If you want to "test" your setup, you can confirm MongoDB is running as a service by running this command:

```
brew services list
```

To connect to your local MongoDB Server, run the MongoDB Shell from your terminal:

```
mongosh
```

It should be noted that macOS may prevent **mongosh** from opening after installation. If you receive a security error when starting **mongosh** indicating that the developer could not be identified or verified, do the following to grant **mongosh** access to run:

1. Open **System Preferences**.
2. Select the **Security and Privacy** pane.
3. Under the **General** tab, click the button to the right of the message about **mongosh**, labelled either **Open Anyway** or **Allow Anyway** depending on your version of macOS.

Running **mongosh** should automatically connect to your local MongoDB Server. You should get some output about your server and then get a prompt that looks something like:

```
test>
```

Using this prompt, you can use the command to list the databases to see some output:

```
show dbs;
```

This should output something like:

```
test> show dbs;

admin   40.00 KiB
config  12.00 KiB
local   40.00 KiB
test>
```

To exit, just type the word **exit** and press enter.

Connecting to MongoDB Server via Compass

You can also choose to install MongoDB Compass, the official GUI for MongoDB, by either downloading it from the MongoDB Website or via Homebrew:

```
brew install --cask mongodb-compass
```

By default, MongoDB Server will run via port `27017`, so we can test connecting to it by using the default connection string in MongoDB Compass. Here is what MongoDB Compass will look like (by default) when opened.

1. Press the green **Connect** button to test, as shown in *Figure 3.8*:

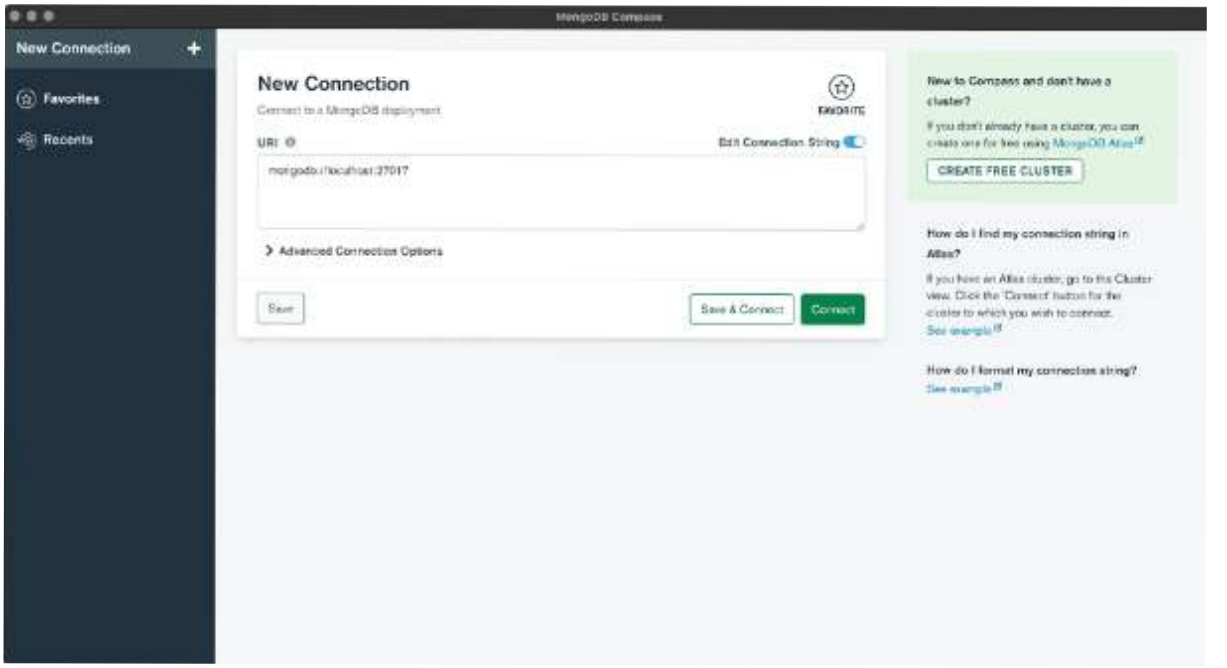


***Figure 3.8**: Test Connecting in MongoDB Compass*

After you connect, you should see **>_MONGOSH** at the bottom left. If you click on that, it will open a Mongo Shell prompt, which is the same as using **mongosh** from the terminal.

Installing MongoDB on Docker

If you would like to use Docker, that is also an option! For our purposes, we will install MongoDB on Docker for "local" usage. Again, as discussed earlier, this should not be used for production, but it can get you started.

We will not discuss how to install Docker, nor target for a particular OS here (or give instructions about that) since the whole idea of Docker is to containerize things. See one of the other install methods if you are not used to Docker.

Running MongoDB server on demand

The simplest way to run MongoDB Server is via a basic container:

```
docker run --name mongodb-server -d -p 27017:27017 mongo:6.0
```

This will start up a container called **mongodb-server** as a "background process", using **-d** which is recommended. It will also output a container id that will look something like:

```
a67ecd3ab80c48625a81a4484b975d4c7947f6442aa1a9e509d134670a22eb9f
```

By default, MongoDB Server runs on port `27017` and we are exposing that port here as part of the command. Lastly, we are targeting version **`6.0`** here, which is not required (you can just use **`mongo`** with no **`:6.0`** if you want). However, this will make sure you are using the same version as assumed in this book.

Persisting database data files

Since the data files for the server will be in the container, *they will be destroyed if the container is deleted*. If you want to make sure they persist, you will need to map the files to your local machine. You can do so by adding an additional option (replace **`LOCAL_DIR`**) with a location on your system:

```
docker run --name mongodb-server -d -p 27017:27017 -v LOCAL_DIR:/data/db
mongo:6.0
```

Now any data files will be saved in whatever local location you chose. Refer to the Docker documentation for more on things, such as how you can change the network settings or set environment variables.

Connecting to MongoDB on Docker

You can now connect to MongoDB via a couple different methods, as discussed.

Connect via the MongoDB Shell mongosh

The default container will have both the MongoDB Server as well as the MongoDB Shell installed, so you should be able to connect to your container and run:

```
mongosh
```

To make things even simpler you can run **`mongosh`** directly:

```
docker exec -it [[YOUR CONTAINER ID]] mongosh
```

This should automatically connect to your local MongoDB Server, output some information about your server and then show you a prompt that looks something like:

```
test>
```

Using that prompt, you can use the command to list the databases to see some output:

```
show dbs;
```

This should output something like:

```
test> show dbs;
```

```
admin   40.00 KiB
config  12.00 KiB
local   40.00 KiB

test>
```

To exit, just type the word **`exit`** and press enter.

Connect via MongoDB Compass

Optionally, you can also install MongoDB Compass. If you have not already done so, see directions in this book for your Operating System or go to:

https://www.mongodb.com/docs/compass/current/install/

Since we have exposed the `27017` port from docker, we can connect using the default connection string:

```
mongodb://localhost:27017
```

This string will most likely already be pre-typed for you in MongoDB Compass and you can connect here, as shown in *Figure 3.9*:

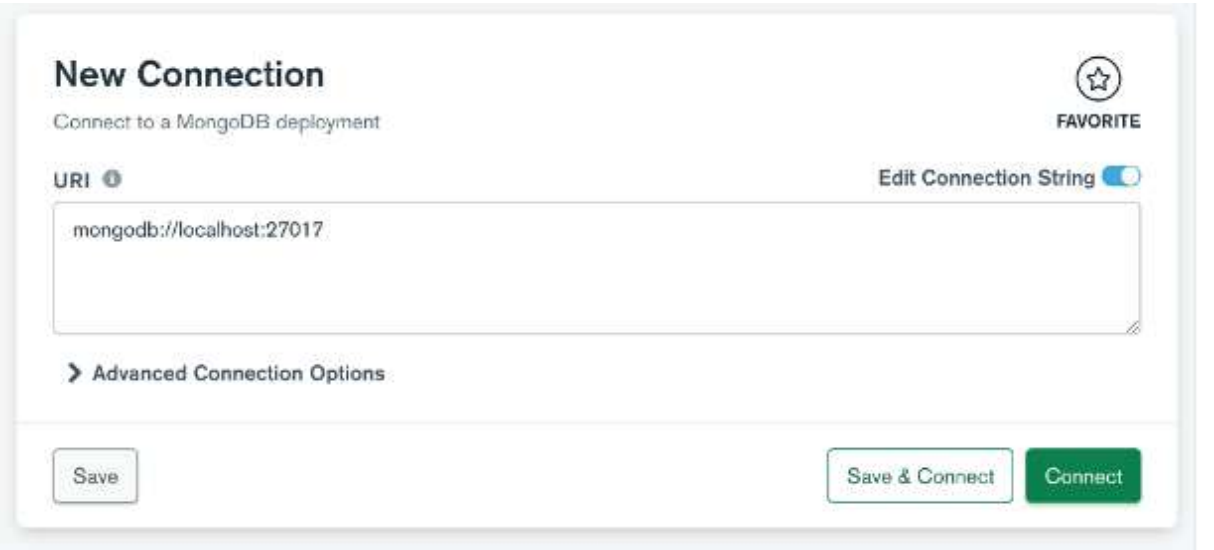


Figure 3.9: *MongoDB Compass with the Default Connection String*

Running MongoDB Server via Docker Compose

You can also run MongoDB via Docker Compose; we will not go in-depth here, but an example Docker Compose file **`docker-compose.yaml`** might look as follows:

```
version: '3'
services:
  mongodb:
    image: mongo:6.0
    environment:
      - MONGO_INITDB_ROOT_USERNAME=root
      - MONGO_INITDB_ROOT_PASSWORD=password123
    ports:
      - 27017: 27017
    volumes:
      - ./data/db:/data/db
```

This will install MongoDB at version `6.0`, setup a username/password (which you will then need to connect). It will also map a volume for the database data files to a folder name **`db`** inside a folder named `data`. You can find a copy of this compose file inside the `docker` folder in the book's git repo.

Place the **`docker-compose.yaml`** file in a folder on your system. Inside that folder create a folder named **`data`** and inside that, create a folder named **`db`**. Then run:

```
docker-compose up -d
```

This will create a MongoDB container for you and run it in "background process" again, using **`-d`** which is recommended. You can now use this connection string to connect:

```
mongodb://root:password123@localhost:27017/
```

See the Docker documentation on their website for more options if you want to use Docker Compose.

Setting up MongoDB on MongoDB Atlas Cloud

Another option is to setup a free "cluster" on MongoDB Atlas (MongoDB's fully managed cloud solution). Here, we will show some basic instructions on how to create a free cluster and connect to it. It should be noted that some of the things we will cover in this book, cannot be done if you opt to only create a free cluster on MongoDB Atlas. It is, however, a nice option for most of what we will do, as it is fully managed.

Note: At the time of writing, a free cluster is only available for MongoDB 5.0 – no need to worry, this will not matter for the purposes of this book as there are no major differences between 6.0 and 5.0 of MongoDB Server.

Follow the given steps:

1. Go to **https://cloud.mongodb.com/** and sign up if necessary. Choose to create/deploy a cloud database. Pick the Free tier, as shown in *Figure 3.10:*

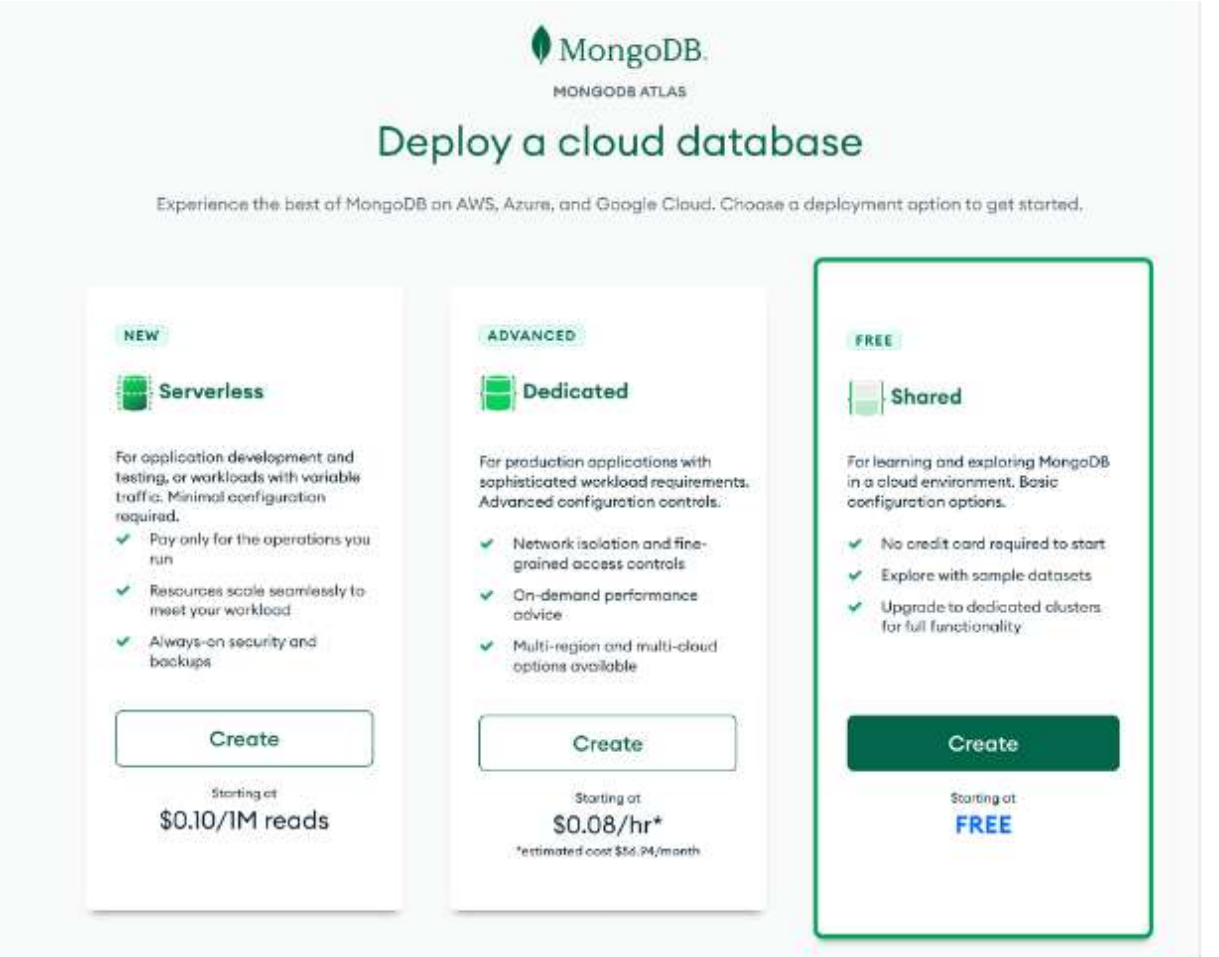


Figure 3.10: *MongoDB Atlas Free Tier*

2. Next, you can choose whatever option you would like for the cloud provider; the defaults are fine. Again, this is free. Here, we have selected AWS located Northern Virginia, USA in *Figure 3.11*:

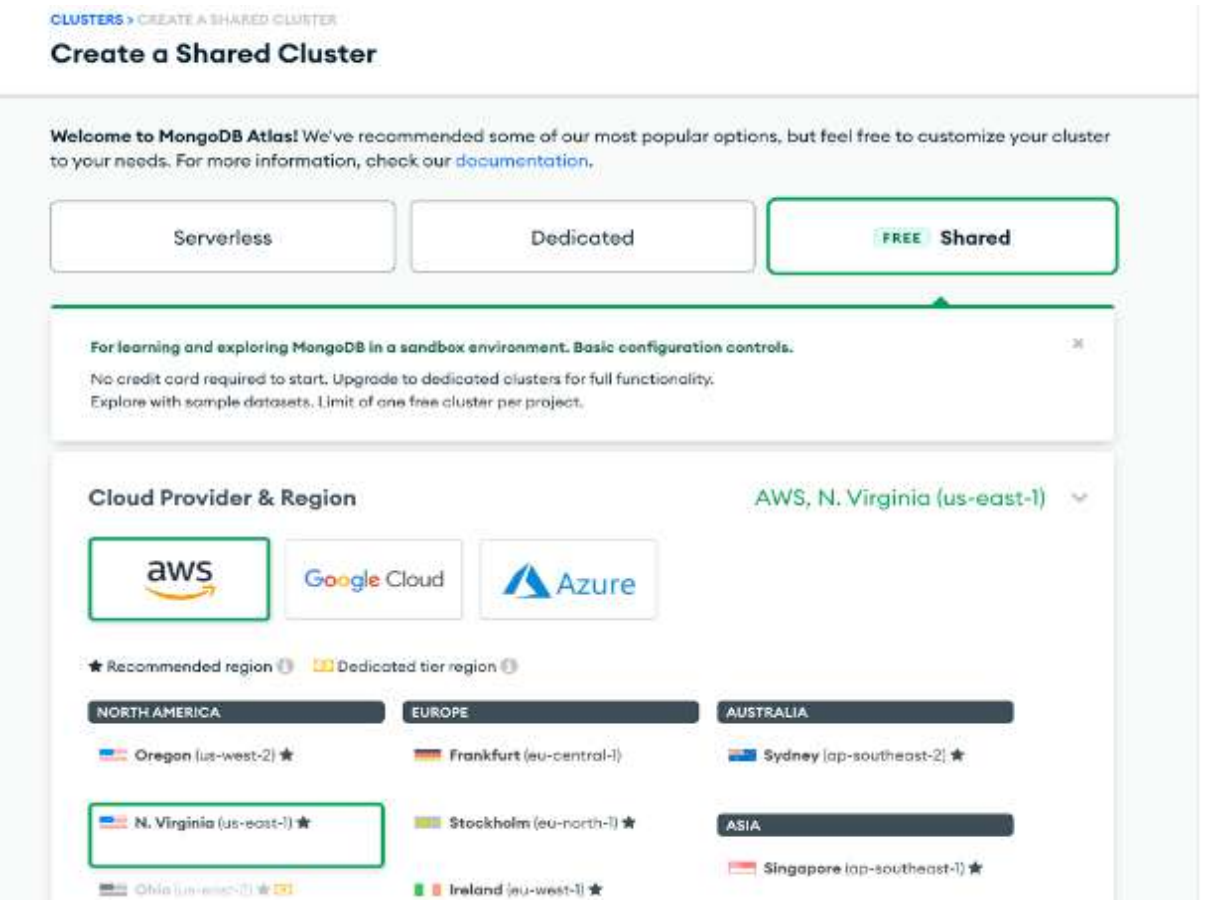


Figure 3.11: *Using AWS as the Provider*

3. Lastly, name your cluster (pick something fun!) and press **Create Cluster**. Here we named it **`jobseekers-book`**, as shown in *Figure 3.12*:

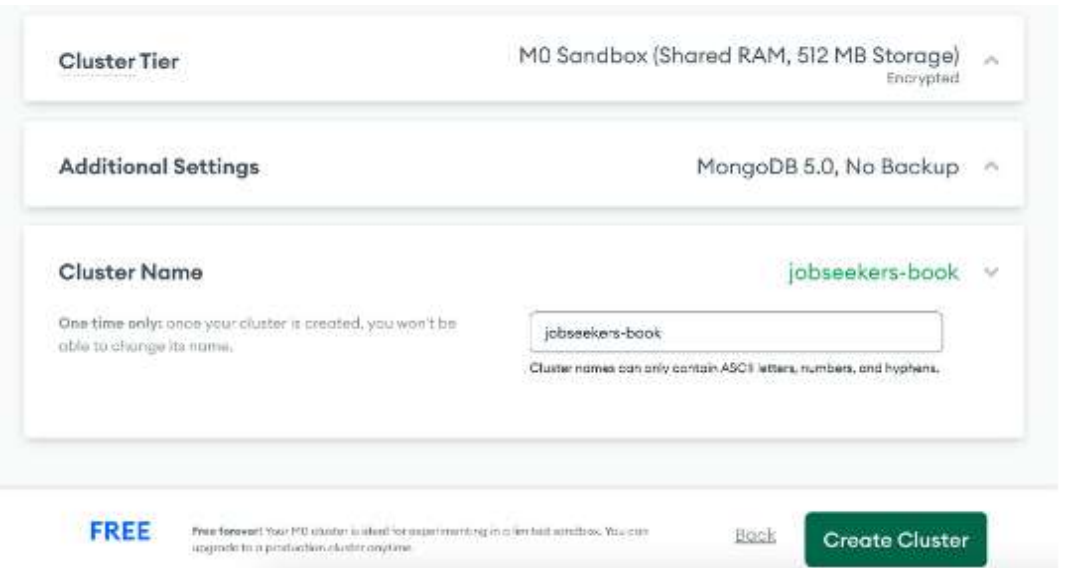


Figure 3.12: *Name Your Cluster*

4. Next, you will be asked to setup security. Again, the defaults are fine here, but to make things easier, it is suggested that you use the username/password option and pick the "**My Local Environment**" option to add your local IP. See the detailed directions on the page.
5. The last step will ask you how you would like to connect to your MongoDB Server, choose whatever seems ideal to you (suggestions are either MongoDB Compass or MongoDB Shell).

Conclusion

Hopefully, you have been able to successfully install the MongoDB Database and test via your preferred way to connect, be it MongoDB Compass or the MongoDB Shell **`mongosh`**. If you encounter any issues, you might want to refer to the MongoDB website documentation since it has a lot of help for niche cases, or if things did not go quite right.

Now so let us get started with creating our first Document!

Chapter 4

A Better Way to Store Data – Documents

"Things are only impossible until they're not!"

— Capt. Jean-Luc Picard

"I don't know where I'm going from here, but I promise I won't bore you."

— David Bowie

Introduction

If you have not worked with MongoDB Documents before, you are in for a treat. Learning about the concept of Documents for the first time is often refreshing and changes how we think about data, in dramatic ways. Less dramatically, the concepts are quite simple.

Structure

In this chapter, we will discuss the following topics:

- Importing Example Documents
    - Importing with MongoDB Compass
    - Importing on the Command Line with mongoimport

- What is a Document?
    - Other Considerations
    - Document Structure
- More About Types
- Examples of Documents

Objectives

In this chapter, we will cover some key details about Documents, rules you should know, and most importantly, some examples of real-life data put into Documents that will help you understand how you might put your own data into Documents.

Importing example documents

Before we review a few different examples, let us first import these Documents into the local MongoDB Server that we have already set up. There are two ways of creating a Database, and then importing out example Document into a Collection within that Database.

You can either use the MongoDB Compass GUI or the command line tools; pick whatever you are comfortable with.

Importing with MongoDB Compass

If you would like to use a GUI to import the example data, make sure you have installed MongoDB Compass (covered in *Chapter 3, Getting Started*). Follow these given steps:

1. Open Compass and connect to your local MongoDB Server, as discussed in *Chapter 3, Getting Started*. It should appear as shown in *Figure 4.1*:

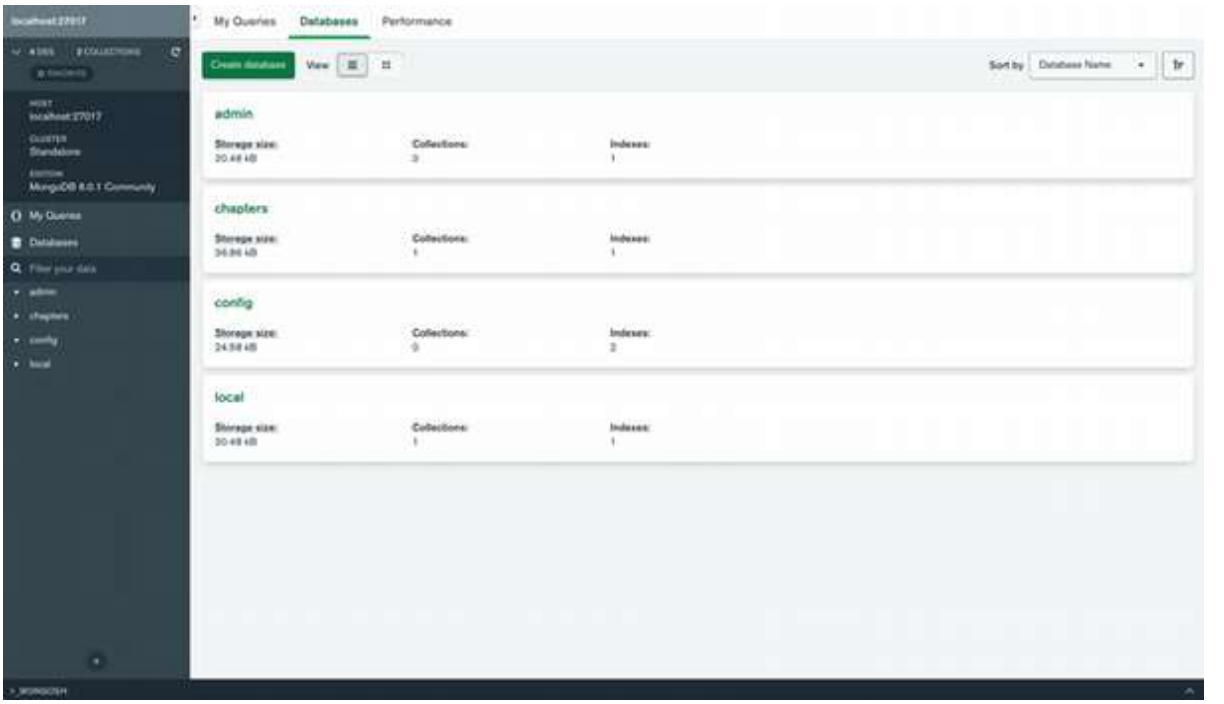


Figure 4.1: *Open MongoDB Compass*

2. Look for the (**+**) button on the bottom left. This will expose a **Create Database** button. Click on that and create a new Database and Collection via the modal window that pops up. In *Figure 4.2*, we have named the Database **book** and Collection **`examples`**:

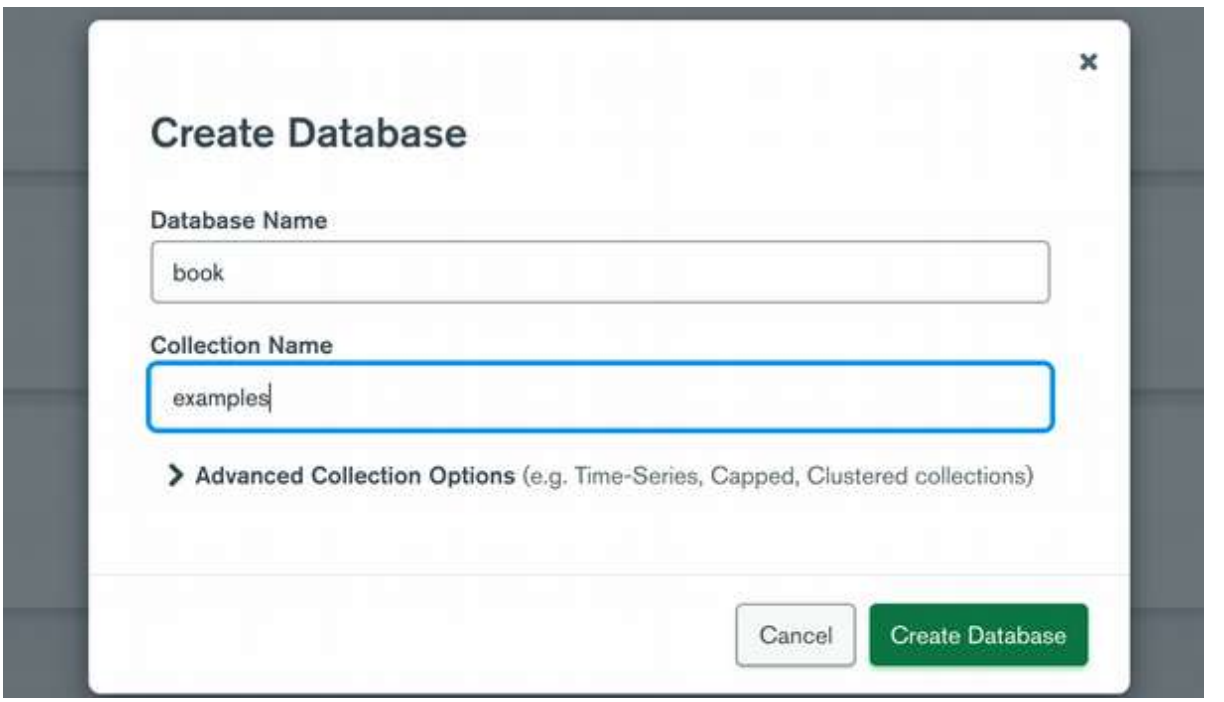


Figure 4.2: *Create Database*

3. Once you have created your Database and Collection, you can click on your Database name, and then the Collection name from the listing in the middle or on the left navigation. Click the **Import Data** button or **Add Data** button, as shown in *Figure 4.3*:

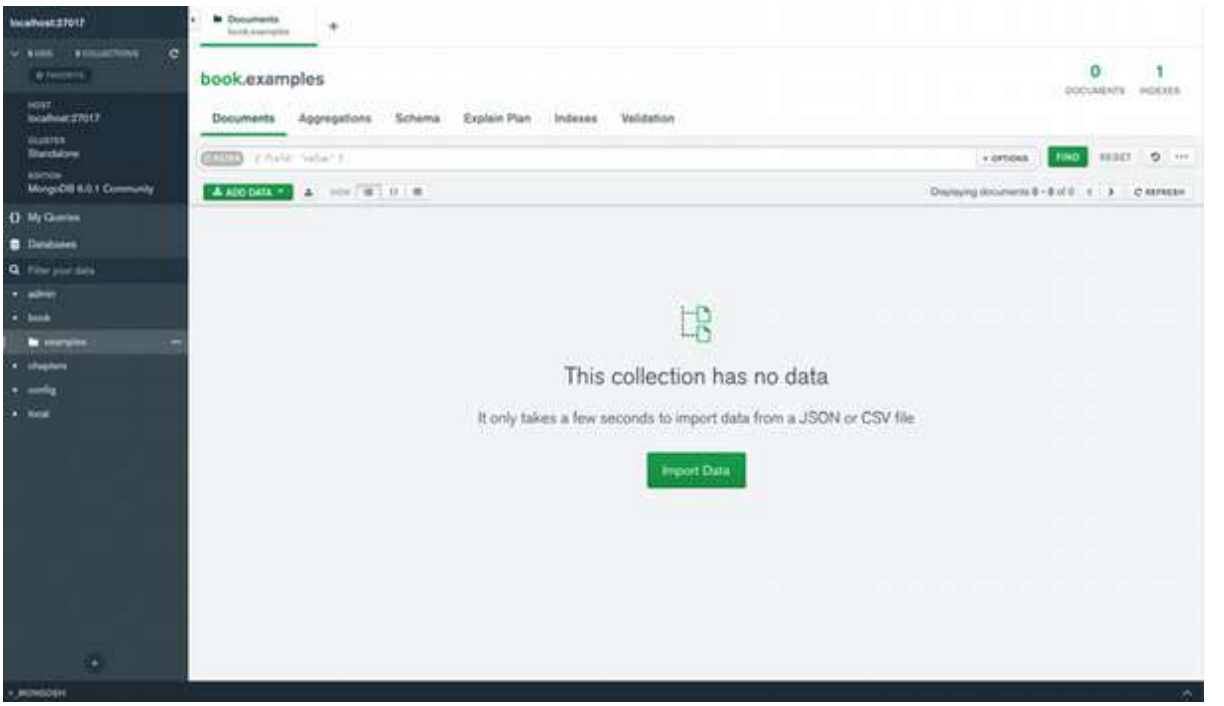


Figure 4.3: *Import Data*

4. This should open an import dialog where you can pick the **`examples.json`** file from the book git repo's **`chapters/04`** folder and make sure to select the input file type as JSON, as shown in *Figure 4.4*:

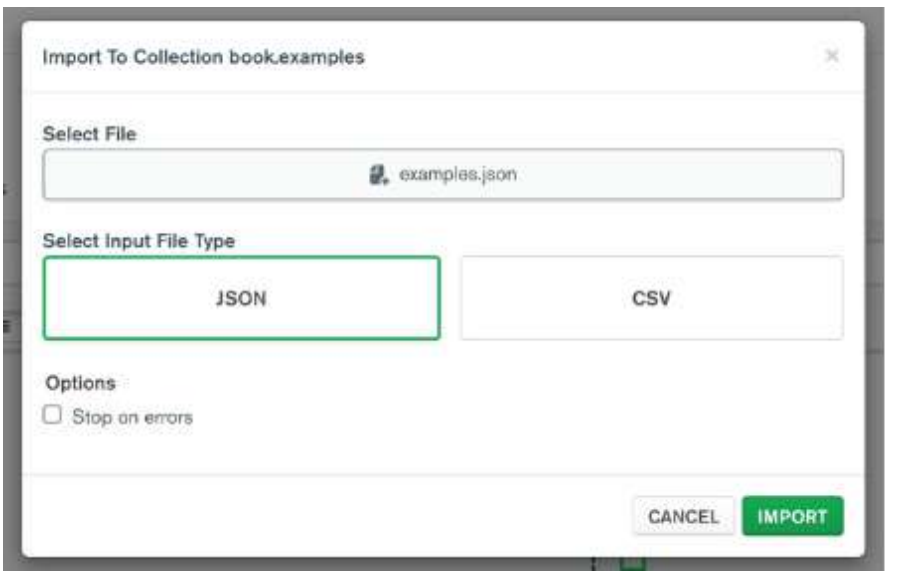


Figure 4.4: *Import Dialog*

This should import the example Documents! Feel free to browse around and look at them, we are going to discuss them in more detail, later in this chapter.

Importing on the Command Line with mongoimport

If you prefer the command line, you can easily import via the **`mongoimport`** (or **`mongoimport.exe`** on Windows) command that you should have installed along with the Database Tool earlier (see *Chapter 3, Getting Started*). If you have cloned the book's git repo, follow these steps:

1. Open a terminal, browse to the **`chapters/04`** folder in the book repo. In that folder, you should find a file called **`examples.json`**. Run this command to import that file into your MongoDB Server:

   ```
   mongoimport -d book -c examples –jsonArray examples.json
   ```

2. This should import the example Documents and have output something like:

   ```
   4 document(s) imported successfully. 0 document(s) failed to import.
   ```

We will cover their structure in the coming chapter, and how to query them in the next chapter.

What is a Document?

As we briefly discussed earlier, Documents are the way MongoDB stores data. Unlike other databases that use methods such as tables and rows, or connected nodes, MongoDB Documents generally live completely independently within their Collection and can contain various combinations of fields, different structures, and different types of data (with some basic rules we will discuss in this chapter). No single document in a Collection needs to have the same structure, or they can all have the same structure; the choice is yours!

In the simplest terms, Documents are comprised of **`field/value`** pairs and are stored within an **`object`** which is represented by curly braces `{ }` (also called curly brackets). Inside these curly braces, we can have one to many of these pairs and the values can be of any **BSON (**Binary JSON) type. As we teased earlier, these might be a number, string, date, array, object, Document, array of Documents, and so on.

```
{
  field1: 123,
  field2: "I am a string",
  field3: new Date('Sep 07, 1981'),
```

```
  field4: ["Red", "Yellow", "Green"],
  field5: { "name": "Mahika Mali", "email": "mwali@zmal.com" }
}
```

As a Document, this might look like:

```
{
  "_id": { $oid": "633a07684a5db24108dee9fa" },
  "userId": 123,
  "title": "I am a string",
  "date": { "$date": { "$date": "1981-09-07T00:00:00Z"} },
  "colors": [ "Red", "Yellow", "Green"],
  "me": { "name": "Mahika Mali", "email": "mwali@zmal.com" }
}
```

You probably noticed we now have an extra field called **`_id`** at the start of the Document (more on that later). For the moment, we can see the various data types we talked about represented here. You can have any combination of these in your Document, as many times as you need, but there are a few rules:

- The field name is a string (and as such also cannot contain the **`null`** character)
- You can use dots (.) if you wish, for example **`first.name`**.
- The name **`_id`** is reserved (and required) for use as a primary key.
- Technically, Documents can have more than one field with the same name, but this might make working with your data more difficult. So it is recommended to avoid this.

Other Considerations

While you can store a lot in Documents, there is a maximum Document size of 16 megabytes. This has to do with optimizations related to how MongoDB loads a Document into memory (RAM), and/or transfers the data over a network when it is used. If that seems limiting let us put that into perspective:

Size Perspective: 1 megabyte can store about 500 pages of text (or the size of a quite large book). The infamously long book *War and Peace* by Leo Tolstoy is roughly 1,300 pages in English. Even with the 16 megabyte limit, you could store six entire copies of *War and Peace* (or nearly eight thousand pages of text) with plenty of space left, in a single Document!

If you need to store more data, MongoDB has a feature called GridFS that will handle breaking up large files for you (for instance, if you need to store large images, or other data files).

Another rule to keep in mind is that a Document's fields are ordered (unlike objects in JavaScript), which may affect your queries.

Document Structure

Something that you have probably noticed in these example Documents is a **`_id`** field which acts as a (required) unique id for every Document in a Collection. This field serves as a "Primary Key", which is a value that is unique to that Document. Think about it like a version of an ID number on your personal identification card, or an address with an apartment number in a building. It is unique for one thing, and one thing only and used to identify or find that Document when needed.

By default, MongoDB will generate an **`ObjectId`** for you and add that value to the field **`_id`** on each Document you create. You can opt to use your own **`_id`** as well (if you want) as long as it is properly unique. You could use an auto incrementing number like many SQL databases, or generate a **Universal Unique Identifier** (**UUID**) if you need your Document's _id to be unique.

In most cases, you will want to use MongoDB's autogenerated **`_id`** as it has several advantages.

The **`ObjectId`** that MongoDB will generate contains a combination of values and pieces of data that are helpful for uniqueness and other purposes. The **`ObjectId`** is made up of 12 bytes:

- Timestamp (when the Document was created): 4 bytes
- Random generated value: 5 bytes
- Incrementing counter, starting with a random value: 3 bytes

If you want to create an **`ObjectId`** you can assign one to a variable called **`myNewId`**:

```
> var myNewId = ObjectId()
```

That will end up with a value similar to:

```
> myNewId
ObjectId("633a072f4a5db24108dee9f9")
```

Since **`ObjectId`** is its own special type of object, and has a date embedded in it, you can do magical things like:

```
> ObjectId("633a072f4a5db24108dee9f9").getTimestamp()
ISODate("2022-10-02T21:48:31.000Z")
```

This will return a datetime. MongoDB can use this to automatically sort by date, even if you did not create a date field in your Document.

MongoDB Shell Commands

In some of these examples, we used the special symbol **>** as the first character, and then some kind of **`command`** or **`variable`** name, as in:

```
> myVarible = { "field": "my new field"};
```

```
Some output shown here.
```

This denotes a command that was run on the MongoDB Shell (**`mongosh`**, which we installed earlier) and its output. We will discuss the MongoDB Shell more in later chapters.

More about types

Here is an overview of some of the BSON data types you will often use. To see all of them, visit: **https://bsonspec.org/**

String

MongoDB strings are UTF-8 by default, and so they can store multiple different characters from different languages. You do not need to specify the size of string fields; MongoDB will take care of this for you.

Numbers

While there are a couple different "number" types (usually), MongoDB will be able to auto detect what type you meant. For example, if you have a number with no quotes (quotes around a value generally means a string), that will usually be an integer. If it is longer, it will be auto converted to a long number; if it has a dot (`.`), it will be converted to a decimal.

Dates

In just about any system, dates can be tricky to deal with the realities of different formats, time zones, Daylight Saving Time and other complications can be a real pain! For example, in the United States, a person's birthday can be abbreviated as 5/10 whereas in most of the rest of the world it would be abbreviated as 10/05 or 10 May since it is the 10th day of May, the 5th month.

To avoid some of these problems, MongoDB uses a "UTC datetime" to represent and store datetimes (or the date and time together).

You will likely see two general ways to create these dates and they both follow a JavaScript like syntax:

```
> var myDate = new Date()


> var myDate = ISODate()
```

With either one, this creates a Date object, and that variable will now have methods available on it, for example:

```
> myDate.toString()
```

This will return a string:

```
Sat Oct 14 2023 23:55:35 GMT+0000 (Coordinated Universal Time)
```

Additionally, there are other date related methods:

```
> myDate.getMonth()
```

```
9
```

This will return the number for the Month:

```
9
```

Again, since dates are confusing, keep in mind that this will return a zero-indexed number for the month. So, with our example of October being the 10th month, this would return `9` just like it would in JavaScript.

These dates are based off "epoch time", the same system UNIX and Linux systems use to track dates.

Epoch dates

In MongoDB "epoch dates" are represented by integers that store the number of milliseconds since January 1st, 1970. In MongoDB, they are "signed" 64-bit integers; this means that negative signed (-) vales are dates before January 1st, 1970, and positive values are after.

August 15th, 1947 would be **`-706291200`** whereas Jan 28th 1986 would be **`507283200`**.

This gives us a date range of roughly *290 million years* of dates into the future, and the past!

As mentioned earlier, types like Date will be converted by the database drivers into an appropriate type, and so you will get back a proper "date" in your programming language.

Arrays and objects

One awesome feature of Documents is that you can use JavaScript arrays and objects. We have teased using these in some of our introductory examples but suffice to say, pretty much anything that you can do with an array in JavaScript, you can do in MongoDB also! You can have an array of simple numbers, or strings, or a mix of both, or arrays within arrays, arrays of objects, and so on. You can even make the data you store in our database, match how you use it in code.

There are various helper methods to query and modify these arrays which we will cover in more detail in *Chapter 7, Complex Data, Made Simple*.

Objects in MongoDB are essentially Documents within Documents. You can store whatever you can in a Document within them; this can be a great way to store smaller related Documents together.

Types when importing/exporting

Sometimes you might see Dates or other types represented in slightly different ways when you import/export data in MongoDB (more on that in *Chapter 10, Getting In and Getting Out – Data Migration*). For example, you might see a date stored like this:

```
"date": {
      "$date": "2023-05-24T16:00:00Z"
}
```

This is a standard ISO Datetime stored as a string, like you might see in JavaScript.

For an **`ObjectId`**:

```
"_id": {
      $oid": "633a07684a5db24108dee9fa"
}
```

MongoDB uses these special **`$`** field names to store the data in JSON, but identifies that the field is actually BSON. Doing this makes sure that MongoDB can convert types appropriately when you import or export your Documents.

Examples of documents

Sometimes the easiest way to understand something is to see some real-life examples. In this section, we will show several different examples of Documents storing a variety of data and then discuss them a little.

Note: Along with the example data (which includes these Documents), you can find copies of each of these Documents in the book's git repo in the chapters/04 folder.

Stock data

There are many ways we might store the complexities of stock data within a Document. There are also things like MongoDB's *Time Series Collections* which is a special type of Collection for Documents with data that changes overtime (see more in *Chapter 9, Planning for Performance – Collections and Indexes*). Here, we will represent change overtime within a Document:

```
{
    "_id": { "$oid": "633a60954a5db24108dee9fb" },
    "symbol": "STOCK",
    "volume": 1400000,
    "high": 23.01,
    "low": 19.95,
    "close": 22.01,
    "quotes": [
        { "price": 20.32, "date": { "$date": "2022-11-01T13:00:00Z" } },
        { "price": 19.95, "date": { "$date": "2022-11-01T14:00:00Z" } },
        { "price": 21.13, "date": { "$date": "2022-11-01T15:00:00Z" } },
        { "price": 23.77, "date": { "$date": "2022-11-01T16:00:00Z" } },
        { "price": 22.01, "date": { "$date": "2022-11-01T17:00:00Z" } }
    ]
}
```

At the bottom of this Document, we have a **`quotes`** array that is keeping track of stock quotes over the course of the day.

We have used an array to store smaller Documents (objects) that track the **`price`** and the **`date`** that the quote was recorded. We could actually store *a lot* more data in each of these "subdocuments" such as the exact volume, other stock's prices, and so on, but for the sake of space here, we just tracked the price and date time.

The rest of the fields are straightforward, storing either string, number, or decimal data.

If we had some application running that updated our stock's data throughout the trading day, we could easily have it append these "mini" Documents to the **quotes** array, thus preserving order so we could use that array to create charts. This application could also check if the price at that moment also necessitates us updating the **high** or **low** fields; for example, if the quote is larger than the current value for **high**, we should update the stock to have a new high!

User profile

Next, we have a very simple example of a user profile Document with a person's name, contact information, and so on:

```
{
    "_id": { "$oid": "633b7b644a5db24108dee9fc" },
    "user_id": 2,
    "first_name": "Grace",
    "last_name": "Hopper",
    "title": "Rear Admiral",
    "email": "grace.hopper@navy.mil",
    "password": "Qzh3NCZ4V0;eZWd3Zdla!c5Y2h3Zg",
    "address": {
        "street": "321 Minor Street",
        "city": "Seattle",
        "state": "WA",
        "zip": "80921"
    },
    "recipe_favorites": [
        "recipe:apple-pie",
        "recipe:chicken-tacos"
    ]
}
```

Notice how we used an object (or subdocument) to store the user's **address** in a way that is a lot more accurate and less brittle than one long string, while also being more structured than separate columns for street, city, state, zip that are not necessarily clear they "go together". We can use this to better store addresses for various countries since our schema, or structure, is more flexible.

We also have a **`recipe_favorites`** array with two of the user's favorite recipes. Why do you think we might have strings like that to represent a recipe?

Recipe

For a recipe document example, let us use a version of an old family recipe for apple pie! Here, we will apply some ideas we used in the house sale listing but expand on them by using a variety of date types in arrays: objects, string, and numbers.

```
{
    "_id": "recipe:apple-pie",
    "title": "Apple Pie",
    "servings": 8,
    "calories_per_serving": 384,
    "cook_time": 45,
    "prep_time": 25,
    "desc": "All American pie",
    "ingredients": [
        { "name": "pie crusts", "amount": { "quantity": 2 } },
        { "name": "granny smith apples", "amount": { "quantity": 6 } },
        { "name": "granulated sugar", "amount": { "quantity": 0.75,
"unit": "cup" } },
        { "name": "cinnamon", "amount": { "quantity": 1, "unit": "tbsp"
} },
        { "name": "nutmeg", "amount": { "quantity": 1, "unit": "tps" }
},
        { "name": "lemon", "amount": { "quantity": 1 } }
    ],
    "directions": [
        "Preheat oven to 425 F",
        "Put bottom crust in pan place in refrigerator",
        "Peal, de-core and chop apple into 1/2 inch pieces (or slice
into thin pieces)",
        "Mix flour with sugar and spices",
        "Mix apples with flour/spice/sugar and juice from lemon",
        "Pour into pan with bottom crust and optionally slice pieces of
button on top",
        "Place top crust on top and pinch sides. Cut small slits into
middle of top.",
```

```
        "Cover edges with foil around top of crust and remove during
last 20 minutes.",
        "Bake 45 minutes."
    ],
    "likes": [2,1,75],
    "likes_count": 3,
    "rating": [5,5,5,5,1,5,5],
    "rating_avg": 4.8,
    "type": "Dessert",
    "tags": ["traditional", "4th of July"]
}
```

First off, you might notice we did not use an **`ObjectId`** here for the **`_id`**. Instead, we used a custom compound id of **`recipe:`** plus, a "slug" (like what you might see used in a custom URL) of the recipe's title. This could be any valid unique string and probably you would still want to use to an **`ObjectId`**, but this is showing how you have the option to use custom ids as well!

We "linked" to this recipe in the user profile example in the **`recipe_favorites`** array above; using custom **`_id`** in this manner might be helpful if you want user readable ids.

The recipe **`ingredients`** are an array of objects with a **`name`** and **`amount`**. The **`amount`** is an object that has a **`quantity`** filed and optional **`unit`** field.

In this example, we did not do any pre-conversion of the quantity/units (that is, storing both the imperial and metric measures) as in all likelihood, we would need to recalculate these, because the user might have requested the recipe with a different **`servings`** amount, and so on. Thus, we kept the data cleaner and simpler here. (See the next example for how we might do this.)

The **`likes`** array here is storing user ids of users on our site that liked this recipe (see the User Profile Document above) and we have stored the count of the total likes in **`likes_count`**.

We could also have some code to count the members of the **`likes`** array, but this and the **`rating_avg`** field are examples of pre-calculating data from other fields and storing that in our Document for ease of use.

Home sale listing

Let us now store a typical house listing:

```
{
    "_id": { "$oid": "633a07684a5db24108dee9fa" },
    "address": {
        "street": "321 Minor Street",
        "city": "Seattle",
        "state": "WA",
        "zip": "80921"
    },
    "total_size": { "sqft": 2100, "m2": 195 },
    "bathroom_count": 3.25,
    "bedroom_count": 4,
    "bedrooms": [
        { "name": "Office / Bedroom", "size": { "sqft": 150, "m2": 14 }
},
        { "name": "Bedroom", "size": { "sqft": 165, "m2": 15.329 } },
        { "name": "Bedroom", "size": { "sqft": 165, "m2": 15.329 } },
        { "name": "Main Bedroom", "size": { "sqft": 180, "m2": 16.7 } },
    ],
    "levels": 3,
    "asking_price": 729000.00,
    "price_history": [
        {
            "date": { "$date": "2019-09-01T00:00:00Z" },
            "price": 536000.00
        },
        {
            "date": { "$date": "2012-05-05T00:00:00Z" },
            "price": 387000.00
        },
        {
            "date": { "$date": "1990-02-23T00:00:00Z" },
```

```
            "price": 250000.00
        }
    ],
    "heating": "Forced Air",
    "cooling": null
}
```

Here we were able to store a typical house listing. Again, we are using an object to store the address. Since the measurements of a house generally do not change, we are storing the **`total_size`** of the house and the room sizes by using an objects with the size in both square feet (sqft) and meters squared (m2), which means that we do not need to do calculations or conversions later.

We also store the price history in an array of objects, from which we could easily build a chart, again without much or any data modification.

Conclusion

In this chapter, we have covered how to import some example Documents, gone into more detail about what you can store in them, as well as some gotchas. Through a couple distinct examples, we have shown how MongoDB's Document model gives you a lot of flexibility for how you store your data.

In the next chapter, we will dive into querying Documents, so if you have not already done so, now would be the time to make sure you import the example Documents we have just discussed! We will be digging into how to find Documents in general, how to query by specific details in a Document and a lot more.

Chapter 5

Let's Do It – Create, Update and Delete Documents

> **"Those who follow the crowd usually get lost in it."**
>
> — *Rick Warren*

> **"Things never happen the same way twice."**
>
> — *Aslan (C.S. Lewis)*

Introduction

Now that we have covered what you can store inside a document, you are probably excited to learn how to create them, update them, and when their time comes, delete them.

You may have heard the term **CRUD**. It stands for **Create Read Update Delete**. In this chapter, we will cover the basics of each one of those, with a focus on Create, Update and Delete. We will tackle a good deal more of the Read part in the next chapter (*Chapter 6, Getting What You Want – Querying)*. That said, since it is not very useful to read from a Database that does not have anything in it, let us get started with making Documents.

Structure

In this chapter, we will discuss the following topics:

- Creating Documents

    - Using the MongoDB Shell
    - Inserting a document
    - Other ways to query
    - Creating more complex documents
    - Inserting multiple documents
    - Inserting using MongoDB Compass
- Updating Documents
    - Adding new fields
    - Removing fields
    - Updating multiple documents
    - Updating using MongoDB Compass
- Deleting Documents

Objectives

By the end of this chapter, you will have a basic idea of how to create, update and delete documents within MongoDB. We will also discuss how to do each of these actions to multiple documents at one time.

Creating Documents

There are several ways to create Documents for MongoDB. In this chapter, we will focus on using the MongoDB Shell (`mongosh`) and MongoDB Compass. You could also use your favorite programming language (more on that in *Chapter 14, Making Stuff – Programming with MongoDB*).

Using the MongoDB Shell

The most simple and straightforward way to create a Document is to use the MongoDB Shell. We have touched on the Shell (`mongosh`) a little bit so far, but we have not really explained what it is or how it works.

The MongoDB shell is a way to connect and use MongoDB from the command line (the operating system does not matter), and since MongoDB uses a lot of JavaScript concepts, its shell uses JavaScript too. If you do not know JavaScript, do not be concerned. You do not need to be able to program to use MongoDB; however, if you do know JavaScript, a lot of these concepts will come very naturally.

To start up the Shell, follow the given steps:

1. Run **`mongosh`** (or **`mongosh.exe`** if you are on Windows) from a command line.
2. Alternatively, if you installed MongoDB Compass, you can press **>_MONGOSH** at the bottom left of MongoDB Compass, and it will open a shell within the Compass UI for you, as shown in *Figure 5.1*:

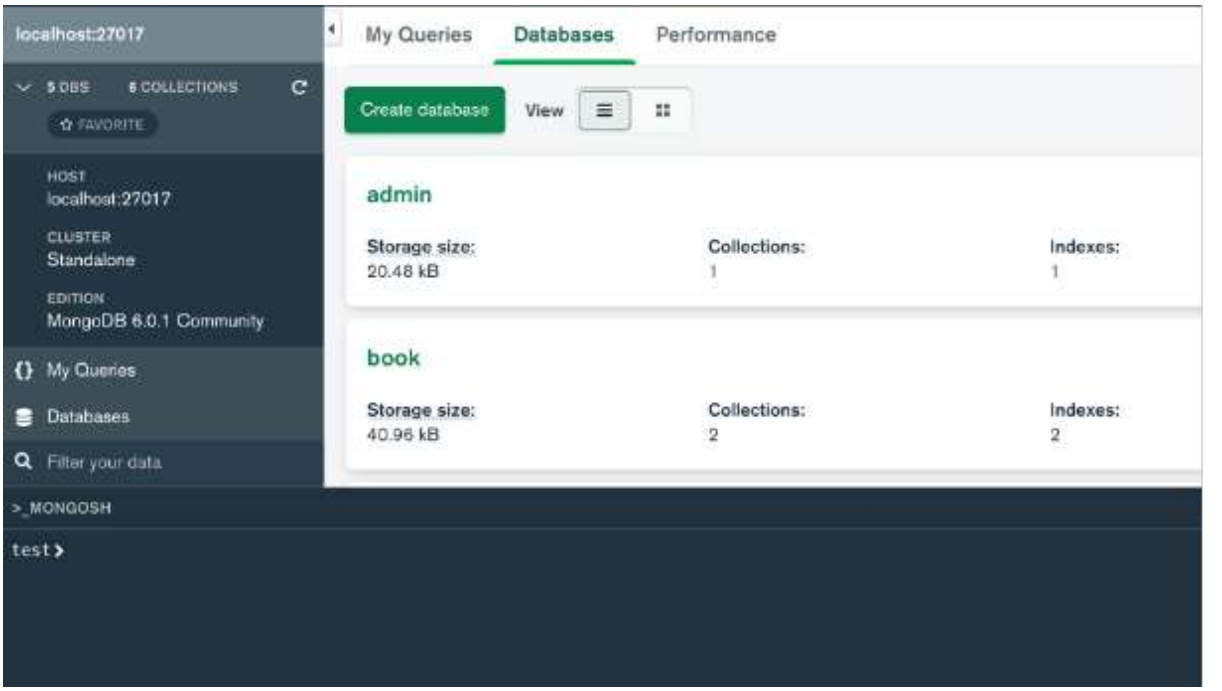


Figure 5.1: *Using the Shell Within MongoDB Compass*

Note on MongoDB Shell Example Input/Output: In the following examples, when you see a ">", it implies a command you can type (input) into the shell. Sometimes, some text shows up under the command. That is what the Shell will output. When you are using the shell, the default will be to either show the Database's name and > or just > as the "prompt". For clarity in our examples, we will just show a >.

From the **`mongosh`** shell prompt, let us first switch to the **`book`** Database:

```
> use book
switched to db book
```

We used a special command called **`use`** to switch to the **`book`** Database (or **`db`**). If we type out **`db`** and press enter, it will show the name of our current Database:

```
> db
book
```

Within the shell, you can assign variables as you would in JavaScript, and we will do that to create our first document:

```
> var myDoc = { "name": "foo" }
```

Type out the aforementioned command and press Enter. With this command, we created a simple variable using a variable called **myDoc**.

If you type out the variable name and then press enter the shell, it will output the contents of the variable as a Document (assuming it can be processed as a valid Document):\

```
> myDoc

{ name: 'foo' }
```

Inserting a Document

That is great, but we have not yet saved this Document anywhere. Currently, it only lives in memory, within your shell instance. So, let us save our Document to a Collection (we will use **docs** as the Collection name, but you can use whatever name you want, if the Collection deos not exist already, it will be created for you automatically when you use it). To create and save our Document, we will use a method called **insertOne()** on our Database and Collection combination **db.docs**:

```
> db.docs.insertOne(myDoc)
```

This should output something like:

```
{ acknowledged: 1, insertedId: ObjectId("634f4ec540bb2c39f4968397") }
```

Once successful, it inserts our Document and gives us back an auto generated **ObjectId** which we talked about earlier. This **ObjectId** will be the Document's unique _id.

View Our New Document

We can now retrieve and view our Document, back from the Database, by using a simple query via another other method called **find()**:

```
> db.docs.find()


[ { _id: ObjectId("634f4ec540bb2c39f4968397"), name: 'foo' } ]
```

Since we have only one Document in this Collection, we get back an array with a single Document inside it, as we can see above (note that the **_id** that was added). If we assign a new variable and insert/save another Document:

```
> var myDoc2 = { "name": "bar" }
> db.docs.insertOne(myDoc2)

{ acknowledged: 1, insertedId: ObjectId("634f513440bb2c39f4968398") }
```

Then run **`find()`** again. We will get back two Documents in an array:

```
> db.docs.find()

[
  { _id: ObjectId("634f4ec540bb2c39f4968397"), name: 'foo' },
  { _id: ObjectId("634f513440bb2c39f4968398"), name: 'bar' }
]
```

Other ways to query

There are a few other ways we could have queried that; and since we have two documents now, we likely will want to do that:

Find By ObjectId

```
> db.docs.find(ObjectId("634f4ec540bb2c39f4968397"))

[ { _id: ObjectId("634f4ec540bb2c39f4968397"), name: 'foo' } ]
```

In this query, we used an **`ObjectId`** directly, which is valid since there can only be one Document in that Collection with that ObjectId. Even though there are multiple Documents in the Database now, we will only get back the one Document that matches the query.

Find By Document Field

```
> db.docs.find({ "name": "foo" })

[ { _id: ObjectId("634f4ec540bb2c39f4968397"), name: 'foo' } ]
```

In this example, we used a "Document" to match; we are asking for where a Document has a `name` field with the value of **`foo`**. We could have used the **`_id`** here too, or any other field in the Document.

Find one document

If we know we just want *one* Document back, we can use a different command:

```
> db.docs.findOne(ObjectId("634f4ec540bb2c39f4968397"))


{ _id: ObjectId("634f4ec540bb2c39f4968397"), name: 'foo' }
```

With **`findOne()`** you might notice the results did not come back in an array, just the single Document that matched. We will cover querying much more in *Chapter 6, Getting What You Want – Querying*.

Creating more complex documents

More complex Documents can be created via the command line as well; take this Document describing a tuna salad sandwich:

```
> var sandwich = {
    name: "Tuna",
    bread: "wheat",
    ingredients: [
      { name: "meat", type: "tuna salad" },
      { name: "cheese", type: "swiss" }
    ],
    price: 7.40
  }


> db.docs.insertOne(sandwich)


{ acknowledged: true, insertedId: ObjectId("634f6a7a0a2d27b2b4cf2b65") }
```

Using this, we have now created a delicious sandwich Document (at least if you like tuna sandwiches!) using strings, arrays, objects, and decimals.

Using the MongoDB Shell as a JavaScript Shell

As an example of how **`mongosh`** is a JavaScript shell, let us use the **`sandwich`** variable we made, but just extract the **`ingredients`** array:

```
> sandwich.ingredients


[
  { name: 'meat', type: 'tuna salad' },
```

```
  { name: 'cheese', type: 'swiss' }
]
```

Cool! We got back just the array using a JavaScript style "dot syntax". You could also use array syntax **`sandwich["ingredients"]`**. We can now go a step further and apply JavaScript operations on our `ingredients` field:

```
> sandwich.ingredients.filter((ingredient) => ingredient.name == 'cheese')


[ { name: 'cheese', type: 'swiss' } ]
```

Since the `ingredients` field is an array, we can use JavaScript array methods on it such as **`filter`**, to get back just the ingredient with the name **`cheese`**. You can do this directly as part of a query as well, by "changing on" the JavaScript methods to a **`find()`** or **`findOne()`** command:

```
> db.docs.findOne({"name" : "Tuna"}).ingredients.filter((ingredient) |
ingredient.name 'cheese')

[ { name: 'cheese', type: 'swiss' } ]
```

Here we are using the **`ingredients`** array we know exists within this Document and then filtering on it to get just the data we want.

Inserting multiple documents

For cases when you need to insert more than one Document, you can use **`insertMany()`** which works mostly the same as **`insertOne()`**, but takes an array of Documents to insert:

```
> db.docs.insertMany([ {"name": "doc 1"}, {"name": "doc 2"} ])


{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("6351e8e9fd50d98494cb0256"),
    '1': ObjectId("6351e8e9fd50d98494cb0257")
  }
}
```

To change things up, here we did not create any variables first. Instead, we created two documents inside an array, and directly inside the method. You can use this style for both types of inserts (the difference being the array) if you prefer.

Insert using MongoDB Compass

If you opted to install MongoDB Compass, you can use the **ADD DATA** button and choose **Insert Document**, as shown in *Figure 5.2:*

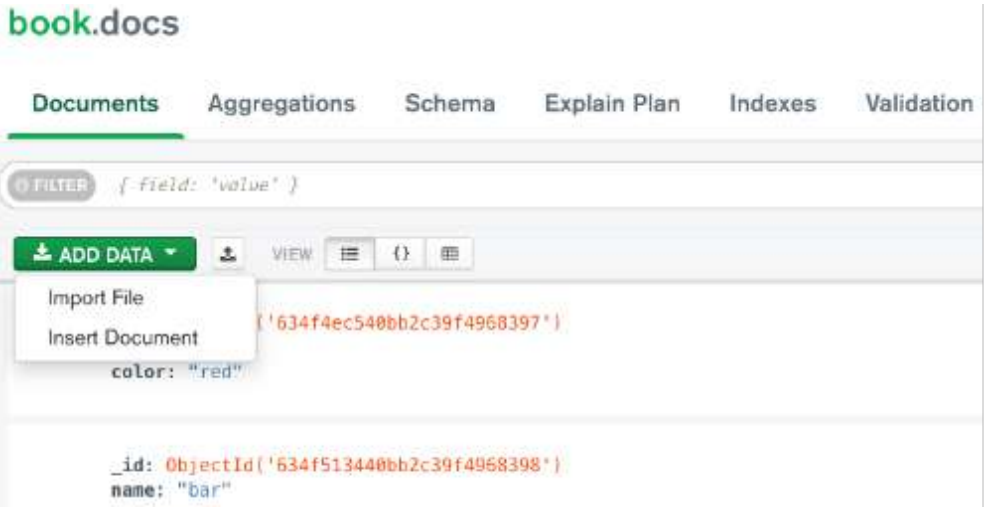


Figure 5.2: *MongoDB Compass Add Data*

This should pop open something like the model in the following *Figure 5.3*, where you can type out the Document a lot like we did with the variables in our prior examples:

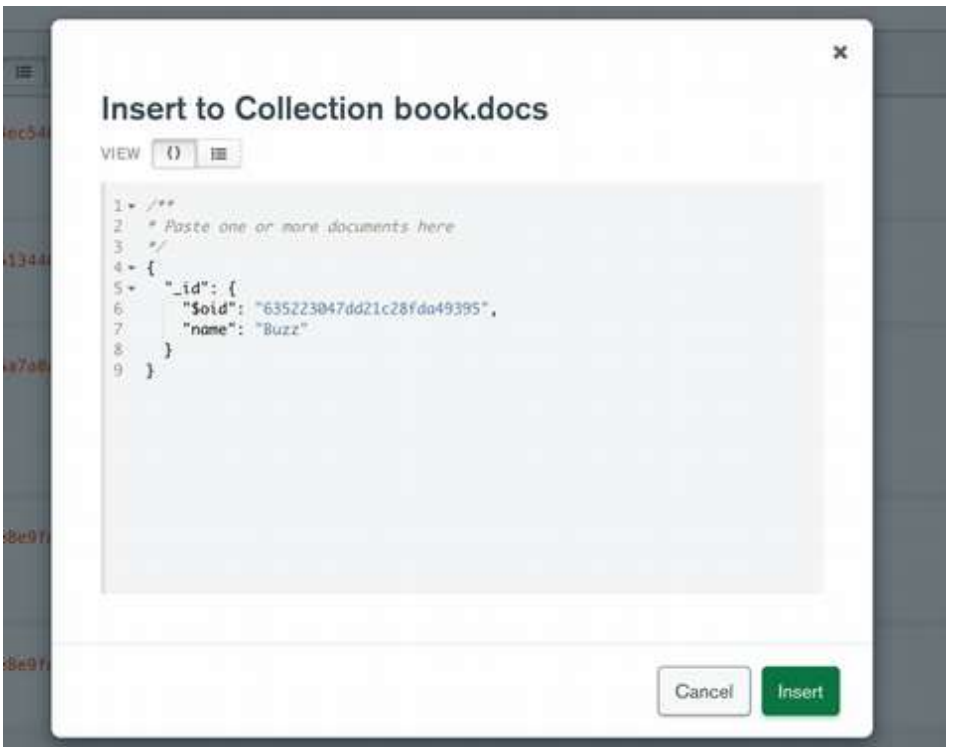


Figure 5.3: *MongoDB Insert*

This model will make sure the Document you are trying to create, is valid before it will let you insert. Therefore, double check what you typed as it might not be clear why the Document is not valid.

Press **Insert**, and you should be able to immediately view your new Document within Compass.

Updating Documents

To update a Document, we can use **updateOne()** and for this method, we will need to provide a query to *match* the Document we want to update, much like we did for **find()** earlier. Here we are using an **_id** and then specifying what we want to update:

```
> db.docs.updateOne(
    { "_id" : ObjectId("634f4ec540bb2c39f4968397")},
    { $set: { "name": "Buz"} }
  )
```

This will output something like:

```
{
  acknowledged: 1,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

The first thing that you might notice is this slightly weird looking **$set**. When you see this special **$** prefix in front of something in query, it is called an "operator". Operators are the way MongoDB distinguishes specific parts of query as being a command. In this query, we asked to **$set** (or change) the value of the **name** field to **Buz**.

When we run **find()** again, we should see that the **name** field of the Document has now changed.

Adding new fields

You can also use **$set** to add new fields to an existing Document:

```
> db.docs.updateOne(
```

```
    { "_id" : ObjectId("634f4ec540bb2c39f4968397")},
    { $set: { "color": "red"} }
  )
```

This will add a field called **`color`** to our Document, and set its value to **`red`**.

Removing fields

If you need to remove a field from a Document use **`$unset`** like so:

```
> db.docs.updateOne(
    { "_id" : ObjectId("634f4ec540bb2c39f4968397")},
    { $unset: "color" }
  )
```

To remove (unset) more than one field at a time, pass an array of field names instead:

```
> db.docs.updateOne(
    { "_id" : ObjectId("634f4ec540bb2c39f4968397")},
    { $unset: ["color", "name"] }
  )
```

It is important to note that for **`updateOne()`**, MongoDB will update the first matching Document it finds. In most of these examples, we used an **`_id`** as our matcher, which is much safer than other fields since they will only ever match one document. However, if you used another field, make sure you are matching just the one Document you expect to update.

Updating multiple documents

Much like inserts, update has both an **`updateOne()`** and **`updateMany()`**, which you can use to update more than one Document at a time.

Be careful with **`updateMany()`**. You will want to be double sure with **`updateMany()`** as it will update *all matching* Documents in your Collection.

```
> db.docs.updateMany(
    { "name" : "doc 1" },
    { $set: { "color": "red" } }
)
```

```
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

This query would update all the Documents in your Collection, that have a field called **`name`** with the value of **`doc 1`** and set (update or add) the separate field called **`color`** with the value `red`.

"Upsert" a Document

There are times when a slightly different operation might be necessary. For example, if you know a field that you need to update in a Document, but you are not actually sure if that exact Document exists yet.

Imagine you have a large Collection with Documents that contain a user's email address along with their preferences, for instance, if they are subscribed to your company's newsletter, and so on. Your website has a page where users fill out a form to subscribe to the newsletter and they provide their email on that form. When the user submits the form, you could choose to handle it a couple different ways:

- Simply insert a new Document using their email and their subscription; but you might already have a Document with their email from some prior interaction and now you have two Documents *with the same email* (which might mean that the user will get multiple emails from us at a later point and complain about SPAM).
- Create a unique index (constraint) on the email address field to make sure you cannot create multiple Documents. However, that will throw an error when you try to insert (if the Collection already has a Document with email address), so that is not much better either. (More about indexes in *Chapter 9, Planning for Performance – Collections and Indexes*.)
- Run a query to look for the submitted email address, get the matching Document's **`_id`** (if any exist) and then do an **`updateOne()`** to update or an **`insertOne()`** to create a new Document. While this works, that is a good amount of logic that will require a lot more code.

To avoid all these issues, we could instead do what is called an "upsert", which will update any matching Documents. If none match, MongoDB will insert (create) a

new Document. Using an "upsert" means we do not need any extra code to handle situations like these, and we avoid all the issues above.

By default, MongoDB will not perform an upsert so to accomplish this, we will need to add a *third* parameter to our **`updateOne()`** and within that parameter set **`upsert`** equal to **`true`** (this parameter can take more options too, see the MongoDB docs):

```
> db.docs.updateOne(
    { "email" : "youngatheart@aol.com" },
    { $set: { "newsletter": true } },
    { upsert: true }
)
```

Now if there is an existing Document with that email, it will be updated. If not, a new one will be inserted, with no SPAM complaints and no extra code!

Updating Using MongoDB Compass

MongoDB Compass offers a graphical interface for updating Documents, which includes some helpful things such as labeling what type each field is, as well as grouping together things like arrays.

Hover over a Document to expose the buttons on the top right. Then choose the "**Pencil**" icon to "**Edit Document**" and you should see a view somewhat like *Figure 5.4*:

Figure 5.4: *MongoDB Compass*

You can press the field's type (for example, String, Array) on the right, to switch its type, as shown in *Figure 5.5*, which will give you a unique interface to edit that field's type:

***Figure 5.5**: MongoDB Compass Change Type*

The best way to understand how this interface works is to just mess around within Compass. So if you prefer a graphical interface, go ahead and mess around!

Deleting Documents

Like insert and update, MongoDB has a **`deleteOne()`** and **`deleteMany()`** with the main difference being that **`deleteMany()`** will *delete all Documents that match the query condition*. With **`deleteOne()`** only the first matching Document will be deleted:

```
db.docs.deleteOne({ "_id" : ObjectId("634f4ec540bb2c39f4968397") })
```

```
{ acknowledged: true, deletedCount: 1 }
```

With this in mind, be very careful, especially while using **`deleteMany()`** with exactly what your query condition looks like or you might end up deleting a lot of Documents you did not mean to!

For example, if you pass an empty Document as your match query, you will delete all Documents in that collection:

```
db.docs.deleteMany({})
```

Note: Even if you delete all the Documents in a Collection, if you have an index, it will not be dropped. You must do this separately (more on indexes in *Chapter 9, Planning for Performance – Collections and Indexes*).

Conclusion

You should now have a basic idea of how to insert, find, update, and delete Documents in MongoDB using methods such as `insertOne()`, `insertMany()`, `find()`, `findOn()`, `updateOne()`, `updateMany()`, `deleteOne()` and `deleteMany()`.

We learned to keep in mind important details such as double checking and being super careful when using `updateMany()` or any sort of delete. You always need to make sure you get your matching query condition correct, so you do not modify or delete Documents by mistake!

We also learned that there are some cool tricks we can do with the MongoDB Shell (`mongosh`) since it is also a JavaScript shell! In the next chapter, we will start to dive more deeply into how to query Documents with MongoDB.

Chapter 6
Getting What You Want – Querying

> **"It's me!"**
>
> — *Mario*

> **"I couldn't read the way that other students read, so I would just cheat, which, in my silly brain, I was like, 'This is a skill that I'm developing - how to just get around everything!'"**
>
> — *Joel McHale*

Introduction

In this chapter, we will cover all the basics of querying within MongoDB from how to find something you want, returning exactly what you need, to getting it back in the order desired.

Structure

In this chapter, we will discuss the following topics:

- Importing Example Documents
- MongoDB Shell vs MongoDB Compass

- Querying MongoDB
- The MongoDB Query API
- MongoDB Query Operators
- Objects and Arrays
- Query Case Sensitivity

Objectives

By the end of this chapter, you should be comfortable with querying MongoDB using various query operators, understand the core concepts of querying with MongoDB, as well as tools you can use to deal with unique challenges such as case sensitively, in MongoDB.

Importing Example Documents

For the examples in this chapter, you will want to import a new batch of example documents. If you cannot recall how to do that, refer to *Chapter 4, A Better Way to Store Data – Documents.*

We will import these documents into a Collection called **`cookbook`**, as they are a collection of recipes. You can find the documents in an import file called **`cookbook.json`** in the **`chapters/06`** folder in this book's git repository.

Once imported, the file you should have 7 new documents in your cookbook Collection.

MongoDB Shell vs MongoDB Compass

In the following examples, we will be using the full query syntax for MongoDB, which you would use in the MongoDB Shell. These same queries can be constructed slightly differently, using just part of the query, for use in the MongoDB Compass UI.

MongoDB Shell Queries

In the MongoDB Shell, to find the recipe for "Toast" you would type out:

```
> db.cookbook.find({ "title" : "Toast" })
```

Even though we used the **`find()`** method here instead of **`findOne()`**, given our example data, this should return just one matching Document.

As a reminder, the **`>`** represents the prompt, type everything after that.

MongoDB compass queries

In the Compass UI, you will only pass in part of the query as the rest is configured within the UI, as seen in the following *Figure 6.1*:

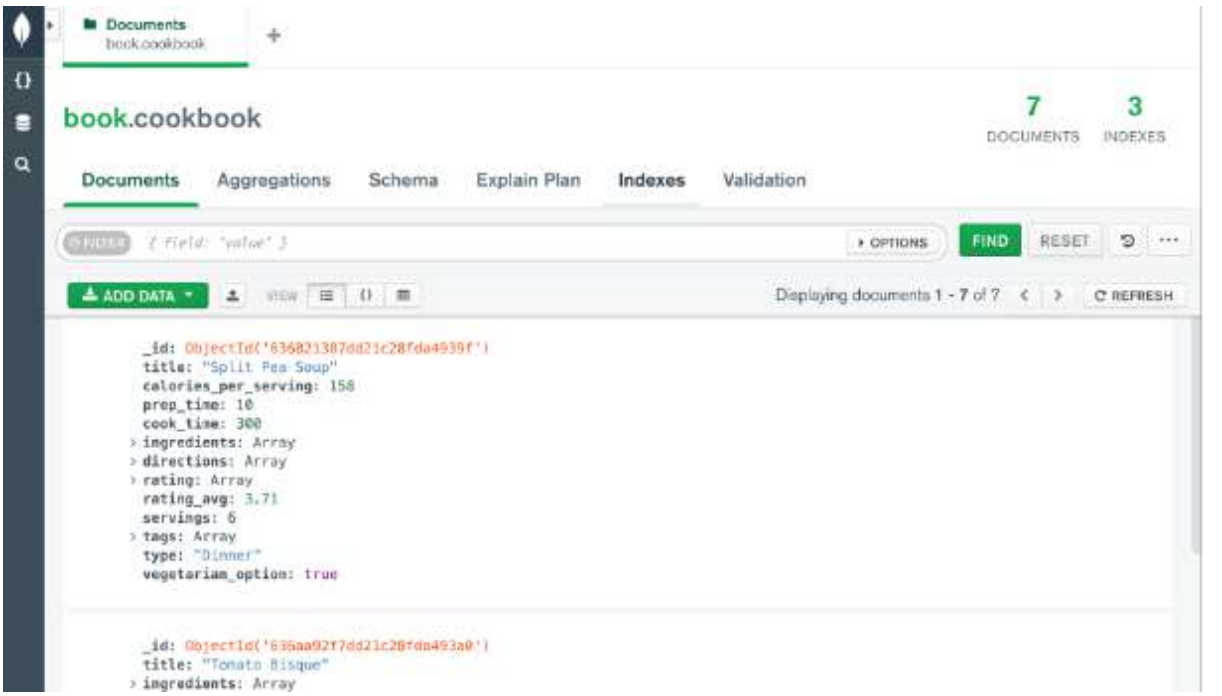


Figure 6.1: *Cookbook Collection Within MongoDB Compass*

Within the **`FILTER`** input field, you would type out the query document for the document you want to match. So, here if we were looking for "Toast", you would type the following within the "filter input":

```
{ "title" : "Toast" }
```

Then you can press the green **`FIND`** button; this will return a single document, the recipe for Toast, as seen in the following *Figure 6.2*:

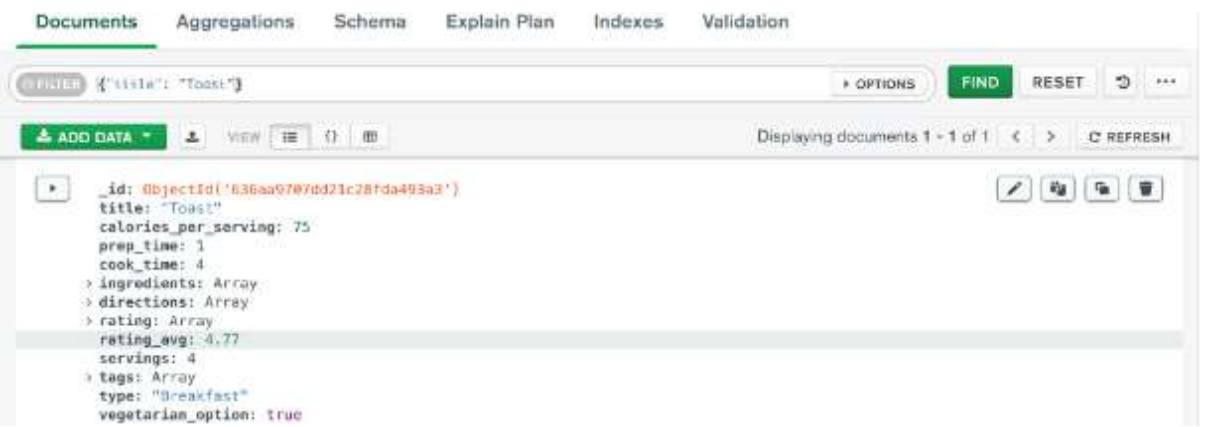


Figure 6.2: *Find the Recipe for Toast in MongoDB Compass*

For the examples in this chapter, we will use the "full" query. So if you are using MongoDB Compass, make sure to simply type in the "document" portion of the query, that is the part within the **`{ }`**.

Querying MongoDB

As a quick review, there are two basic ways to select documents from your Collection **`findOne()`** and **`find()`**. For most of the examples in this chapter, we will use **`find()`** as it will return all matching documents, and not just the first one to match. It does this by returning a "cursor" which can then be "iterated" on. If you are familiar with iterators from a programming language, this might immediately make some sense to you. If not, there is no need to worry; the concept is easy to understand.

Why cursors?

The example documents in our **`cookbook`** Collection adds up to 7 in total, but in a production database, this count might be in the hundreds, thousands or even millions of documents. When you are working with sizes like that, it important for the database to handle things efficiently. One way MongoDB does this is by using the concept of a "cursor". You have probably used a cursor before and not known it, as they are generally obfuscated away from you by things such as database drivers. MongoDB is no different.

In very simple terms, a cursor is a "pointer" to a result set. This differs from how queries can work in other databases that will typically attempt to return all the data you requested back to you in one set.

By default, MongoDB will return a smaller portion of the documents that your query matched, and the cursor gives you a way to get back more documents as you need them. Most of the time, this is handled by a basic **`while()`** loop, continually requesting more data until the cursor is exhausted. You might also apply some sort of operation while doing this (more about that in *Chapter 14, Making Stuff – Programming with MongoDB*).

If you are using the MongoDB Shell or MongoDB Compass, this will be mostly taken care of for you (behind the scenes) and you will simply see the output as an array of documents.

The MongoDB Query API

Breaking things down more granularly, **`find()`** has three basic parts. A **filter** (or matching document), a **projection** (what fields you want returned) and any additional things you want to do with the results such as, **`sort()`** or **`limit()`** and so on.

To compose a query that will return all the documents in a collection, and bring back the entire document, with all its fields, you can use an empty matching document for both:

```
> db.cookbook.find({},{})
```

This will bring back all the documents, sorted by how they are currently stored in the Collection, with no limit. You could express the same query this way:

```
> db.cookbook.find()
```

Using Filter to match documents

If you wanted to find recipes in our cookbook with the title "Toast", you will need to specify a more precise filter document:

```
> db.cookbook.find({ "title": "Toast" })
```

This should return our recipe for toast.

Note: This sort of search is case sensitive, so using a lowercase "toast" would not match our Document (more on that later).

This query is equivalent to the following SQL query:

```
SELECT * FROM cookbook WHERE title = "Toast"
```

Using Projection to Control Output

By default, a query will bring back the entire document. In many cases, this is not necessary or might cause performance issues.

To configure what fields of a document you want back, you can use the **project** (or projection) parameter of **`find()`**. Here, we will request to get back just the **`title`** field of each document:

```
> db.cookbook.find({}, { "title": 1 })


[
  { _id: ObjectId("636821387dd21c28fda4939f"), title: 'Split Pea Soup' },
  { _id: ObjectId("636aa92f7dd21c28fda493a0"), title: 'Tomato Bisque' },
   { _id: ObjectId("636aa94c7dd21c28fda493a1"), title: 'Zucchini Fudge
Cake' },
  { _id: ObjectId("636aa9617dd21c28fda493a2"), title: 'Stuffed Peppers'
```

```
},
  { _id: ObjectId("636aa9707dd21c28fda493a3"), title: 'Toast' },
  { _id: ObjectId("636aa9817dd21c28fda493a4"), title: 'Eggs Benedict' },
  { _id: ObjectId("636ab56e956f91c56f02f049"), title: 'Blue Cheese Burg-
ers' }
]
```

For the **projection**, we have the **`title`** field and a **`1`** (which stands for true). As a result, MongoDB will return the **`title`** instead of the whole document. You will notice MongoDB still returned the **`_id`** field. This is the default behavior since the **`_id`** field is the unique key for the document.

In this case, since we did not specify a **filter**, we got back all 7 documents. This sort of query is the same as the following SQL query:

```
SELECT title FROM cookbook
```

Within MongoDB Compass, you can press the **`OPTIONS`** button next to the **`FIND`** button to expand more fields, including the **project**, where you can insert **`{ "title": 1 }`**, as shown in the following *Figure 6.3*:

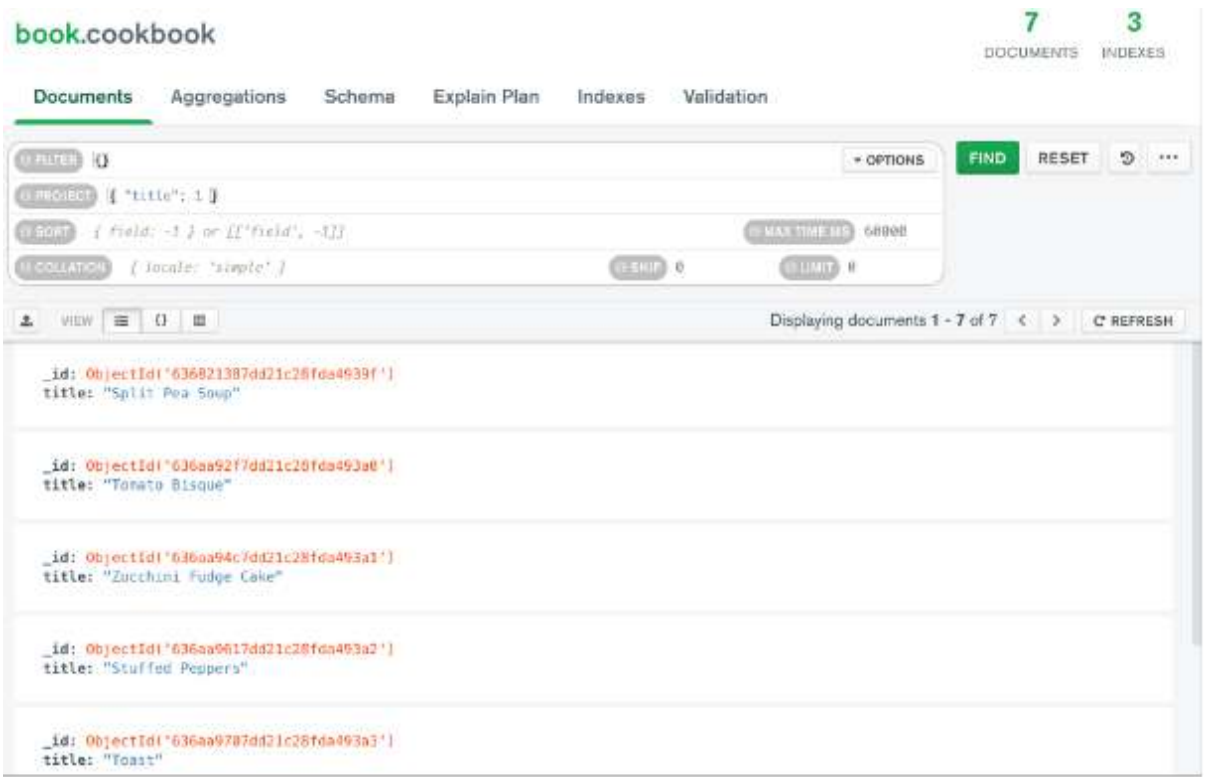


Figure 6.3: *Using the project option in MongoDB Compass*

If we did not want the **_id** field (or any field for that matter), we can do the opposite and specify the field name and use a **0** (which stands for false):

```
> db.cookbook.find({}, { "title": 1, "_id": 0 })


[
  { title: 'Split Pea Soup' },
  { title: 'Tomato Bisque' },
  { title: 'Zucchini Fudge Cake' },
  { title: 'Stuffed Peppers' },
  { title: 'Toast' },
  { title: 'Eggs Benedict' },
  { title: 'Blue Cheese Burgers' }
]
```

You can use an object consisting of multiple field names and either a **1** to "add", or a **0** to "remove" fields, to get the output you desire. If that field is not in your document, it will not cause a problem as MongoDB knows that a document's schema is flexible.

Using sort() to Order Output

You will have noticed thus far that the documents have come back in a non-alphabetized order; in this case, it is by "insertion order". In other words, this is done in the order the documents were created. To change that order, we will use an additional method "chained on" onto **find()** with a **.** called **sort()**. This is a "cursor method" or an operation where we are telling MongoDB to run on the data *before* it is returned to us from the server.

We can use **sort()** like so:

```
> db.cookbook.find({}, { "title": 1, "_id": 0 }).sort({ "title": 1 })


[
  { title: 'Blue Cheese Burgers' },
  { title: 'Eggs Benedict' },
  { title: 'Split Pea Soup' },
  { title: 'Stuffed Peppers' },
```

```
  { title: 'Toast' },
  { title: 'Tomato Bisque' },
  { title: 'Zucchini Fudge Cake' }
]
```

Now we have our recipes in A-Z order or ascending, by the **`title`** field. In this case, the **`1`** stands for **`ascending`** (the default) and **`-1`** stands for **`descending`**, so you could change the query like so to get back the documents in Z-A order:

```
> db.cookbook.find({}, { "title": 1, "_id": 0 }).sort({ "title": -1 })
```

Within MongoDB Compass, you can press the **`OPTIONS`** button to expose additional fields where you can then input **`{ title: 1 }`** or **`{ title: -1 }`** in the sort field, as seen in the following *Figure 6.4*:

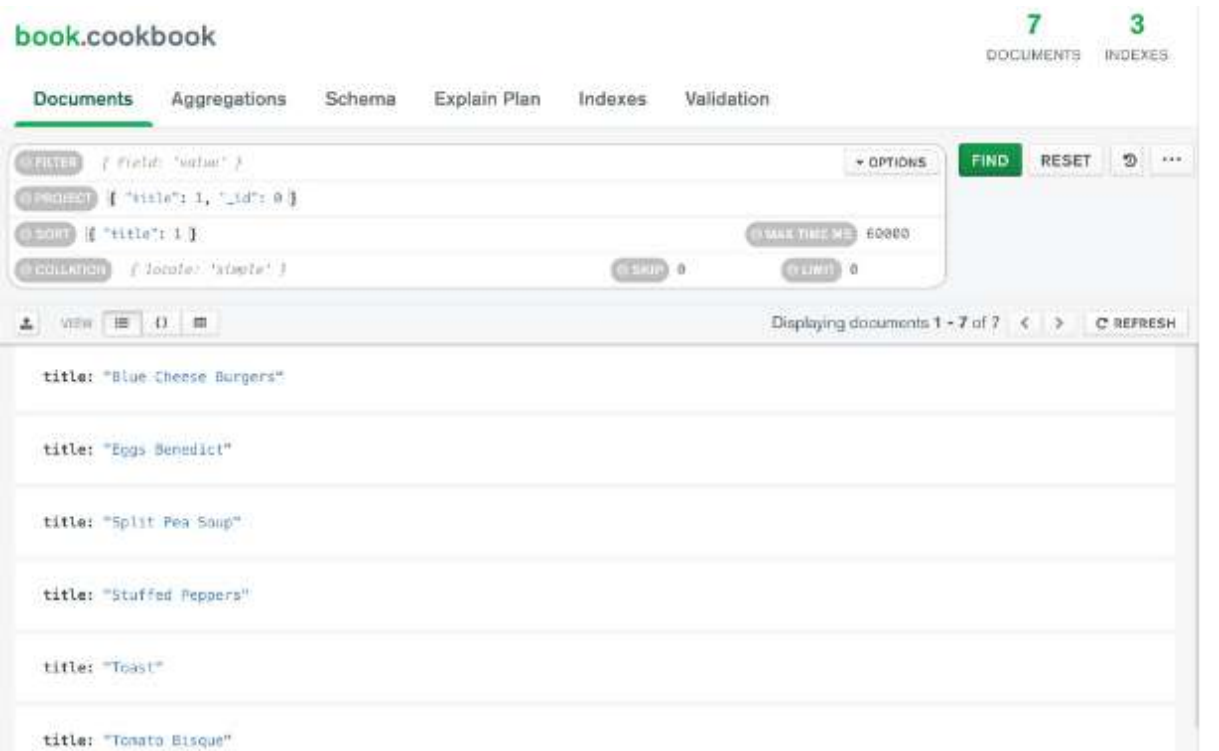


Figure 6.4: *Using the sort option in MongoDB Compass*

These queries are the same as using **`ORDER BY`** in SQL, for example:

```
SELECT title FROM cookbook ORDER BY title DESC
```

We can also sort (or order by) multiple fields. In this case, we will add another field to our output, the recipe **`type`** and order by that and then the recipe's **`title`**:

```
> db.cookbook.find({}, { "title": 1, "type": 1, "_id": 0 }).sort({ "type":
1, "title": 1 })
```

```
[
  { title: 'Eggs Benedict', type: 'Breakfast' },
  { title: 'Toast', type: 'Breakfast' },
  { title: 'Zucchini Fudge Cake', type: 'Dessert' },
  { title: 'Blue Cheese Burgers', type: 'Dinner' },
  { title: 'Split Pea Soup', type: 'Dinner' },
  { title: 'Stuffed Peppers', type: 'Dinner' },
  { title: 'Tomato Bisque', type: 'Dinner' }
]
```

Now we have our recipes first ordered by the **`type`**, and then the **`title`**!

Note: You can sort on things not in the projection or "output" if you need to. We have included both fields in the projection here to make it easier to visualize.

Using Variables in Queries

If you are using the MongoDB Shell, you can also assign these options to variables. For example, to make our query shorter, we can define two variables (named whatever you want), and then use them in our **`find()`** query (or any query after that):

```
> var output = {"title": 1, "type": 1, "_id": 0}
> var sortBy = {"type": 1, "title": 1}


> db.cookbook.find({}, output).sort(sortBy)


[
  { title: 'Eggs Benedict', type: 'Breakfast' },
  { title: 'Toast', type: 'Breakfast' },
  { title: 'Zucchini Fudge Cake', type: 'Dessert' },
  { title: 'Blue Cheese Burgers', type: 'Dinner' },
  { title: 'Split Pea Soup', type: 'Dinner' },
  { title: 'Stuffed Peppers', type: 'Dinner' },
  { title: 'Tomato Bisque', type: 'Dinner' }
]
```

This gives us the same results, but within a much more compact query. As a reminder, all this is possible within MongoDB Compass using the **MONGOSH** option, as shown in the following *Figure 6.5*:

Figure 6.5: *Using JavaScript within MongoDB Compass*

Using count() and limit() and skip()

To illustrate some other cursor methods, let us use the **type** field from on our recipe documents. Here, we will get a count of how many documents have a particular **type**:

```
> db.cookbook.find({ "type": "Dinner" }).count()

4
```

So, there are four recipe documents that have the **type** of "Dinner". You could also use "Breakfast" or "Dessert" here. Anything else will result in a count of **0**. You can chain the **count()** onto pretty much any query, to get the count of matching documents.

We can also purposely limit the results to whatever number we want:

```
> db.cookbook.find({ "type": "Dinner"}, { "title": 1, "_id": 0 }).limit(2)


[
 { title: 'Split Pea Soup' },
 { title: 'Tomato Bisque' }
]
```

In this case, we limited our request to two, so we get the first two matching documents. You can use this with **`sort()`** to get the first two, ordered by whatever you need to:

```
> db.cookbook.find({}).sort({ "title": 1 }).limit(2)
```

This will get us the first two documents, ordered by the **`title`**.

Another operation is **`skip()`** which you can use to "skip into" your results. In the following example, we constructed a query that used all three together: sort the recipes by **`title`** descending (Z-A), skip to the second document and limit to just one result:

```
> db.cookbook.find({}).sort({ "title": -1 }).skip(2).limit(1)
```

Pop Quiz: What recipe document did that query match? Why don't you run it and find out!

MongoDB Query Operators

So far, we have been using very simple "something equal to" queries, but there is a lot more to the query language in MongoDB, or more properly the MongoDB Query API. In the upcoming section, we will cover additional ways to query, using what are called "operators".

Comparison Operators

First off, if you have had a chance to look around at the recipe documents, you will notice there are **`cook_time`** and **`prep_time`** fields which are numbers (in minutes). Let us try querying one of on those:

```
> db.cookbook.find({ "cook_time": 30 })


{
  _id: ObjectId("636aa9617dd21c28fda493a2"),
  title: 'Stuffed Peppers',
  ...
}
```

This will return our "Stuffed Peppers" recipe since it is the only recipe that takes exactly 30 minutes to cook. However, that sort of query is not super helpful with more varied data. What we probably would rather get back, are recipes that take 30 minutes *or less*, for example. To do this, we will use a query operator instead. Mon-

goDB has a special way to express these operators (and other things that would not be valid in JSON) and that is by using a dollar sign **$**.

For example, to compare "less than" you would use **$lt** or, for "less than or equal to" **$lte** within a smaller document and assign that to the field you want to compare to in the query as shown:

```
> db.cookbook.find({ "cook_time": { $lte: 30 } }, { "title": 1 })


[
  { _id: ObjectId("636aa94c7dd21c28fda493a1"), title: 'Zucchini Fudge
Cake' },
  { _id: ObjectId("636aa9617dd21c28fda493a2"), title: 'Stuffed Peppers'
},
  { _id: ObjectId("636aa9707dd21c28fda493a3"), title: 'Toast' },
  { _id: ObjectId("636aa9817dd21c28fda493a4"), title: 'Eggs Benedict' }
]
```

With this query, we got back more than one recipe; all the recipes that take 30 minutes or less. We also reduced the fields returned by requesting just the recipe **title** in the **projection**, to make the results a little clearer.

Unsurprisingly, there are also "greater than" **$gt** as well as "greater than or equal to" **$gte** operators available. You can use these operators on number and decimal fields as well as for values inside arrays, but we will cover that later.

It is often useful to get just the count of matching documents. For this, we can use the **count()** method we learned about earlier, which will allow us to find out the total count of recipes that take 30 minutes or less to cook with a query like this:

```
> db.cookbook.find({ "cook_time": { $lte: 30 } }).count()

4
```

There are some other useful comparison operators available to determine if values exist: **$in** and **$nin** for "in" and "not in". You can use these to match, or unmatch, one or more value in a single field. Using our **type** field again, we could match all recipes that are of the **type** "Dinner" or "Dessert" using this query:

```
> db.cookbook.find({ "type": { $in: ["Dinner", "Dessert"] } })
```

Or perhaps we want to opposite; any recipes that aren't "Dinner" or "Dessert":

```
> db.cookbook.find({ "type": { $nin: ["Dinner", "Dessert"] } })
```

This sort of query is searching for "unmatching" documents, as opposed to "matching" ones as we normally do. Both operators can be used for values in arrays as well, but more on that later in *Chapter 7 Complex Data, Made Simple*.

Using Operators in update queries

Keep in mind that you can use these for the matching (or unmatching) documents in an **`update()`** as well, like have here:

```
> db.cookbook.updateMany(
   { "type": { $in: ["Second Breakfast", "Elevenses"] } },
   { $set: { hobbit_meal: true } }
)
```

This would match any documents that have a **`type`** of "Second Breakfast" or "Elevenses" and set a field called **`hobbit_meal`** to **`true`** (in this case, add the field and set it; or if it already existed, update the field).

Unfortunately, for any hobbits out there, they will need to wait for their next meal because there are no recipes that would match if we ran this query. Sorry hobbits!

Field Update Operators

There are several other useful operators to update fields. For instance, if you need to change the name of a field in one of your documents, you can use the **`$rename`** operator:

```
> db.cookbook.updateOne(
  { "_id": 100 },
  { $rename: { "title": "name" } }
)
```

This will change the name of the **`title`** field to **`name`**, and you can add as many fields as you need within this operation.

Take another situation. What if you had a document that was tracking views of a certain movie and you want to be able to update it each time the movie is watched. Say our document looks like this:

```
{
    _id: ObjectId("6371c0ab47082f5188ff9bfc"),
    title: 'The Hudsucker Proxy',
```

```
    views: 99,
    last_viewed: ISODate("2021-10-02T17:00:00.000Z")
  }
```

Now, you have just watched the movie and we would like to increment the value of **views** by one, and set the **last_viewed** to the current date. At first, you might think we would need to query the collection for our document, get the number of views, add one to that value and then update the document.

Luckily MongoDB makes this much easier using two operators, **$inc** and **$currentDate**:

```
db.movies.updateOne(
  { "title": "The Hudsucker Proxy" },
  {
    $inc: { "views": 1 },
    $currentDate: { "last_viewed" : true  }
  }
)
```

In the preceding query, we used **$inc** and a value of **1** to increment the **views** by one. However, it could be any number, positive or negative, to increment however you need.

Imagine the case of a game where you need to increment the **score** by **1000** when the player beats a certain boss, but also decrement their **life** by **150** because they took some damage. To accomplish this, you could do something like:

```
db.game.updateOne(
  { "user": "PlayerOne" },
  { $inc: { "score": 1000, "life": -150 } }
)
```

You can also multiply a field's value. Take the case where a player's **life** gets doubled for finding a key item in the game:

```
db.game.updateOne(
  { "user": "PlayerOne" },
  { $mul: { "life": 2 } }
)
```

Now we have doubled the player's **life**, by multiplying it by **2**.

Atomic Operations

In the preceding examples, with **$inc** and **$mul**, MongoDB is using something called an **Atomic Operation**. In short, that means that the operation happens within only that document, at the exact time of the update.

So, if something else is trying to update the **views** at the same time, MongoDB will handle each request one by one and make sure the new total in **views** reflects the *current* value of the **views** field plus 1, every time.

This means you do not need to worry about multiple operations happening to the document or "locking" your documents, as each operation will be handled atomically.

Logical Operators

You may have been thinking that the **$in** operator sounds a lot like a logical "OR" statement, and it is! The main difference is that **$in** matches on a single field. So, if you need to do an "OR" comparison on *multiple fields*, you can use the **$or** operator. You can also use more than one field or additional operators when comparing with **$or** as follows:

```
> db.cookbook.find({ $or: [ { type: "Dinner"}, { cook_time: { $lte: 30 }
} ] })
```

Notice that here we start the query with an **$or** instead of a single field like **type**, and then search on two different fields, with two different types of comparisons within an array. Here, we match any document that has the **type** dinner *or* has a **cook_time** of 30 minutes or less.

There is also an inverse option for **$or** called **$nor**. You can think about it like *"I would not like that, nor this!"* and you use it as follows:

```
> db.cookbook.find({ $nor: [ { type: "Breakfast"}, { cook_time: { $gt: 30
} } ] })
```

This will match, or rather *unmatch* to get us any recipe documents that are not "Breakfast" as well as ones that will not take more than 30 minutes to cook.

To see this a little more clearly, we might want to specify what fields we want back with the **projection** option we learned about earlier:

```
> db.cookbook.find({
   $nor: [ { type: "Breakfast"}, { cook_time: { $gt: 30 } } ] },
   { "_id": 0, "title": 1, "type": 1, "cook_time": 1}
```

```
)

[
  { title: 'Tomato Bisque', type: 'Dinner' },
  { title: 'Zucchini Fudge Cake', cook_time: 25, type: 'Dessert' },
  { title: 'Stuffed Peppers', cook_time: 30, type: 'Dinner' },
  { title: 'Blue Cheese Burgers', type: 'Dinner' }
]
```

When you need to be more restrictive in your searches, you can use the **$and** operator which, unsurprisingly, means a document must match all comparisons in our comparison array:

```
> db.cookbook.find({
  $and: [{ type: "Breakfast" }, { cook_time: { $lte: 30 } }] },
  { "_id": 0, "title": 1, "type": 1, "cook_time": 1 }
)

[
  { title: 'Toast', cook_time: 4, type: 'Breakfast' },
  { title: 'Eggs Benedict', cook_time: 6, type: 'Breakfast' }
]
```

This will return two recipes, whereas, if we switch this to an **$or** we will get back 4 recipes because the recipes can match either of our conditions:

```
> db.cookbook.find({
  $or: [{ type: "Breakfast" }, { cook_time: { $lte: 30 } }] },
  { "_id": 0, "title": 1, "type": 1, "cook_time": 1 }
)

[
  { title: 'Zucchini Fudge Cake', cook_time: 25, type: 'Dessert' },
  { title: 'Stuffed Peppers', cook_time: 30, type: 'Dinner' },
  { title: 'Toast', cook_time: 4, type: 'Breakfast' },
  { title: 'Eggs Benedict', cook_time: 6, type: 'Breakfast' }
]
```

Element Operators

For more niche cases, there is an operator called **`$not`**, that will match if a field *does not exist* in a document, since documents are flexible and may not have a particular field in every document in a collection.

Additionally, there are some other operators you might want to use in special cases called **`$exists`** and **`$type`**. These operators will let us use match on if a field exists in an individual document or if a field is of a particular type (like string, or number, or array).

If you are familiar with **`EXISTS`** in SQL, **`$exists`** is not the same thing. In MongoDB's Query API, that is more analogous to the **`$in`** we covered earlier.

Field Type Considerations

Being able to query by a fields type can be useful in some particular situations; take a document like this:

```
{
  "_id": 123,
  "item": "Bananas",
  "price": "2.00"
}
```

Most of the time storing the **price** as a string (as we did here) will probably be fine, but it does make calculations a little trickier, and we cannot use operators as robustly if MongoDB does not know the string value in the field really represents a **decimal**.

For example, if we ran this query for items that cost less than 3.00:

```
> db.items.find({ "price": { $lt : 3.00 }})
```

We will get back zero results. If, however we convert that to a decimal, we could take advantage of the **`$lt`** operator. To find and update this document (and any others like it) we could use **`$type`** like so:

```
> db.items.find({ "price": { $type: "string" }})
```

Then, to update that field (there are other ways to do this, but by way of example), we could create a function to convert the field via JavaScript:

```
> var updatePrice = (doc) => db.items.updateOne(
  { "_id": doc._id },
  { $set: { price: NumberDecimal(doc.price) }}
)
```

This will create a function called **updatePrice()** within the MongoDB Shell that will take a **doc** as input. We can then run the **updateOne()** MongoDB command, and **$set** the **price** field to a converted value. This handled by a helpful built-in function called **NumberDecimal()**.

We can then combine our function with our query, and JavaScript's built in **forEach()** method as follows:

```
> db.items.find({ "price": { $type: "string" } }).forEach(doc => update-
Price(doc))
```

This will find all documents where the **price** field is of type "string", and then loop through each one, converting the value of **price** to a decimal.

If this is feeling a bit over your head, do not be concerned. This is just an example of the power available to you and is not necessarily something you need to fully understand at this point.

You can use all sorts of JavaScript, or even Node methods, within the MongoDB Shell to do quick, functional operations. Additionally, there many drivers for various programming languages that have methods to handle most use cases.

Objects and arrays

We will cover this in more detail in the next chapter, but as a quick overview, you can query fields inside an array or object by using the "dot" syntax just like you would in JavaScript.

If we are looking recipe with an ingredient named "salt", we can run a query on the **ingredients** array, which has objects (sub documents) inside it for each ingredient. Each ingredient has a **name** field which we can access like so:

```
db.cookbook.find( { "ingredients.name": "salt" }, { "title": 1 })


[
  { _id: ObjectId("636821387dd21c28fda4939f"), title: 'Split Pea Soup'
},
  { _id: ObjectId("636aa92f7dd21c28fda493a0"), title: 'Tomato Bisque' },
  { _id: ObjectId("636aa94c7dd21c28fda493a1"), title: 'Zucchini Fudge
Cake' },
  { _id: ObjectId("636aa9617dd21c28fda493a2"), title: 'Stuffed Peppers'
}
]
```

Query Case Sensitivity

An important difference with MongoDB compared to some other database systems is that MongoDB queries on text or "string" fields in a case sensitive fashion. That means that if we search on the **`title`** field for recipes named "toast", none will be returned. This is because the **`title`** is capitalized as "Toast" in our document, and "toast" does not strictly equal "Toast".

Using regex queries

A common way to deal with this issue is to use the **`$regex`** operator in the search:

```
> db.cookbook.find({"title": { $regex: /toast/i } })
```

This will match a document with the **`title`** "toast" or "Toast" because we are telling MongoDB to use a regular expression instead. That regular expression specifies a case insensitive search with the **`/i`** option.

It should be noted that in certain cases, this is not ideal, because indexes cannot be used as efficiently.

Options for Dealing with Casing Issues

If this is a problem for your use case, there are a couple of other options that can be implemented on the document level. One of these is quite simple but does require a little overhead. Let us take this example document representing an episode of the television series Star Trek:

```
{
  "_id": "tos_s01_e05",
  "title": "The City on the Edge of Forever",
  "directors": ["Joseph Pevney"],
  "writers": ["Harlan Ellison", "D. C. Fontana"],
  "tags": ["Time Travel", "World War II", "Cordrazine", "Keeler"]
}
```

Since the **`title`** is formatted in a "title case", we have a mix of capitals and lower-case letters that might easily not match a user's search exact search string. A way through which we can both preserve the exact casing we want in the **`title`** and make searching on it easier, is to create a shadow field that contains the value of the **`title`** field, but without any formatting called **`title_search`** and use that field to search on:

```
> db.episodes.find({ "title_search": "the city on the edge of forever" })

{
  "_id": "tos_s01_e05",
  "title": "The City on the Edge of Forever",
  "title_search": "the city on the edge of forever",
  "directors": ["Joseph Pevney"],
  "writers": ["Harlan Ellison", "D. C. Fontana"],
  "tags": ["Time Travel", "World War II", "Cordrazine", "Keeler"]
}
```

Maintaining Shadow Fields

To maintain these fields, when we insert or update the document, we could first run a small bit of code in whatever programing language our app uses, in order to make sure all the letters are lowercase.

You can do this right within the MongoDB Shell since it uses JavaScript! First assign a variable with your document inside it:

```
> var doc = {
  "title": "The City on the Edge of Forever",
  "directors": ["Joseph Pevney"],
  "writers": ["Harlan Ellison", "D. C. Fontana"],
  "tags": ["Time Travel", "World War II", "Cordrazine", "Keeler"]
}
```

Then we will update that variable by using the "splat" or spread syntax (…) from JavaScript, to copy all the fields from our **`doc`** and then add an additional field called **`title_search`**. We will use our **`title`** field, **`doc.title`** and the JavaScript method **`toLowerCase()`** to lowercase the text in that field (you could also assign this to a new variable):

```
> doc =  { ...doc, title_search: doc.title.toLowerCase()}
```

If we type out our variable and press enter, you will see we now have both fields in our variable document:

```
> doc

{
```

```
  title: 'The City on the Edge of Forever',
  directors: [ 'Joseph Pevney' ],
  writers: [ 'Harlan Ellison', 'D. C. Fontana' ],
  tags: [ 'Time Travel', 'World War II', 'Cordrazine', 'Keeler' ],
  title_search: 'the city on the edge of forever'
}
```

We can then use that variable to insert the document:

```
> db.episodes.insert(doc)
```

With this change, when we do any searches, we can simply lowercase all the letters in our search string, and we do not need to worry about casing at all.

Storing Shadow Fields as Objects

Another variation might be to use an object to store the two different values:

```
{
  "_id": "tos_s01_e05",
  "title": {
    "display": "The City on the Edge of Forever",
    "searchable": "the city on the edge of forever"
  },
  "directors": ["Joseph Pevney"],
  "writers": ["Harlan Ellison", "D. C. Fontana"],
  "tags": ["Time Travel", "World War II", "Cordrazine", "Keeler"]
}
```

You can do this in your programming language, or again in MongoDB Shell using the spread syntax:

```
> var doc = { ...doc,
  "title": {
    "display": doc.title,
    "searchable": doc.title.toLowerCase()
  }
}
```

When you search, you can use **`title.searchable`** and when you display in your application, you can use the **`title.display`** field:

```
> db.episodes.findOne(
  { "title.searchable": "the city on the edge of forever" },
  { "title.display": 1 }
)


{
  _id: 'tos_s01_e05',
  title: { display: 'The City on the Edge of Forever' }
}
```

While this does require some overhead when inserting or updating, it makes read queries both simple and efficient, especially if you put an index on the **`title.search`** field. If your application has a lot more reads than writes on a field, this may be the most straightforward way to deal with this issue. The very, very small amount of extra data makes essentially no difference in storage.

Conclusion

In this chapter, we learned all about query filters, how to use operators to query our data in various ways, and how to use projection to return just the parts of our documents we need as well as how to deal with casing issues.

In the next chapter, we will build upon these concepts and learn how to query and manipulate more complex data, such as arrays and objects as well as arrays inside of objects, and more!

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 7

Complex Data, Made Simple

"The improvement of understanding is for two ends: first, our own increase of knowledge; secondly, to enable us to deliver that knowledge to others."

— John Locke

"Let everything you say be good and helpful, so that your words will be an encouragement to those who hear them."

— Paul of Tarsus

Introduction

In the previous chapters, we have touched upon the concept of storing data in MongoDB using arrays and objects (or embedded documents) but not let us explore how we can query and maintain these more complex data types.

How exactly do you modify a particular element of an array? Is there a way to maintain array order? What sort of limitations might we run up against? In what way, can we remove some array elements while leaving others untouched? How do we "reach into" an embedded document to query on its fields and not the parent document's?

Structure

In this chapter, we will discuss the following topics:

- Arrays and Embedded Documents
- Querying Arrays
- Querying Embedded Documents
- Array Update Operators

Objectives

As we discussed in *Chapter 4, A Better Way to Store Data – Documents*, MongoDB's document model allows us to store much more complex data than legacy databases by using arrays and embedded documents. These data types give us a lot of flexibility, but with flexibility can come complexity. Fortunately, the MongoDB Query API provides robust tools for dealing with these complex types. By the end of this chapter, you should have a solid understanding of how to perform typical queries to find and modify arrays as well as embedded objects. Additionally, you will have a high-level understanding of the many MongoDB operators available for these data types, and how to use them.

Arrays and embedded documents

Using arrays and objects (or embedded documents), we can store pieces of data that belong with each other, together. The recipe documents we have been using are a prime example, and within them, we can store the directions in an ordered array, the ingredients as (embedded) documents with their own fields, individually in an array, and more.

A Note on quotes

In most of the examples in this book, you will see double quotes around a field's name like:

```
{ "count": 123 }
```

They are also present around string values, for example in an array like this:

```
{ "colors": ["red", "white", "blue"] }
```

These double quotes are not *always* strictly necessary, and indeed you may see some examples where a field name does not have double quotes around it, or where string

values in arrays have single quotes instead. This possibly confusing mix of notations has to do with differences between JavaScript (which the MongoDB Shell **`mongosh`** uses) and JSON (or more technically BSON). In JavaScript, you can express a string or an array in number of different valid ways. Take this example in Node.JS:

```
> var test = ["one", "two", "three"]
> console.log(test)
[ 'one', 'two', 'three' ]
```

We defined an array with double quotes, but when we log it out, it has single quotes because both are valid in JavaScript. However, in this next example, where we have a field name which begins with a number called **`2nd`**, we get an error:

```
> var object = { first: "1st", 2nd: 'second' }
var object = { first: "1st", 2nd: 'second' }
                             ^^

Uncaught SyntaxError: Invalid or unexpected token
```

The first field named **`first`** is valid, the second named **`2nd`** is not due to the leading number. If instead we make sure to wrap that field's name in quotes, it will now be valid (since it is now an explicit string):

```
> var object = { first: "1st", '2nd': 'second' }
> console.log(object)
{ first: '1st', '2nd': 'second' }
```

For this reason, along with other niche cases and JSON standards, it is a good habit to wrap fields names and strings in double quotes, so we will do that for most examples in this book.

Example Documents

We will be using the same example documents in our "cookbook" collection from *Chapter 6, Getting What You Want – Querying*, for this chapter, so you will not need to import any other documents. That said, there are some of the other example documents that are included in the book's git repo in the **`chapters/07`** folder if you are interested.

As a visual reminder, here is an example of the sort of documents we will be working with. They contain a recipe as well as other data using various types:

```
{
  "_id": { "$oid": "636aa9817dd21c28fda493a4" },
  "title": "Eggs Benedict",
  "calories_per_serving": 400,
  "prep_time": 4,
  "cook_time": 6,
  "ingredients": [
    {
      "name": "eggs",
      "quantity": { "amount": 6 },
      "vegetarian": true
    },
    {
      "name": "Virginia ham",
      "quantity": { "amount": 6, "unit": "rounds" },
      "vegetarian": false
    },
    {
      "name": "English muffins",
      "quantity": { "amount": 3 },
      "vegetarian": true
    },
    {
      "name": "Hollandaise sauce",
      "quantity": { "amount": 3, "unit": "oz" },
      "vegetarian": true
    }
  ],
  "directions": [
    "Cut the ham in rounds, split, toast and butter the english muffins,
to fit the muffins.",
```

```
    "Poach the eggs and place them on the ham and pour over the hollandaise
sauce."
  ],
  "rating": [5, 3, 5, 5, 5, 4, 5, 4, 5, 4, 5, 3, 4, 4],
  "rating_avg": 4.35,
  "servings": 3,
  "tags": ["ham"],
  "type": "Breakfast",
  "vegetarian_option": false
}
```

Querying arrays

Thus far, we have been focusing on so called scalar values such as strings, numbers, and decimals. Another data type, the array, provides us with an even more powerful and elegant way to store data, as well as some challenges with how to deal with them. We will cover the basics of how to query data within arrays, and some of the helpful operators available to us, and some key things needed to keep in mind when working with them.

Array Order Matters

An important thing to realize when using arrays in MongoDB is that the order of the items in the array matters. Consider the following queries:

```
> db.cookbook.find({ "tags": ["quick"] }).count()
0
```

From this simple query, it seems that there are no recipes that have the tag of "quick". However, if we change the query slightly, removing the array **[ ]**, we get a different count:

```
> db.cookbook.find({ "tags": "quick"}).count()
2
```

What is going on here? Is it zero documents, or two documents? Well, it depends on what exactly you are trying to do. By default, MongoDB will perform an *exact match*, meaning if you query for where:

```
{ "tags": ["quick"] }
```

MongoDB will look for documents that have an array *equal to* an array, with one item inside it, that has the value of "quick" as the single item. This is essential for the times when you need to find documents that match that array and only that array value. The order of the items in the array matter too; if they are the same values but in a different order, it will not be considered an exact match.

By comparison, if you do not use an array, but rather just a value and your array's name, this will match two documents since there are two documents that have an array named **`tags`**, with a value of "quick" *somewhere inside* that array.

Matching multiple array values

What about when you need to match multiple values in an array? You might wonder if you could do something like:

```
> db.cookbook.find({ "tags": ['quick', 'vegetarian']} ).count()

0
```

However, that will return zero documents because there are no documents with that exact value for the array **`tags`**. This is the same issue as before. If we change the query slightly however, using the **`$all`** operator, we will find multiple documents:

```
> db.cookbook.find({ "tags": { $all: ['quick', 'vegetarian'] } }).count()

2
```

This is because the **`$all`** operator will look for documents that meet all the condition of the **`$all`** array (not the **`tags`** array you are querying). To make this easier to understand, recall that the **`$all`** operator works as an "and" operation. So, this query could be composed slightly differently:

```
> db.cookbook.find(

{

  $and: [ { "tags": 'quick'}, { "tags": 'vegetarian' } ]

}).count()

2
```

Both versions of the query are valid, so ultimately, it comes down to what makes sense in your use case, or your personal preference.

Mixed data type arrays

You might be wondering what happens if you store more than one type of data in an array. Just like in JavaScript, this is perfectly fine. You can have an array with a whole mix of types like:

```
{ "mixed": [ 1, "two", false, { "sometimes": "taco" }, [3,2,1], "nice"] }
```

However, you probably do not want to mix types. Mixing types could end up complicating things later, or worse, you could end up with data integrity issues if you mix types like numbers and "strings as numbers" as in this array:

```
{ "numbers": [ 1, "2", "3", 4 ] }
```

While this might be fine when you read back this array's items and show them as strings on a web page, you cannot do operations on these numbers. If instead you keep your types consistent, you can use a lot of the same built-in operators that we discussed earlier. Let us explore that more deeply and illuminate why type consistency can be very powerful.

Arrays and query operators

Using field query operators with array data is often very straightforward. For example, most of our recipes have a field called **`rating`**, which is an array of numbers between **`1`** and **`5`**, and it might look something like this:

```
{ "rating": [3, 3, 4, 4, 5, 5, 3, 4, 4] }
```

If we wanted to find recipes that have any single instance of a rating of less than **`4`** in the **`rating`** field array, we could do a query like so:

```
> db.cookbook.find({ "rating": { $lt: 4} }).count()

5
```

This will look for documents that have at least one value of less than **`4`** in that array. You can also use the **`$elemMatch`** operator to get the same result:

```
> db.cookbook.find({ "rating": { $elemMatch: { $lt: 4} } }).count()

5
```

Using that same field, we can use the **`$size`** operator which works off the size of the array to find if any recipes have only one rating, or only three ratings:

```
> db.cookbook.find({ "rating": { $size: 1 } }).count()

1


> db.cookbook.find({ "rating": { $size: 3 } }).count()

0
```

In this case, one recipe has only one single rating, and zero have exactly three. Let us switch to a slightly different example. If we query for recipes with a **5** rating, we will get back these results:

```
> db.cookbook.find({ "rating": 5 }, { "_id": 0, "title": 1})
[
  { title: 'Split Pea Soup' },
  { title: 'Tomato Bisque' },
  { title: 'Zucchini Fudge Cake' },
  { title: 'Stuffed Peppers' },
  { title: 'Toast' },
  { title: 'Eggs Benedict' }
]
```

This is basically all our recipes. What if we wanted to know any recipes that have a **5** as the first rating in the array? We can do that by specifying a field and array index, as follows:

```
> db.cookbook.find({ "rating.0": 5 }, { "_id": 0, "title": 1, "rating": 1
})
[
  { title: 'Zucchini Fudge Cake', rating: [5,3,5,5,5,5,5,5,5] },
  { title: 'Toast', rating: [5] },
  { title: 'Eggs Benedict', rating: [5,3,5,5,5,4,5,4,5,4,5,3,4,4] }
]
```

The **rating.0** means the first, or zero index item in the **rating** array and is also an example of why we generally are using quotes around field names.

Querying embedded documents

There are several ways to describe fields with object data inside them, embedded documents, nested documents, or simply objects. Throughout this book, we will often use the generic term "object", but it is important to understand these terms are synonymous.

Exact matches

Much like the array examples, there is a difference between "exact matches" and other sorts of matches for embedded documents. Given our recipe documents, see

how these two queries differ when trying to find recipes with the ingredient of "pepper":

```
> db.cookbook.find({ "ingredients": { "name": "pepper" } }).count()
0


> db.cookbook.find({ "ingredients.name": "pepper" }).count()
3
```

In the first query, an *exact match* is required, which is not found. However, in the second query, we have used "dot notation" to "reach inside" the embedded document and specifically asked for documents where there is a field called **`name`** inside **`ingredients`**, with a value of "pepper". This matches three documents. This is sometimes referred to as matching an "embedded field".

Matching inside embedded documents

If you look at the embedded documents inside **`ingredients`**, you will see that some have their own embedded documents which describe the **`quantity`** of the ingredient:

```
{
  "name": "Hollandaise sauce",
  "quantity": { "amount": 3, "unit": "oz" },
  "vegetarian": true
}
```

To query on the value of **`unit`**, which is embedded inside the embedded **`ingredient`** field, you can use the dot syntax again:

```
> db.cookbook.find({ "ingredients.quantity.unit": "oz" }).count()
2
```

From this, we can tell that two documents use the **`unit`** of **`oz`** (ounces) somewhere in them; we can nicely see exactly which recipes matched by using the **`projection`** parameter:

```
> db.cookbook.find({ "ingredients.quantity.unit": "oz" }, { "title": 1,
"_id": 0 })


[
```

```
  { title: 'Zucchini Fudge Cake' },
  { title: 'Eggs Benedict' }
]
```

The **`projection`** also accepts dot notation, and here, we will retrieve just the **`title`** *and* each ingredient **`name`** for our Eggs Benedict recipe:

```
> db.cookbook.findOne(
  { "title": "Eggs Benedict" },
  { "title": 1, "_id": 0, "ingredients.name": 1 }
)


{
  title: 'Eggs Benedict',
  ingredients: [
    { name: 'eggs' },
    { name: 'Virginia ham' },
    { name: 'English muffins' },
    { name: 'Hollandaise sauce' }
  ]
}
```

You can also use **`$elemMatch`** with different operators in the **`projection`**. Refer to the official documentation for more.

Array update operators

More than likely, you will need to update and maintain the array fields you have, either because you need to add more items, remove them or move items around within the array. In this section, we will consider each of these actions.

Optional example documents

While we will continue the use of example documents in the **`cookbook`** collection in other portions of this book, in this section we will switch to a different collection. If you would like to follow along in your MongoDB instance, feel free to import the example documents from the **`chapters/07/playground.json`** file from the book's git repository.

These documents use the same format as our recipe example documents, but we will import them into a collection called **`playground`** to make sure we do not accidently modify our **`cookbook`** documents.

Adding array items

When you need to modify the items in an array, there are a handful of operators available which generally track with the names of common array methods in various programming languages.

If a user on our fictional cooking website added a new rating for one of our recipes, we will want to append that rating to our **`rating`** array field which looks something like this:

```
rating: [ 3, 4, 2, 5, 3, 4, 4]
```

To do this, we will use one the most common array operators, **`$push`**, which adds an item to an array. If the recipe we wanted to update had an **`_id`** of **`1`**, we could do a query as follows:

```
> db.playground.updateOne({ "_id": 1 }, { $push: { rating: 4 } } )
```

We could also use **`updateMany()`** and either pass multiple document matchers, for example multiple **`_id`**, or any empty matching document **`{ }`**, which will update all documents in the collection. Using a query like this will add a new tag "**`free`**" to all documents in the cookbook collection:

```
> db.playground.updateMany({ }, { $push: { "tags": "free" } } )
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 3,
  modifiedCount: 2,
  upsertedCount: 0
}
```

Each time you run the query, it will add the item again to the array, always at the end. If the **`tags`** field does not exist, it will add that field to the document for you using an upsert.

Appending multiple items

If you have more than one item to add to an array, you will need to use a slightly different syntax. You might think you could use an array to append multiple items like this:

```
> db.playground.updateMany({ }, { $push: { "tags": ["new", "hot"] } })
```

However, this will add a new *array* as a value to the array instead:

```
{ "tags": [ "free", ["new", "hot"] ] }
```

To add more than one item, for example, two tags at one time, we will use the **`$each`** operator:

```
> db.playground.updateMany({ }, { $push: { "tags": { $each: ["new", "hot"]
} } })
```

Using this syntax, we will have a tags array field that looks like:

```
{ "tags": [ "free", "new", "hot" ] }
```

Sorting array items

Sometimes the exact order of an array's items are important. If this is the case, you will want to use the **`$sort`** operator which will make sure that the order array is maintained on inserts and updates. For example, if we had a new field called **`featuring`** with the following values:

```
{ "featuring": ["apples", "cinnamon", "sugar"] }
```

We can use **`$sort`** as an additional parameter, when we add the two new items to the array, "zucchini" and "blueberries":

```
> db.playground.updateOne(
  { _id: 1 },
  {
    $push: {
      "featuring": {
        $each: ["zucchini", "blueberries"],
        $sort: 1
      }
    }
  })
```

Now our array would look like this:

```
{ "featuring": ["apples", "blueberries", "cinnamon", "sugar", "zucchini"]}
```

Much like sorting a query, **1** means ascending and **-1** means descending. You can also use an empty array **[ ]** here if all you want to do is sort, and not modify:

```
> db.playground.updateOne(
  { _id: 1 },
  {
    $push: {
      "featuring": {
        $each: [ ],
        $sort: -1
      }
    }
  })
```

Perhaps you want to make sure the "**most recently added**" tag is always the first item in our **tags** array. To do this, we can use the **$position** operator, along with the **$each** operator:

```
> db.playground.updateOne(
  { _id: 1 },
  {
    $push: {
      "tags": {
        $each: ["red", "blue"],
        $position: 0
      }
    }
  })
```

The **$position** corresponds to a position in the array, so if you set **0**, it will add the item, or items to the start of the array, and if you set **-1**, it will insert at the position *just before* the last item in the array. By default, **$push** will add items to the end of the array.

Removing array items

There are times when you might want to constrain or reduce the number of items in your array. Perhaps you want to store a convenience field called **last_four_**

ratings which you will use to store the last four user ratings, which you will show as a preview on your website.

For these situations, we can use **$slice**. Imagine we had a **last_four_ratings** array field that looked like:

```
{ "last_four_ratings": [ 2, 3, 3 , 4 ] }
```

If we wanted to add two new **5** star ratings to the **last_four_ratings** array (while also making sure *only four items* are ever in the array), we could run a query as follows:

```
> db.playground.updateOne(
  { _id: 1 },
  {
    $push: {
      "last_four_ratings": {
        $each: [5, 5],
        $position: 0,
        $slice: 4
      }
    }
  })
```

Here, we have two values of **5** for **$each** and a value of **4** for **$slice**, which will make sure the array (after the update) has only four items in it. Lastly, we set **$position** to **0** which will insert our new values *at the front* of the array, resulting in a final array like this:

```
{ last_four_ratings: [ 5 , 5, 2, 3 ] }
```

A negative number for **$slice** or **$position** would have the opposite effect position wise. You can also use **$slice** to cut down the number of items in an array without updating or adding anything new by passing an empty array to **$each**:

```
> db.playground.updateOne(
  { _id: 1 },
  {
    $push: {
      "last_four_ratings": {
        $each: [ ],
```

```
        $slice: 4
      }
    }
  })
```

Aside from **`$slice`**, there are two more commonly used ways to remove items from an array: **`$pop`** which will remove the first or the last element of an array and **`$pull`** which will accept a query to match items you wish to remove from the array.

Referencing an earlier addition to our recipe document, imagine we have a **`featuring`** array which looks like the following:

```
{ "featuring": ["apples", "blueberries", "cinnamon", "sugar", "zucchini"]}
```

If we want to simply remove the *last item* in the array, we can use **`$pop`** and **`1`**. Conversely, to remove the *first item* in the **`featuring`** array, we can construct a query like this using **`$pop`** and **`-1`**:

```
> db.playground.updateOne( { _id: 1 }, { $pop: { "featuring":  -1 } })
```

To confirm the removal, we can use the **`projection`** to just return the **`featuring`** array of the document:

```
> db.playground.findOne({ "_id": 1 }, { "_id" : 0, "featuring" : 1 } )


{ _id: 1, featuring: [ 'blueberries', 'cinnamon', 'sugar', 'zucchini' ] }
```

To perform a more targeted removal, we can use **`$pull`** and a matching query. This query can be a simple match, or it can a use query operator such as **`$gte`** or **`$in`**, and so on, as follows:

```
> db.playground.updateOne(
  { _id: 1 },
  { $pull: { "featuring": { $in: ["blueberries", "zucchini"] } } }
)
```

Again, confirming the removal, we now just have "cinnamon" and "sugar" left in the array:

```
> db.playground.findOne({ "_id": 1 }, { "_id": 0, "featuring" : 1 } )


{ featuring: [ 'cinnamon', 'sugar' ] }
```

You can also use **$pullAll** to remove any instance of a certain value from an array, without a query. Here we remove "sugar" from our array leaving only "cinnamon" left:

```
> db.playground.updateOne( { _id: 1 }, { $pullAll: { "featuring": ["sugar"]
} })


> db.playground.findOne({ "_id": 1 }, { "_id": 0, "featuring" : 1 } )


{ featuring: [ 'cinnamon' ] }
```

There are handful of other operators and ways of using **$** with arrays. Check out the official MongoDB documentation for more on that if the need arises.

Conclusion

While dealing with more complex data types like arrays and embedded documents might be a little intimidating at first, hopefully now you know MongoDB has robust support for these types and you can be empowered to use them whenever you need to! On the other hand, perhaps, this really excites you and you can see exactly how the MongoDB Document Model and Query API will allow you to store data, exactly how you need to for your application.

Either way, you should be excited for the next chapter. We will discuss various strategies for structuring your documents, and how that structure can affect performance, data integrity and usability.

Chapter 8

The MongoDB Aggregation Framework

"The same set of statistics can produce opposite conclusions at different levels of aggregation."

— Thomas Sowell

"All the world's a stage. And all the men and women merely players; they have their exits and entrances; and one man in his time plays many parts."

— William Shakespeare, As You Like It

Introduction

Even the best thought out schema cannot handle every single use case or need that your application may have, and even with a powerful query engine, you may find at times, that you cannot quite get the results you need. For more complex cases, MongoDB offers what is called the Aggregation Framework, which allows a structured way to formulate a series of steps, called a "pipeline", to get back just the data you need. In this chapter, we will discuss this framework and dive into some examples of its use.

Structure

In this chapter, we will discuss the following topics:

- Typical Aggregation Pipelines

- Complex Pipelines
- Additional Uses

Objectives

Covering the Aggregation Framework in-depth is beyond the scope of this book. However, by the end of this chapter, you should have a solid idea of how the framework works and how you can use it to fit your needs.

Typical aggregation pipelines

The stages of a pipeline are expressed in operators, like the query operators or query parameters (like the **`projection`**), that we have learned about so far. However, the Aggregation Framework adds many more options and additional operations, which allow you to get the exact results you need.

Pipeline stages

While there is a long list of distinct stage types you can use in a pipeline, there are generally four categories of actions that these stages perform: **filtering**, **transforming**, **grouping,** and **sorting**. The exact order of stages generally does not matter as MongoDB's query optimizer will attempt to ensure each step processes at the best time. However, there are some general rules of thumb we will touch on later.

To compose your pipeline, try and work out what you need in a result and then break it up into steps. Perhaps a first stage narrows down the number of documents you are processing, by checking if they meet a condition, like a field being equal to a certain string value, or that a different field is greater than or equal to a certain number. The next stage might sort the documents by their title field, while another stage might control which fields come back in the final query.

Building a pipeline

Each stage of the aggregation pipeline goes into an array, and you can typically use the same stage type multiple times, if needed. Instead of using a command such as **`find()`** to run a query, we will use **`aggregate()`** like so:

```
> db.collection.aggregate([ stageOne, stageTwo ])
```

You can have a pipeline with as little as a single stage, or as many as 1,000. The main limiter is memory, which is limited to 100 megabytes unless you allow the query to use disk (which is considerably slower).

Projecting aggregated fields

In prior examples, we have used the **`project`** parameter to control which fields come back in the results of the query. Using the Aggregation Framework, we can use the **`$project`** stage to not only control which fields come back but also compose or aggregate fields together into new fields.

For the following examples, we will again use the documents in our **`cookbook`** collection. As you will recall, the recipe documents contain a **`rating`** array, which looks something like:

```
rating: [ 4, 2, 3, 3, 4, 5, 1, 2 ]
```

If we want to return the average rating for each recipe, we can use two operators: first, **`$project`** which defines the stage and then **`$avg`** to calculate the average of the numbers in the **`rating`** array. We can then assign this result to a new field called **`avgRating`**, using query like this:

```
> db.cookbook.aggregate([
{
  "$project": {
    "avgRating": { "$avg": "$rating" }
  }
}
])
```

Notice that we told MongoDB which field to average from the document by putting it in quotes and using a **`$`**, so **`$rating`** to use the **`rating`** field. This will return a result for each document with the **`_id`** as well as a new field **`avgRating`** containing the average:

```
[
  { _id: ObjectId("636821387dd21c28fda4939f"), avgRating: 3.7142857142857144 },
  { _id: ObjectId("636aa92f7dd21c28fda493a0"), avgRating: 3.888888888888889 },
  { _id: ObjectId("636aa94c7dd21c28fda493a1"), avgRating: 4.777777777777778 },
  { _id: ObjectId("636aa9617dd21c28fda493a2"), avgRating: 3.888888888888889 },
  { _id: ObjectId("636aa9707dd21c28fda493a3"), avgRating: 5 },
  { _id: ObjectId("636aa9817dd21c28fda493a4"), avgRating: 4.357142857142857 },
  { _id: ObjectId("636ab56e956f91c56f02f049"), avgRating: null }
]
```

This is purely the output of the query. The underlying documents have not been modified.

Note: In most examples, you will see quotes around fields, but technically you can run these queries without the quotes except for the field we are working with, so here rating or "$rating". Therefore, a query formatted like this is also valid:

```
db.cookbook.aggregate([
{
  $project: {
    avgRating: { $avg: "$rating" }
  }
}
])
```

However, writing your queries like this makes them invalid JSON, which can make checking your query syntax and other things more difficult. So, as previously discussed, this book generally prefers quotes.

Aggregations in MongoDB Compass

The MongoDB Compass UI has built-in support for using Aggregation and building pipelines. It breaks out each step into a graphical stage and sometimes is also able to show you previews of what your pipeline's data currently looks like. To use Aggregation in Compass, browse to a Collection and press the **Aggregations** tab at the top right under the collection's name, as shown in *Figure 8.1:*

Given our example that gets an average of the ratings for a recipe, the same query might look like *Figure 8.1* within the Compass UI:

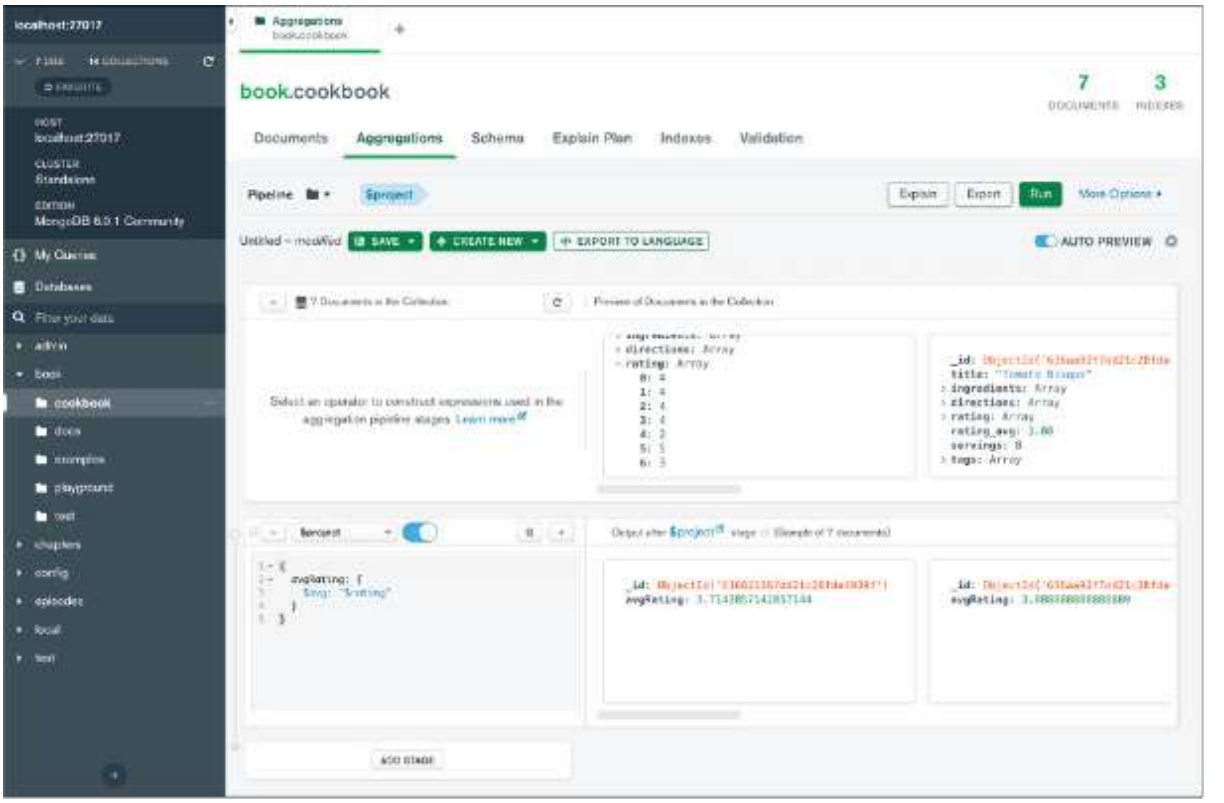


Figure 8.1: *Using Aggregation in MongoDB Compass*

You can use the **Add stage** button to compose new stages and drag and drop them around to build up your pipeline. The UI will show you different stage operators you can use, do some sanity checking on your queries and show a preview (if possible), as seen in *Figure 8.2*:

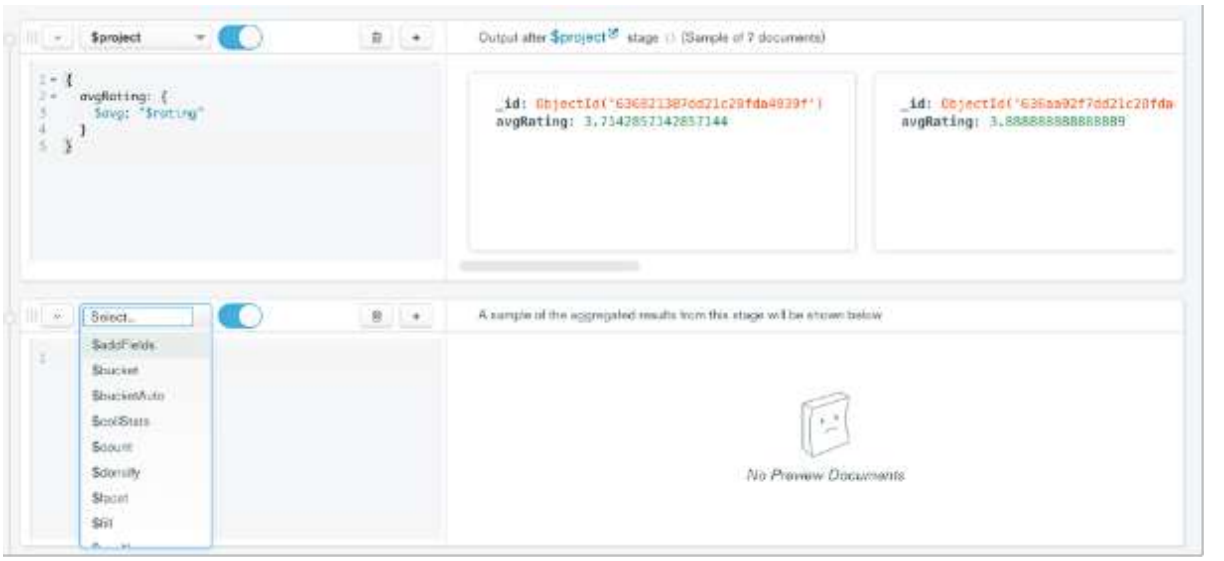


Figure 8.2: *Adding Aggregation stages in MongoDB Compass*

Note that within the Compass UI, the "stage" operator (**`$project`** in this case), is set via the dropdown and *not within* the query document that goes inside the editable box under it. So instead of a document like this:

```
{
  "$project": {
    "avgRating": { "$avg": "$rating" }
  }
}
```

In Compass, you would select **`$project`** from the dropdown and then pass in a smaller document like this:

```
{
  "avgRating": { "$avg": "$rating" }
}
```

Beyond that, the queries are composed the same way.

MongoDB compass pipeline features

Since these types of queries can get rather complicated, be sure to make use of the **SAVE** and **Explain** features as well as **EXPORT TO LANGUAGE**, as seen in *Figure 8.3*:

Figure 8.3: *Save and Export buttons in MongoDB Compass*

Writing a query in **`mongosh`** or Compass is very helpful if you just need the results of the query for some general purpose. However, to use these aggregation queries in an application, they will need to be converted to a syntax that your language can understand. Luckily, Compass has a great feature which will do this conversion for you!

To export your current aggregation query into your favorite programming language, press the **EXPORT TO LANGUAGE** button and choose your language, as seen in *Figure 8.4*:

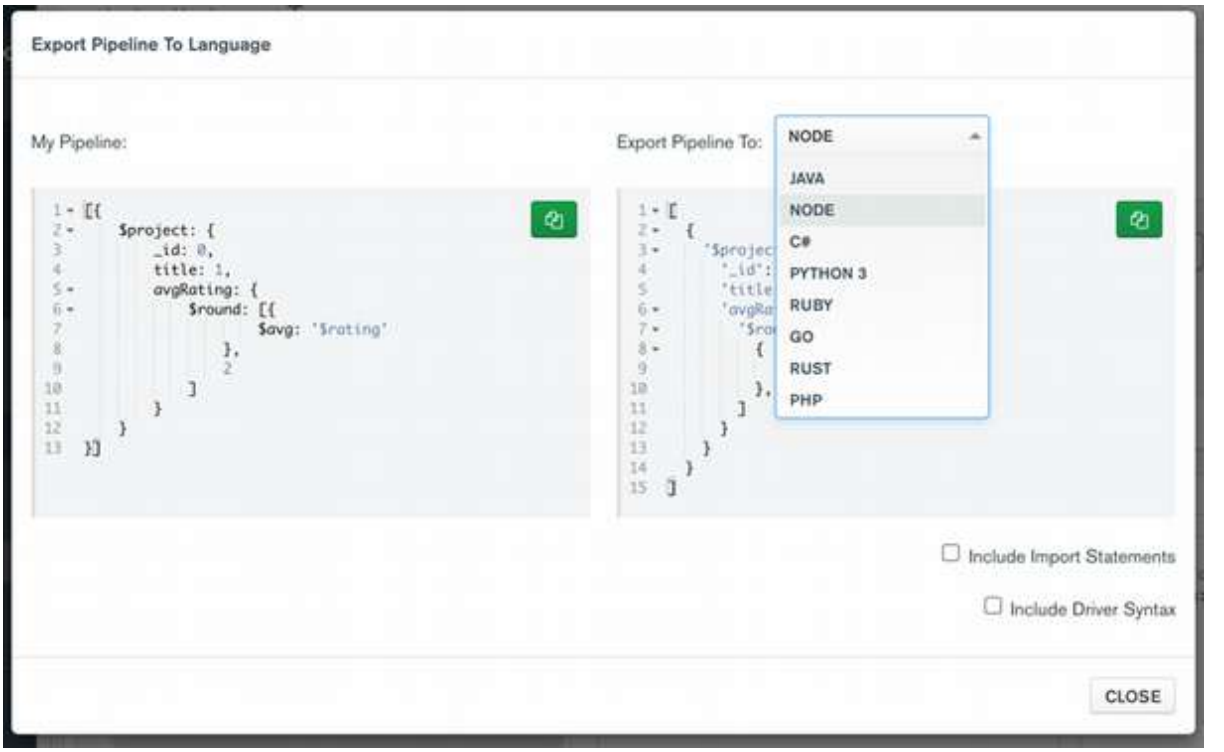


Figure 8.4: *MongoDB Compass Export to Language Dialog*

Combining operators

We can make these results a little more appetizing by tweaking our **`$project`** a little bit, to hide and add fields as well as manipulate our output further:

```
> db.cookbook.aggregate([

{
  "$project": {
    "_id": 0,
    "title": 1,
    "avgRating": {
      "$round": [ { "$avg": "$rating"}, 2]
    }
  }
}
])
```

Here we have used another operator **`$round`** and asked for numbers to be limited to 2 decimal spaces, as well as removing the **`_id`** and adding the **`title`** field. Now our results will be more suitable for use in our application, and more human readable:

```
[
  { title: 'Split Pea Soup', avgRating: 3.71 },
  { title: 'Tomato Bisque', avgRating: 3.89 },
  { title: 'Zucchini Fudge Cake', avgRating: 4.78 },
  { title: 'Stuffed Peppers', avgRating: 3.89 },
  { title: 'Toast', avgRating: 5 },
  { title: 'Eggs Benedict', avgRating: 4.36 },
  { title: 'Blue Cheese Burgers', avgRating: null }
]
```

Or the same update query in the MongoDB Compass UI, here in *Figure 8.5*:

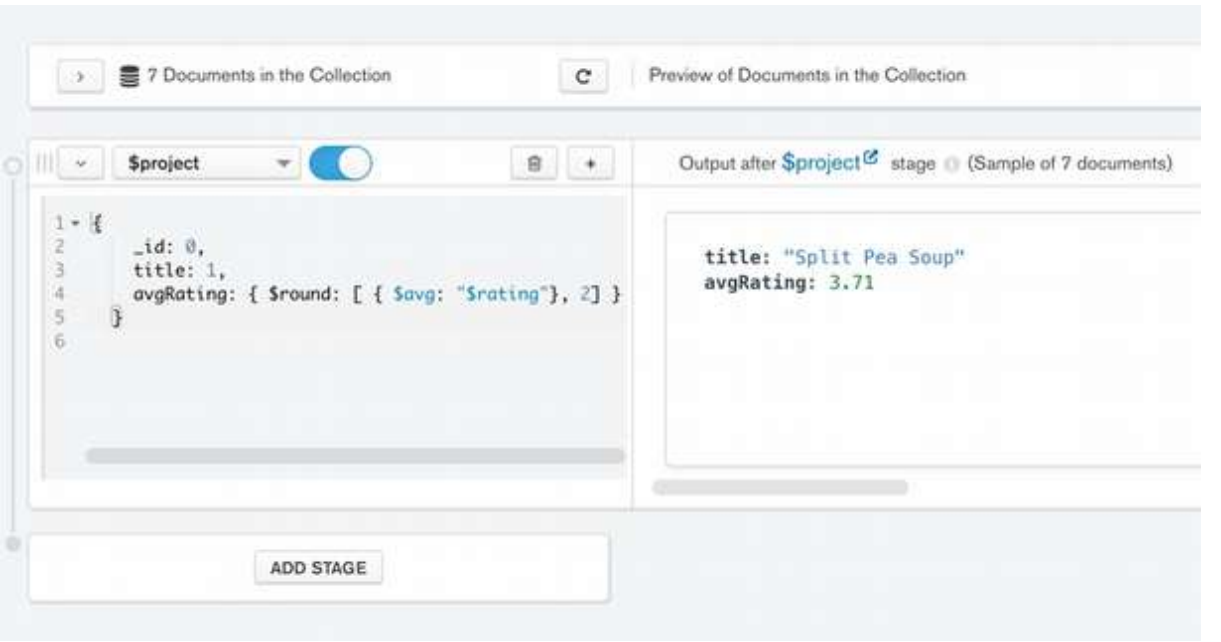


Figure 8.5: *MongoDB Compass # project stage*

You will notice the recipe for "Blue Cheese Burgers" has a value of **`null`** for **`avgRating`**. This is because the **`rating`** field does not exist within that document, as it does not yet have any ratings. We will discuss options to deal with that in the next section.

Filtering matching documents

There are many ways to filter documents using the pipeline but one of the simplest and most useful is the **`$match`** stage. Using the **`$match`** stage, we can filter out documents we do not want to pass onto the rest of our pipeline, by using standard MongoDB queries. Simply add a new stage document with the **`$match`** operator and your matching (or unmatching) query to the pipeline array:

```
> db.cookbook.aggregate([
{
  "$match": {
    "rating": { "$exists": true }
   }
},
{
  "$project": {
    "_id": 0,
    "title": 1,
    "avgRating": {
      "$round": [{ "$avg": "$rating"}, 2]
    }
  }
}
])
```

This will give us the same output as before, but excluding the "Blue Cheese Burger" document, since it does not have the **`rating`** array field. Or we could look for only "Breakfast" **`type`** recipes that have a vegetarian option, using the **`vegetarian_option`** boolean field:

```
> db.cookbook.aggregate([
{
  "$match": {
    "rating": { "$exists": true },
    "type": "Breakfast",
    "vegetarian_option" : true
```

```
  }
},
{
  "$project": {
    "_id": 0,
    "title": 1,
    "avgRating": {
       "$round": [{ "$avg": "$rating"}, 2]
    }
  }
}
])
```

This will end up with only recipe:

```
[ { title: 'Toast', avgRating: 5 } ]
```

Within the Compass UI, you will see each stage in their own group, as seen in *Figure 8.6*:

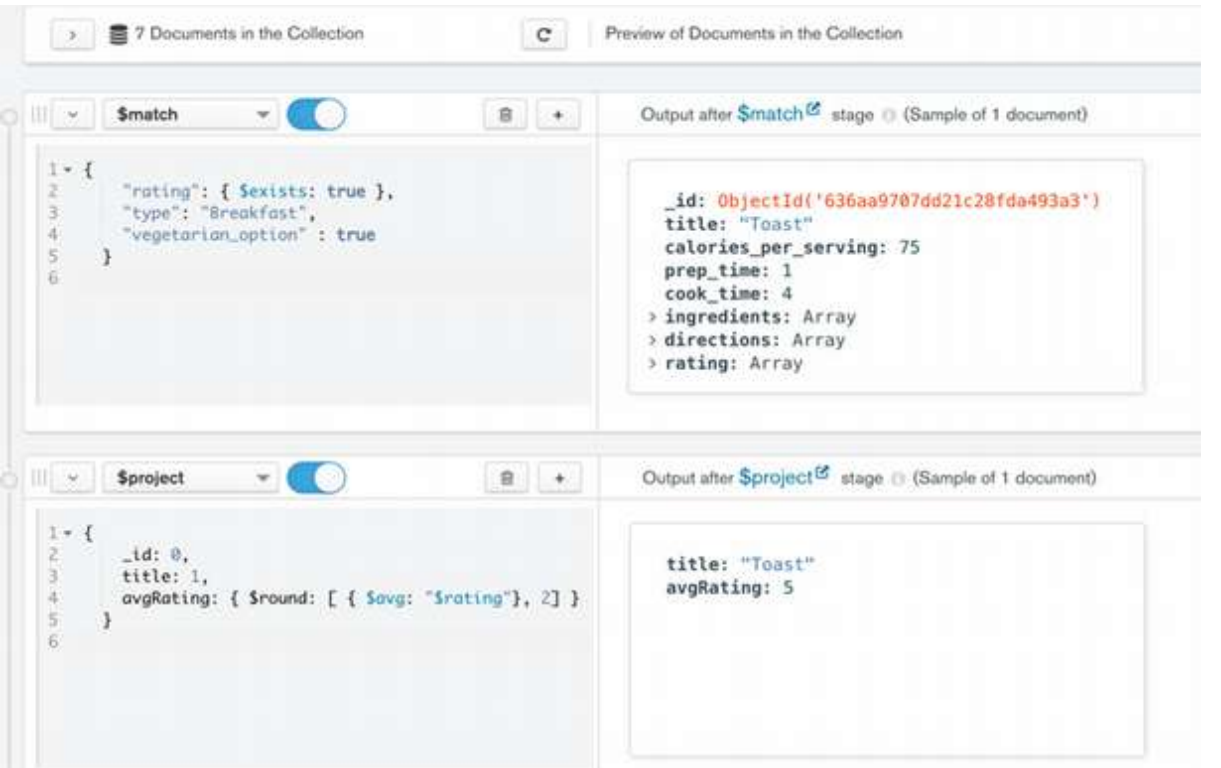


Figure 8.6: *$match and $project stages in Compass*

In newer versions of MongoDB Compass, you can use the "switch to text" option to view and edit your pipeline in a text editor as in *Figure 8.7*:

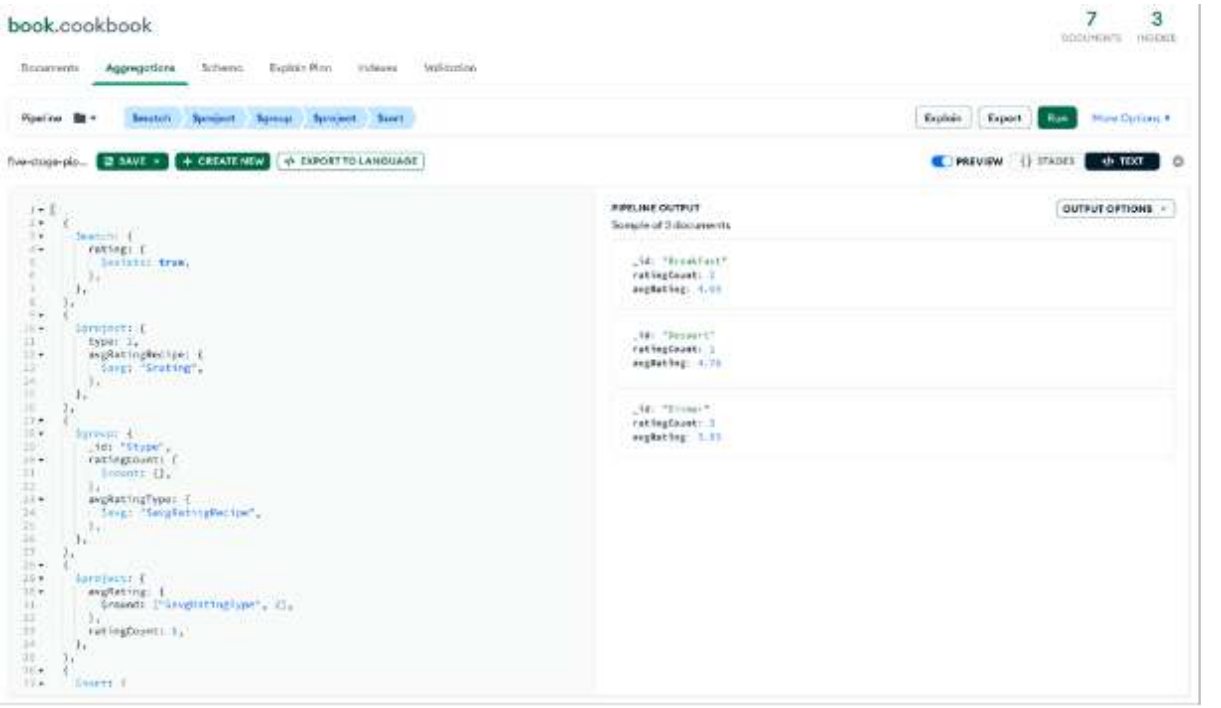


Figure 8.7: *Pipeline Text Editor in MongoDB Compass*

Using stages multiple times

You can add **`$match`** or just about any stage multiple times, so the same results can be achieved by two **`$match`** stages:

```
> db.cookbook.aggregate([
{
  "$match": {
    "rating" : { "$exists": true },
    "type": "Breakfast"
  }
},
{
  "$match": {
    "vegetarian_option" : true
  }
```

```
},
{
  "$project": {
    "_id": 0,
    "title": 1,
    "avgRating": {
      "$round": [ { "$avg": "$rating"}, 2]
    }
  }
}
])
```

While using stages multiple times is very powerful, and sometimes the only way to get the results you need as you will see later, this can cause unanticipated results if you do not account for the stages properly, so do be careful while doing this.

Making large pipelines more readable

Adding on more and more stages can at some point become unwieldy to read, even if MongoDB's query engine is not fazed at all. You can run all your pipelines within MongoDB Compass to somewhat avoid this issue, or you can opt to break up your stages into variables as we did with larger **`find()`** queries earlier.

For example, we can break up our **`$match`** stage and **`$project`** stage into two separate variables:

```
> var matchRecipes = {
  "$match": {
    "rating" : { "$exists": true },
    "type": "Breakfast",
    "vegetarian_option" : true
  }
}


> var projectRecipes = {
  "$project": {
```

```
    "_id": 0,
    "title": 1,
    "avgRating": {
      "$round": [ { "$avg": "$rating"}, 2]
    }
  }
}
```

After we have these variables defined, we can use them as stages in our pipeline (and reuse them as needed):

```
> db.cookbook.aggregate([
  matchRecipes,
  projectRecipes
])
```

Using this approach, we can reduce complex, long pipelines into almost "readable sentences" if we name our variables well. Unfortunately, the Compass UI does not support this method.

Grouping and sorting stages

You can also use **`$avg`** and other operators as part of another stage called **`$group`**, which works somewhat like **`GROUP BY`** in SQL databases. A simple example might be grouping by a recipe's **`type`** and getting a count of how many recipes match each **`type`** and then sorting by **`type`** alphabetically:

```
> db.cookbook.aggregate([
{
  "$match": {
    "rating": { "$exists": true }
  }
},
{
  "$group": {
    "_id": "$type",
    "recipeCount": { "$count": {} }
  }
```

```
},
{
  "$sort": { "_id": 1 }
}
])
```

This will give us the following counts:

```
[
  { _id: 'Breakfast', recipeCount: 2 },
  { _id: 'Dessert', recipeCount: 1 },
  { _id: 'Dinner', recipeCount: 3 }
]
```

We can view this a little more clearly in MongoDB Compass, as shown in *Figure 8.8*:

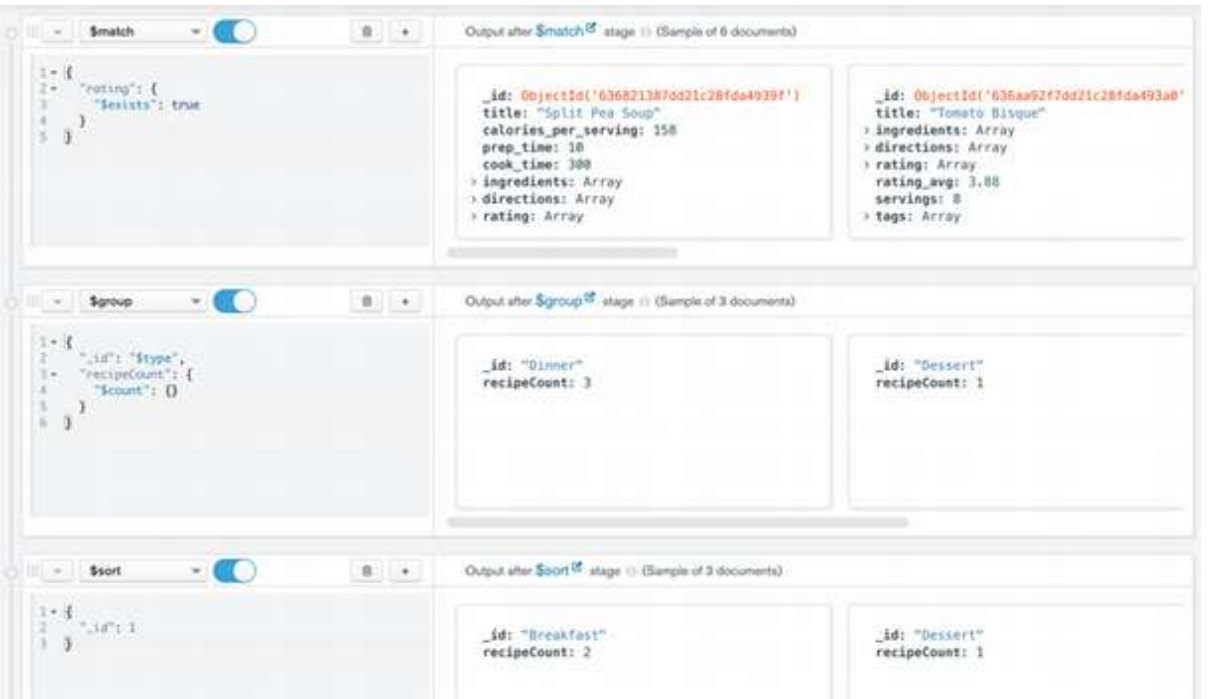


Figure 8.8: *$match, $group and $sort stages in Compass*

Within **`$group`** we assign **`_id: "$type"`**, which is telling MongoDB what field we want to "group by", so here, we are grouping by the **`type`** field which is why the results now have a **`_id`** equal to the **`type`** field.

The document's individual **`_id`** fields are no longer relevant with these query results:

```
{
  "$group": {
    "_id": "$type",
    "recipeCount": { "$count": {} }
  }
},
```

Note the **`$count`** operator here is matching any document **`recipeCount: { $count: {} }`** in that group, but it could also contain its own match, like the count of documents of that **`type`** that have **`prep_time`** of less than 60 minutes.

We added a **`$sort`** stage because **`$group`** *does not sort*, it only groups; so, if the order matters, make sure to include a **`$sort`** stage.

Complex pipelines

Getting a good deal more complex, we can mix stages multiple times to get interesting and useful results. Perhaps we have been asked to make a new section of our website that takes recipes, groups them by their type and give users some stats about each type. In the end, we would like a query result that looks something like this:

```
[
  { _id: 'Breakfast', recipeCount: 2, avgRating: 4.68 },
  { _id: 'Dessert', recipeCount: 1, avgRating: 4.78 },
  { _id: 'Dinner', recipeCount: 3, avgRating: 3.83 }
]
```

To break down our needs, we want:

- All recipes grouped by **`type`**.
- A **`count`** of recipes that are of each type.
- The **`average`** of the "average ratings" per recipe, for recipes within that **`type`**.
- Ratings should be averaged to *at most* two decimal places.
- Recipes without ratings should be excluded.
- Final results ordered by the **`type`**, alphabetically.

The third point might be a little confusing. In the example documents, we have created a **`rating_avg`** field that has *precomputed* the average of that recipe's **`rating`** array. In this example, we are assuming that either the field does not exist or it is not up to date. So, we need to get the *average for that recipe's ratings*, and then use that number *from each recipe* to get the *average rating for each recipe type*.

Create a complex pipeline

Take a moment to see if you can construct this query yourself from what we have learned so far. Remember you can use stages multiple times, and you probably do not want to construct your stages in exactly the order in the bullet points above.

For this example, we have included a **`complex-pipeline.json`** file you can use via **`mongosh`** or import into Compass UI in the **`chapters/08`** folder of the book's git repository.

You can import **`complex-pipeline.json`** into MongoDB Compass by pressing the **+ CREATE NEW** button and picking the Pipeline from text option, as shown in *Figure 8.9*:

Figure 8.9: *Create New pipeline button*

This will open a modal within which you can paste the contents of the pipeline array in the **`complex-pipeline.json`** file, as shown in *Figure 8.10*:

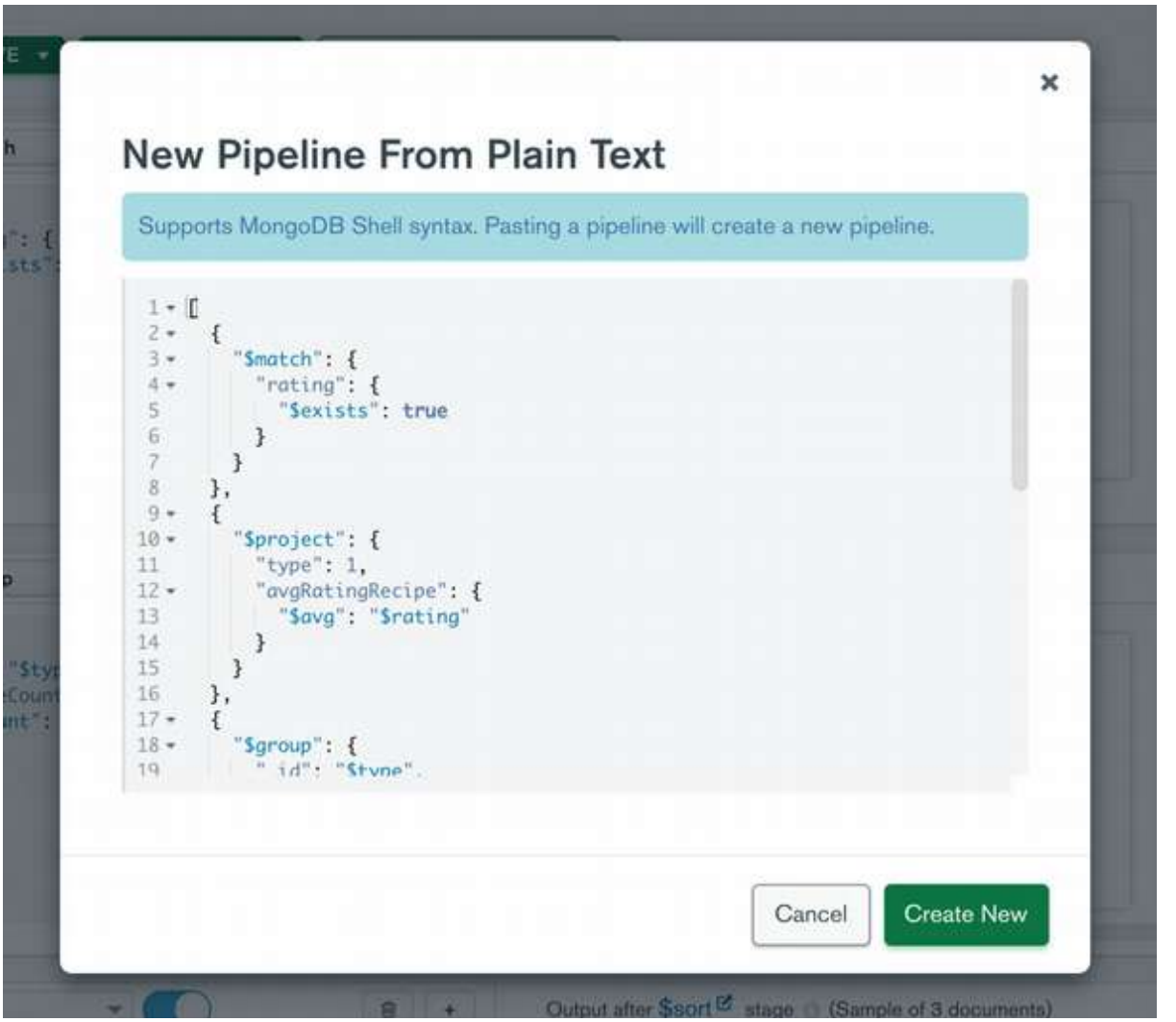


Figure 8.10: *Import new pipeline in MongoDB Compass*

Complex Pipeline Example

Here is one possible solution to get the results we need; compare it to what you came up with:

```
> db.cookbook.aggregate([
// stage one
  {
    "$match": {
      "rating": { "$exists": true }
    }
  },
```

```
// stage two
  {
    "$project": {
      "type": 1,
      "avgRatingRecipe": { "$avg": "$rating" }
    }
  },
// stage three
  {
    "$group": {
      "_id": "$type",
      "recipeCount": { "$count": {} },
      "avgRatingType": {
        "$avg": "$avgRatingRecipe"
      }
    }
  },
// stage four
  {
    "$project": {
      "avgRating": {
        "$round": ["$avgRatingType", 2]
      },
      "recipeCount": 1
    }
  },
// stage five
  {
    "$sort": { "_id": 1 }
  }
])
```

How did you do? Were you able to get the results you needed? Did you use a different method or set of stages? Are you still feeling a little bit confused or stumped? No need to worry; we will break down this pipeline stage by stage in the next section.

Stages of a complex pipeline explained

Let us break up these stages a little so we can understand better what is happening.

Stage one

First, we are matching just the documents we want, the ones with ratings. We did this first so that we are only working with the documents we need going forward. In a larger collection, this could provide a big performance improvement since MongoDB can disregard and exclude any documents that would be irrelevant to our end result:

```
{
  "$match": {
    "rating": { "$exists": true }
  }
},
```

Stage two

Then we are defining what fields we want to project for the next stage in the pipeline. These fields include the **`type`** and average of the **`rating`** array for each recipe, a field we called **`avgRatingRecipe`**:

```
{
  "$project": {
    "type": 1,
    "avgRatingRecipe": { "$avg": "$rating" }
  }
},
```

Stage three

Next, we are grouping by the **`type`**, getting a count that we called **`recipeCount`** and then *averaging the average* ratings for each document or the **`avgRatingRecipe`**, grouped by **`type`** into a field we called **`avgRatingType`**:

```
{
  "$group": {
    "_id": "$type",
    "recipeCount": {
      "$count": {}
```

```
    },
    "avgRatingType": {
      "$avg": "$avgRatingRecipe"
    }
  }
},
```

Stage four

In this stage, we used **`$project`** again, but this time, to define that we want back a *rounded* value for **`avgRatingType`** and to include the **`recipeCount`** in our final results:

```
{
  "$project": {
    "avgRating": {
      "$round": ["$avgRatingType", 2]
    },
    "recipeCount": 1
  }
},
```

Stage five

Lastly, we sort by the type (which is now our **`_id`**) alphabetically:

```
{
  "$sort": { "_id": 1 }
}
```

This will give us a result of each **`type`**, in alphabetic order, with the average of the recipes in that **`type`** as well as a count:

```
[
  { _id: 'Breakfast', recipeCount: 2, avgRating: 4.68 },
  { _id: 'Dessert', recipeCount: 1, avgRating: 4.78 },
  { _id: 'Dinner', recipeCount: 3, avgRating: 3.83 }
]
```

There are of course multiple ways to solve this problem. Maybe even yours was different! Take some time to experiment and see if there are some other ways you can think of solving this challenge.

Additional uses

There are some other uses of the Aggregation Framework that might not be immediately apparent. For example, do you remember that issue with case sensitivity and MongoDB queries we encountered earlier? How might we use the Aggregation Framework to alleviate that issue?

Case insensitive searching and sorting

MongoDB's searching is case sensitive by default, and so is its sorting. There are ways to store data in your documents to avoid this issue, as we discussed before, but we can also use the Aggregation Framework to solve this problem using the **`$addFields`** stage. This stage works a lot like how we used **`$project`** in prior examples, but it is a bit more robust and persists the new field throughout our pipeline.

Using the **`$addFields`** stage, we could change our search and sorting, to be case insensitive. First, we will start by adding a field that has our recipe **`title`** in lowercase:

```
> db.cookbook.aggregate([
{
  "$addFields": {
    "search_title": {
      "$toLower": "$title"
    }
  }
},
{
  "$match": { "search_title": "toast" }
},
{
  "$project": { "title" : 1}
}
])
```

Note the **`$match`** stage is searching for "toast"; this will still return a match for "Toast", even though we searched for a lowercase "toast" thanks to our new field **`search_title`**. we composed using the **`title`** field **`"$title"`** and the **`$toLower`** operator:

```
{
  "$addFields": {
    "search_title": {
       "$toLower": "$title"
    }
  }
}
```

The results of this pipeline will be as follows:

```
[ { _id: ObjectId("636aa9707dd21c28fda493a3"), title: 'Toast' } ]
```

Note: If we did not add the last `$project` stage specifying only the title field, we would get the whole document back, including our new field called `search_title`. However, the actual document will not be updated or changed.

Using a pipeline for better searches

We can take this to another level by creating a more powerful and flexible search. Imagine we had a lot of other recipes in our cookbook; let us say ten more recipes that are all for different types of tacos such as: "chicken street tacos", "Tacos Al Pastor", "Easy Ground Beef Tacos" and "Carnitas Street Tacos".

Note: You can find a `tacos.json` file with the example collection of documents in the chapters/08 folder of the book's repository, if you would like to import them to test this query.

None of these will match a simple search for "tacos" but all would be relevant results. We can do a couple tricks to get the results we want using a pipeline. It should be noted that this is much more of an example than a suggestion of a best practice!

Stage one

First, we will add a lowercase **`title`** field **`search_title`**:

```
{
  "$addFields": {
```

```
    "search_title": {
      "$toLower": "$title"
    }
  }
},
```

Stage two

Then, using that new field we will break up each word in the title into an array of words called **`search_words`** using the **`$split`** operator:

```
{
  "$addFields": {
    "search_words": {
      "$split": ["$search_title", " "]
    }
  }
},
```

We could do that in one pass, without a separate **`search_title`** field, but we plan to use it later.

Stage three

At this point, we can do a simple search match on items in the **`search_words`** array:

```
{
  "$match": {
    "search_words": "tacos"
   }
},
```

Stage four

Then, we can use our **`search_title`** to sort our recipes (this will make sure the lower cased "chicken street tacos" comes back ordered right before "Chorizo Breakfast Tacos"):

```
{
  "$sort": { "search_title": 1 }
},
```

Stage five

Lastly, we will return just the original **`title`** of the recipe.

```
{
  "$project": {
    "_id": 0,
    "title": 1
  }
}
```

Now we should get a result as follows:

```
[
  { title: 'Barbacoa Tacos' },
  { title: 'Carne Asada Tacos' },
  { title: 'Carnitas Street Tacos' },
  { title: 'chicken street tacos' },
  { title: 'Chorizo Breakfast Tacos' },
  { title: 'Easy Ground Beef Tacos' },
  { title: 'Lengua Tacos' },
  { title: 'Tacos Al Pastor' },
  { title: 'Tacos De Pescado' },
  { title: 'Tacos de Pollo Asado' }
]
```

The entire pipeline will look like the following (you can find a copy in **`case-insensitive-words.json`** in the chapter's folder in the GitHub repository):

```
> db.tacos.aggregate([
// stage one
{
  "$addFields": {
    "search_title": {
      "$toLower": "$title"
    }
  }
},
// stage two
```

```
{
  "$addFields": {
    "search_words": {
      "$split": [ "$search_title", " "]
    }
  }
},
// stage three
{
  "$match": {
    "search_words": "tacos"
   }
},
// stage four
{
  "$sort": { "search_title": 1 }
},
// stage five
{
  "$project": {
    "_id": 0,
    "title": 1
  }
}
])
```

If we wanted to change our search string slightly, using two words such as "street tacos", we could change our **`$match`** like so (recalling using **`$all`** to match multiple elements in an array field):

```
{
  "$match": {
    "search_words": {
      "$all" : ["street", "tacos"] }
   }
},
```

Now we will get back just two tacos as the result:

```
[
  { title: 'Carnitas Street Tacos' },
  { title: 'chicken street tacos' }
]
```

Updating documents

You can also use pipelines to update documents, not just modify query results. This allows you to create things such as pre-calculated fields, like the **`rating_avg`** field in the example recipe documents or do other modifications.

To sum up the average of the **`rating`** field for a document and add or set that into a **`rating_avg`** field you can run a query as follows:

```
> db.examples.updateOne(
{ "_id": "recipe:apple-pie" },
[{
  "$set": {
    "rating_avg": {
      "$round": [ { "$avg": "$rating"}, 2]
    }
  }
}])
```

To update that field for all the documents in the collection that match a certain criteria, you can change your match and use **`updateMany()`**:

```
> db.examples.updateMany(
{ "type": "Dessert" },
[{
  "$set": {
    "rating_avg": {
      "$round": [ { "$avg": "$rating"}, 2]
    }
  }
}])
```

If you run the same query with an empty document, **`{ }`** that will add the field *to every document*. That means if there is no **`rating`** field in that document, you will end up with a **`rating_avg`** field with the value of **`null`**. To avoid this problem, you can change your match to use an **`$exists`** query:

```
> db.examples.updateMany(
{ "rating": { "$exists": true } },
[{
  "$set": {
    "rating_avg": {
      "$round": [ { "$avg": "$rating"}, 2]
    }
  }
}])
```

Now, only documents with a **`rating`** array field will get or set a **`rating_avg`** field. You can also use the **`$replaceWith`** stage, which offers a more expressive update statement:

```
> db.examples.updateOne(
{ "_id": "recipe:apple-pie" },
[{
  "$replaceWith": {
    "$setField": {
      "field": "rating_avg",
      "input": "$$ROOT",
      "value": {
        "$round": [ { "$avg": "$rating"}, 2]
      }
    }
  }
}])
```

There are many more stages available, such as **`$lookup`** which allows you to preform SQL like "joins" between documents in different collections (if they are in the same database). There is also **`$merge`** which will write the results directly to a collection.

Note: the `$merge` stage must be the last stage in a pipeline.

Conclusion

In this chapter, we have only touched on the many powerful things you can do with the aggregation framework, but you should now have a solid grasp of the concepts and be able to learn more independently as needs arise. All these stages and operators are heavily documented on the official MongoDB website **https://www.mongodb.com/docs/manual/** so make sure to check that out.

Now, empowered with the knowledge of basic queries as well as MongoDB's Aggregation Framework, you can compose all sorts of complex queries to get just about any result you need.

In the next chapter, we will learn more about Collections and indexing and how both can affect performance and other aspects of your database and application.

Chapter 9

Planning for Performance – Collections and Indexes

"There is nothing like looking, if you want to find something. You can certainly usually find something, if you look, but it is not always quite the something you were after"

— Thorin, J.R.R. Tolkien, The Hobbit

Introduction

One of the more important things you can do to help query performance on your collections is to create and use indexes properly. There are several ways to use indexes in MongoDB, and various types of collections that either use special indexes or have special features for different use cases.

Structure

In this chapter, we will discuss the following topics:

- Indexing collections
    - Basic Indexes
    - Compound Indexes
    - Unique Indexes

    - Query Plans
    - Special Index Types
- Maintaining Indexes
    - Hiding Indexes
    - Deleting Indexes
    - Modifying Indexes
- Collection Settings and Types
    - Capped Collections
    - Time-Series Collections
    - Storing Files with GridFS
- Document Schema Validation
    - Basic Schema Validation
    - Validation in MongoDB Compass
- Collection Maintenance
    - Collection Statistics
    - Deleting Collections

Objectives

In this chapter, we will explain what an index is, how MongoDB uses indexes, different index types, different collection types, and how to create, configure and delete indexes and collections. By the end of this chapter, you should have a solid foundation of indexing and collection options in MongoDB and lots of areas you can look further into, if you want to learn more.

Indexing collections

If you are unfamiliar with the concept of an index, it is not unlike an index you might find in a book. It gives your database a way to "lookup", where the data you want is in your collection. The query planner can "jump" to the right place and pluck out just the data you need, instead of having to "read" through your whole collection, document by document.

In fact, you have been using an index all along, or at least MongoDB has. The **`_id`** for each document is indexed as a regular "single field" index and referred to as the "primary key". MongoDB offers many options for indexing as well as some specialized types of indexes, which we cover in this chapter.

Basic Indexes

The most straightforward kind of index uses a single field, like our **`title`**, or the **`type`** and a "direction" which is essentially ascending (specified by **`1`**) or descending (specified by **`-1`**).

You can see this more visually in the following *Figure 9.1* where we have created an index on **`title`** in *ascending* order (A-Z):

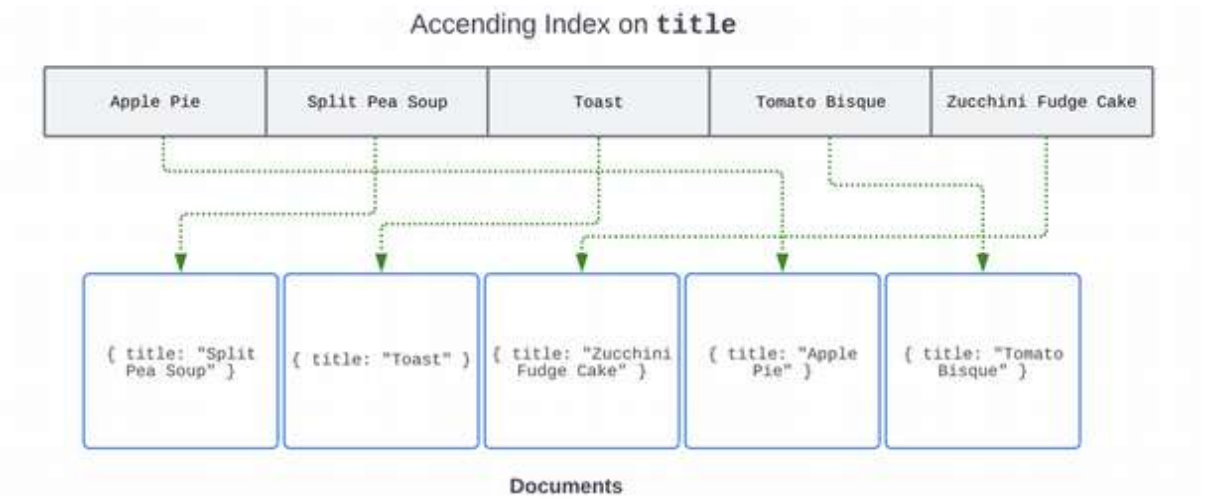


Figure 9.1: *Example of an Index in MongoDB*

In your collection, the documents are in the order they were inserted, but the index takes out just the **`title`** field, and a reference to document (like its **`_id`**) and then orders itself A-Z by the **`title`**. This allows MongoDB to know where to find the documents in the collection, but do so by whatever arrangement the index has, instead of the insertion order.

Creating an index

You can construct an index using the **`createIndex()`** method on your collection like so:

```
> db.collection.createIndex( { "title": -1 } )
```

This will create an index on the **`title`** field in *descending* (**`-1`**) order.

Creating an Index in MongoDB compass

You can also use the **`Indexes`** tab in MongoDB Compass and press the **Create Index** button. This will launch a model as seen in *Figure 9.2*, to create your index along with the chance to set some options, which we will cover later in this chapter.

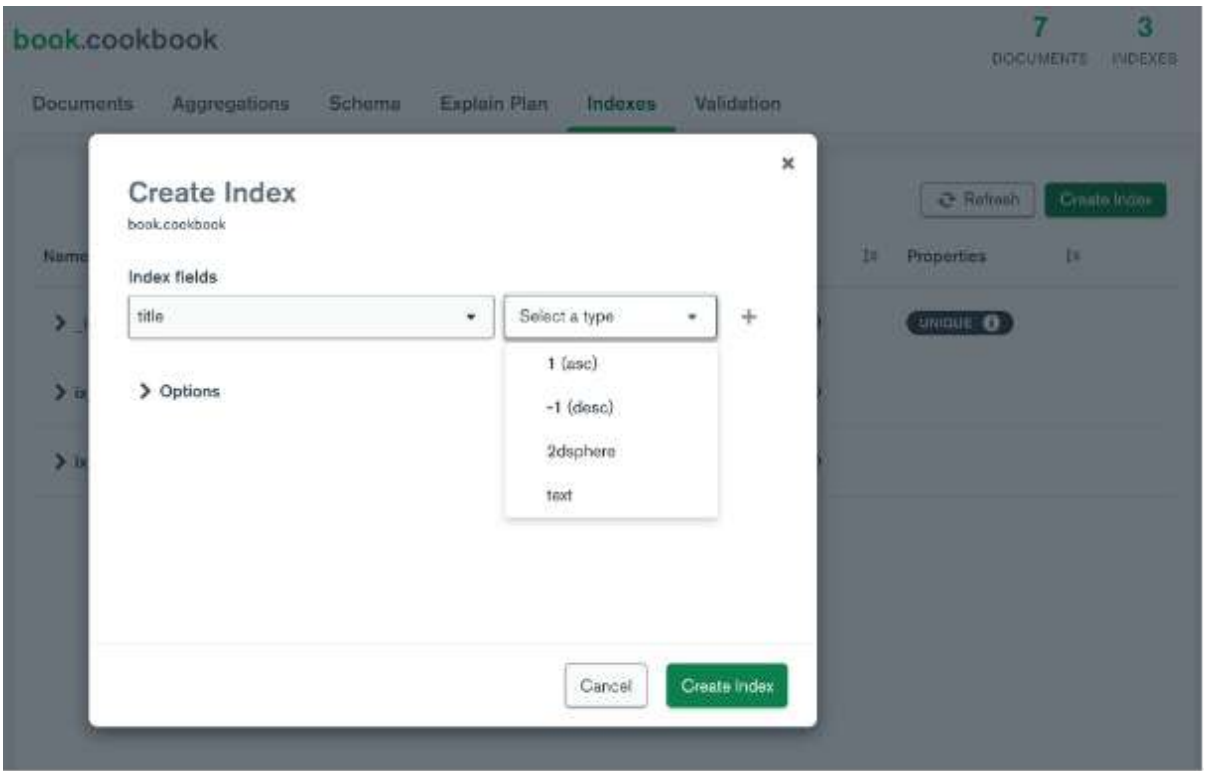


Figure 9.2: *Create Index within MongoDB Compass*

Often, the exact direction will not matter, but if you happen to be ordering your query results on the same field, having it already in the same order/direction will improve performance.

Naming an index

By default, MongoDB will give your index a unique name based off the field name and direction. However, you can provide your own name as well, if you wish, by setting it as part of the second parameter:

```
> db.collection.createIndex( { "title": -1 }, { "name": "ix_my_name" } )
```

This can be any valid string. You can see the indexes in your collection either by using the **`Indexes`** tab in MongoDB Compass, or using the following command:

```
> db.cookbook.getIndexes()
[
```

```
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { title: 1 }, name: 'ix_title', sparse: false },
  {
    v: 2,
    key: { 'ingredients.name': 1 },
    name: 'ix_ingredients_name',
    sparse: false
  }
]
```

This will always return at least the index on **`_id`** since that is required, and you cannot delete that primary index. In this example, it is also returning the two other indexes we created on the **`title`** and the **`name`** field *within* the **`ingredients`** array.

Indexes on Arrays

As you can see from the preceding example, you can also create indexes on fields *inside* indexes. Simply use dot notation to pinpoint the field you wish to index:

```
> db.collection.createIndex( { "ingredients.name": 1 } )
```

This is referred to as a mulitkey index, since it technically has two keys, or fields: **`ingredients`** and **`name`**.

Compound indexes

In contrast to multikey indexes, MongoDB also supports compound indexes. This index type is composed of two or more fields, and each field can have its own direction. The index is then ordered by the keys within it, for example if we create a compound index like so:

```
> db.collection.createIndex( { "title": 1, "type": -1 } )
```

The index will be ordered first by **`title`** in ascending, and by **`type`** in descending order. This ordering is very important as it can mean the difference between a query being able to use the index to both find and order results back for you, or the index not being able to be used at all. Make sure to try and match your indexes to how you plan to query.

Unique indexes

Like most databases, MongoDB has the concept of a "unique" index that enforces a rule that only one document can have a particular value, in a certain field. You can

create a unique index just like a single, or compound field index, but add the extra option of **`unique`** set to **`true`**:

```
> db.posts.createIndex( { "slug": 1 }, { unique: true } )
```

In this example, we have set a unique index on the field **`slug`**, which is often used as part of a blog post's unique URL, thus making sure no two documents in this collection can have the same value in their **`slug`** field.

Note: You cannot create this sort of index if you already have two or more documents in your collection that have the same value in the unique field as this would "violate" that index.

Query plans

To get a better idea of how MongoDB is running your query, you can add the **`explain()`** method to the end of your **`find()`** query. This will return a large amount of data, explaining how the database ended up executing your query. This is known as a "query plan". For our purposes, we only care about the "winning plan" so we can reduce our output by requesting just that object:

```
> db.cookbook.find({ "type": "Dinner" }).explain().queryPlanner.winningPlan
{
  stage: 'COLLSCAN',
  filter: { type: { '$eq': 'Dinner' } },
  direction: 'forward'
}
```

You can also do this in MongoDB Compass by clicking on **Explain Plan** tab in your collection, as seen in *Figure 9.3*:

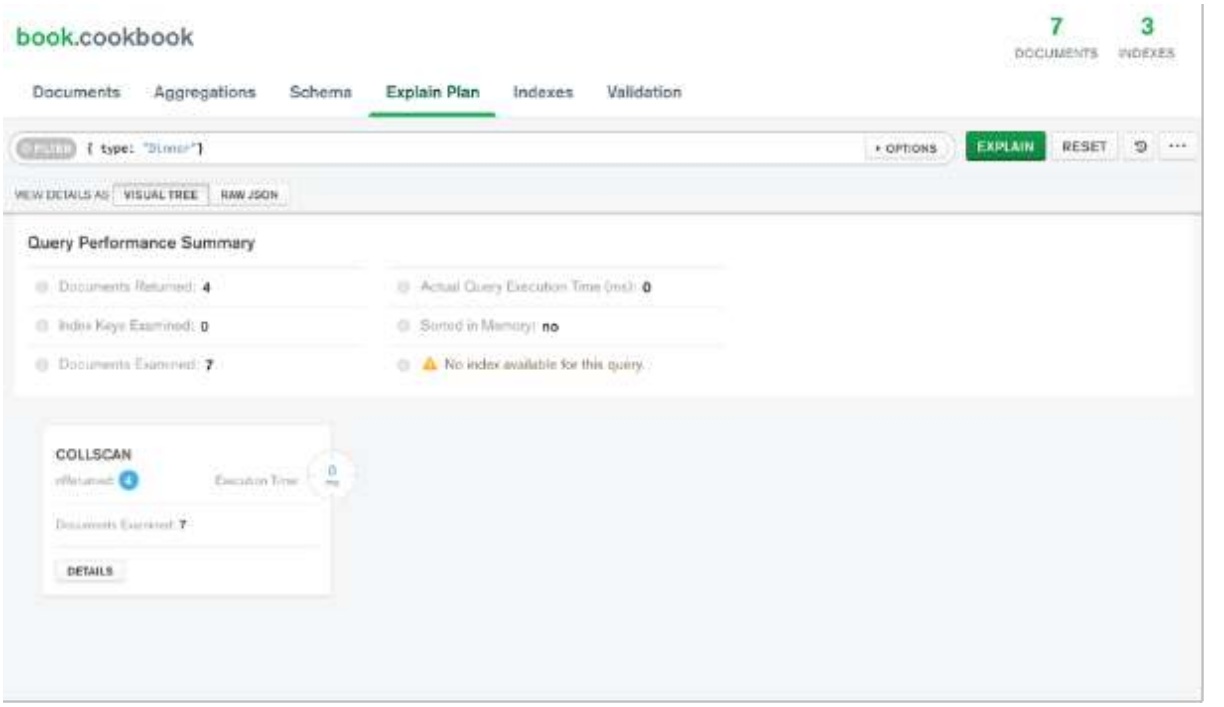


Figure 9.3: *Using Explain Plan in MongoDB Compass*

This lets us know the database needed to perform a **`COLLSCAN`** (or collection scan), which means that it had to "scan" or read each document in the collection to find our match since there was no index available for this query.

If you look closely at *Figure 9.3*, you can see the **Documents Examined** is **7**, which is all the documents in the collection. That is not very efficient.

Using explain() With an Index

If we change our query to search by **`title`**, you see will a different query plan. This is because within the example **`cookbook`** collection, we have created an index on the **`title`** field. The database can use that index instead of "scanning" the whole collection:

```
> db.cookbook.find({ "title": "Split Pea Soup" }).explain().queryPlanner.
winningPlan

{
  stage: 'FETCH',
  inputStage: {
    stage: 'IXSCAN',
```

```
    keyPattern: { title: 1 },
    indexName: 'ix_title',
    isMultiKey: false,
    multiKeyPaths: { title: [] },
    isUnique: false,
    isSparse: false,
    isPartial: false,
    indexVersion: 2,
    direction: 'forward',
    indexBounds: { title: [ '["Split Pea Soup", "Split Pea Soup"]' ] }
  }
}
```

Or, in MongoDB Compass as shown in *Figure 9.4*:

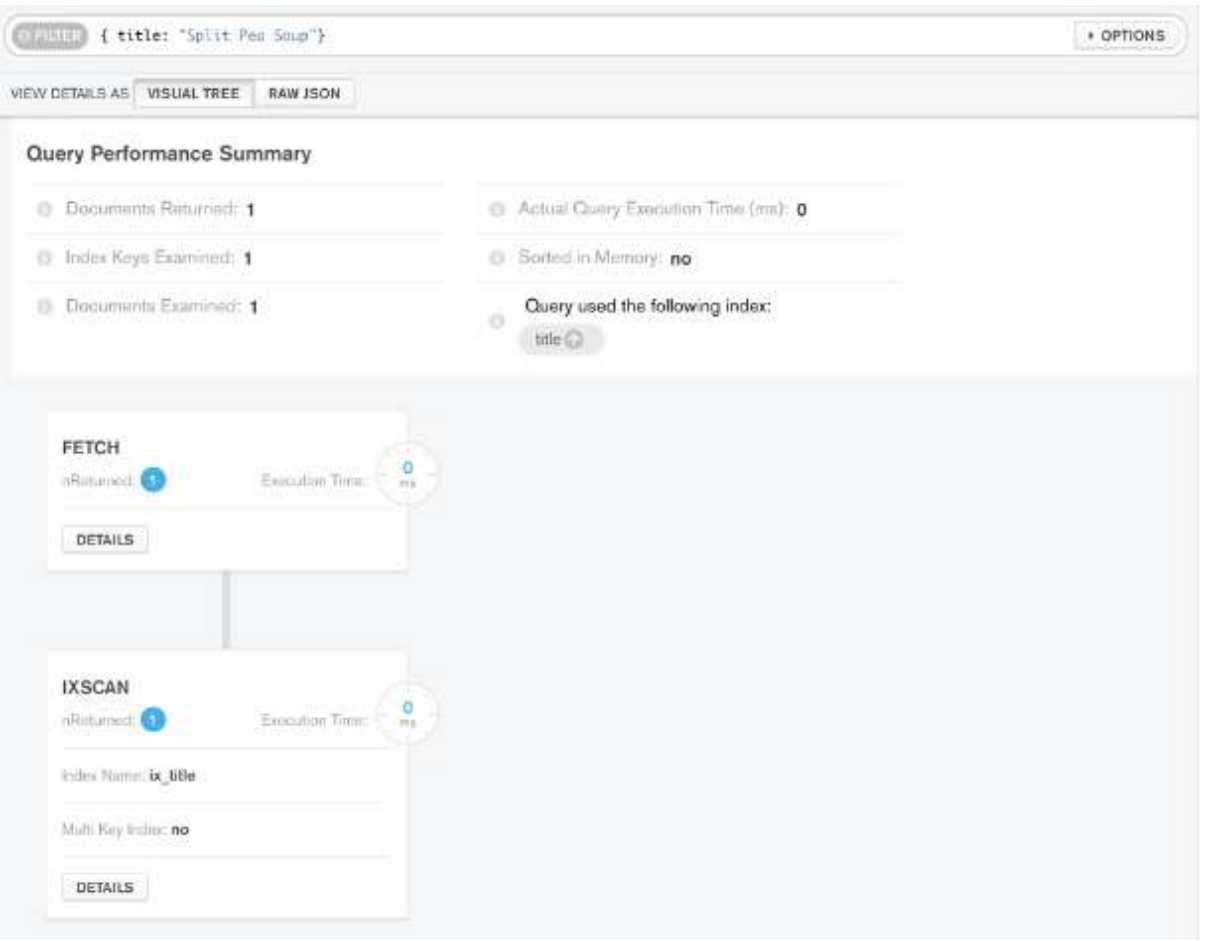


Figure 9.4: *Using Explain with an Index in MongoDB Compass*

Note the **IXSCAN** instead of **COLLSCAN** and the **Documents Examined** has been reduced to **1**, which is the same as our result count. Much better!

Special index types

Aside from the array indexes, the index types we have examined so far are common in most databases. However, there are some special types of indexes that are more unique to MongoDB, such as indexes that can automatically delete documents based on when they were created, or indexes that can adapt to fields you have not even created yet! We will discuss some of these index types in the next section.

Case insensitive indexes

As we have discussed at length, by default, MongoDB is case sensitive, so "Red Pepper" is not equal to "red pepper" when searching. Among the various ways to deal with this, beyond what we have covered thus far, are case insensitive indexes. These require a bit more planning and do have some performance impacts. However, they might be something you will want to use in some cases.

To create a case insensitive index, you will need to specify a **`collation`** with a **`locale`** and a **`strength`**:

```
> db.cookbook.createIndex(
  { "title": 1},
  { "collation": { "locale": "en", "strength": 2 }
})
```

With this index, if you run the following query, on a collection with a recipe titled "Apple Pie", you will still get no results:

```
> db.cookbook.find({ "title": "apple pie" })
```

If, however, you add the **`locale`** you specified, and the corresponding **`strength`**, you will get a match:

```
> db.cookbook.find(
  { "title": "apple pie" }
).collation(
  { "locale": "en", "strength": 2 }
)
```

You can use a **1** or **2** for your **strength**. Optionally, you can specify **locale** and **strength** when you create the collection, if you do not want to add it to the end of your queries:

```
> db.createCollection("myCollection",
  { "collation": { "locale": "us", "strength": 2 } }
)
```

This will change the "default collation" of your collection and will have other effects, so read up on this option in the official documentation for more, to be sure you understand all the consequences.

Wildcard indexes

Since document schemas are so flexible, you may run into situations where you cannot be sure what fields might exist in a document, but you are sure you would want an index on each field. To solve this problem, you can preemptively create a wildcard index, which will automatically index any field in the scope that you setup.

This is particularly useful if you embedded documents inside your document. Perhaps you have a lot of options for your site's user profile. As a user fills out more information, you add more to their profile embedded object, instead of having a lot of "empty" or **null** fields. So, your profiles might look something like:

```
{ "profile": { "type": 1, "subscribed": true, "cat_count": 4 } },
{ "profile": { "type": 2, "tags": [ "cheese", "basil" ] } },
{ "profile": { "type": 1, "location": "Olympic City, USA" } }
```

If we create a wildcard index on the **profile** field, any fields within it will get an index, even if we add more fields later, or not all documents have the same fields. To do this, specify the field you want to wildcard and then add a **$**** like so:

```
> db.users.createIndex( { "profile.$**" : 1 } )
```

Now any of the following queries will be "covered" by an index:

```
> db.users.find({ "profile.type": 1 })

> db.users.find({ "profile.cat_count": { "$gt": 1 } })

> db.users.find({ "profile.tags": "cheese" })

> db.users.find({ "profile.location": { "$exists": true } })
```

You can also create a wildcard index *for all the fields in the document* (**`_id`** is excluded from this by default, but you can add it):

```
> db.collection.createIndex( { "$**" : 1 } )
```

Now all fields will have an index. You will want to be careful to do this only in cases where it is more helpful than hurtful; a good rule of thumb would be to do this if you expected to query directly on all your fields at different points.

You can also include or exclude fields from a wildcard index by using the **`wildcardProjection`** option:

```
> db.collection.createIndex(
  { "$**" : 1 },
  { "wildcardProjection" :
    { "profile.type" : 0 }
  }
)
```

This will create an index on all the fields except the **`profile.type`** field because we set **`0`** for **`false`**.

Time to live indexes

For some collections, you might want to automatically delete documents when their usefulness has expired. This might be time sensitive information that is no longer useful after a certain date, such as log data, or perhaps data that you have told the user you will only store for 48 hours. Instead of having to write some sort of process to "clean up"' this data, you can use a **Time To Live** (**TTL**) index.

To create a TTL index, you will need a field that is of type date, and then set the **`expireAfterSeconds`** option, which is the number of seconds that document should "live" for. In other words, the number of seconds until it will be automatically deleted.

To delete a document after 24 hours based off a date field called **`createdDate`**, you would use the following syntax:

```
> db.collection.createIndex(
  { "createdDate": 1 },
  { "expireAfterSeconds": 86400 }
)
```

There are a few limits on this type of index, so check out the official documentation if you run into issues. There is also the option of using a "capped" collection, which we will discuss later in this chapter.

Geospatial indexing

While it is beyond the scope of this book, there is also some other special types of indexes that can be used for geospatial data and geospatial queries, such as geometries on an earth-like sphere or to calculate distances on a Euclidean plane. If that piques your interest, make sure to consult the official documentation.

Maintaining indexes

It is important to keep an eye on how your indexes are performing, and delete or adjust them as needed. If you are interested how much usage an index is getting you can use **`$indexStats`** like so:

```
> db.cookbook.aggregate([ { $indexStats: { } } ])
```

This will return an array with each index and its stats; you can get some of this same information from the **`Indexes`** tab in MongoDB Compass, as seen in *Figure 9.5:*

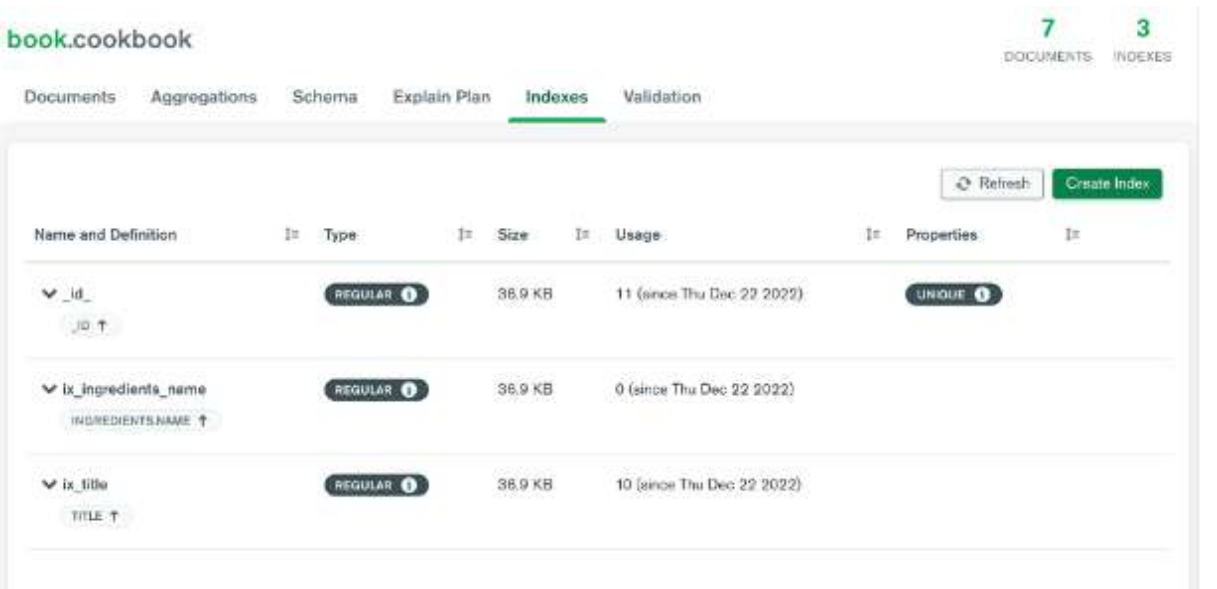


Figure 9.5: *Indexes Tab in MongoDB Compass*

If you would like to get the stats of a particular index, you can add a **`$match`** stage and the index name:

```
> db.cookbook.aggregate([
  { $indexStats: { } },
```

```
  { $match: { "name": "ix_title" } }
])


[
  {
    name: 'ix_title',
    key: { title: 1 },
    host: '7d1c8990e16e:27017',
  accesses: { ops: Long("200"), since: ISODate("2022-12-22T07:20:50.665Z")
},
    spec: { v: 2, key: { title: 1 }, name: 'ix_title', sparse: false }
  }
]
```

Here, we can see that this index has been accessed 200 times (the value of **`accesses.ops`**), since MongoDB started tracking statistics on it, which is the **`accesses.since`** date. This will get reset if the database is restarted, and it is not the lifetime usage. However, if there has been proper time to gather statistics and the usage is low, or if it the index has never been used, you might consider dropping (deleting) it.

Hiding Indexes

Before you delete an index, unless you are very sure, you might want to do a quick test. MongoDB gives you the option to "hide" an index, which means it will still be there, but the query planner will not use it for queries. This can be helpful as you can then run some test queries to see if there are any performance impacts before deleting the index. You can hide an index like so:

```
> db.collection.hideIndex("index_name")
```

Now, if you run **`getIndexes()`**, you will still see the index, but it will have a **`hidden: true`** attribute set, meaning the query planner cannot see it. To make it visible to the query planner again, use the following command:

```
> db.collection.unhideIndex("index_name")
```

Since the index was being maintained while it was hidden, it can be used by the query planner immediately after it has been made visible again.

Note: If a TTL index is hidden, the documents will still be expired (deleted) when they reach their TTL.

Deleting indexes

If you are sure that you should delete an index, you can do so with the following command:

```
> db.collection.dropIndex("index_name")
```

You can also delete all the indexes in a collection at once:

```
> db.accounts.dropIndexes()
```

Modifying indexes

You cannot directly modify an index; you can however drop an existing index and recreate it with your modifications. To avoid performance issues while you figure out what modifications you need to make to an index, you could create a temporary index on the field your current index uses, and then delete that temporary index when you are finished.

Collection settings and types

Thus far, we have been using collections with their default settings, but there are some helpful optional settings that will change your collection and allow you to use it for more specific purposes. As discussed earlier, a collection is "automatically" created the first time it has anything inserted into it. So, if you type the following command, you will create a collection called **`myNewStuff`**:

```
> db.myNewStuff.insertOne({})
```

This will create the collection and insert one empty document into it even though **`myNewStuff`** did not yet exist. If you list the collections in your database, you will see the **`myNewStuff`** is now present:

```
> show collections
cookbook
examples
myNewStuff
test
```

This created the collection with the default settings. To set the *optional* settings, you can use the **`createCollection()`** method on the database like so:

```
> db.createCollection(<name>, { <option>: <value> })
```

For this method, the **`<name>`** is any valid string and the **`{ options }`** document contains the options that you wish to set for the collection, or you can leave off the **`options`** document to use the default settings.

From within MongoDB Compass, you can press the **Create Collection** button, or select your database on the left side navigation, and press the + button, as seen in *Figure 9.6*:

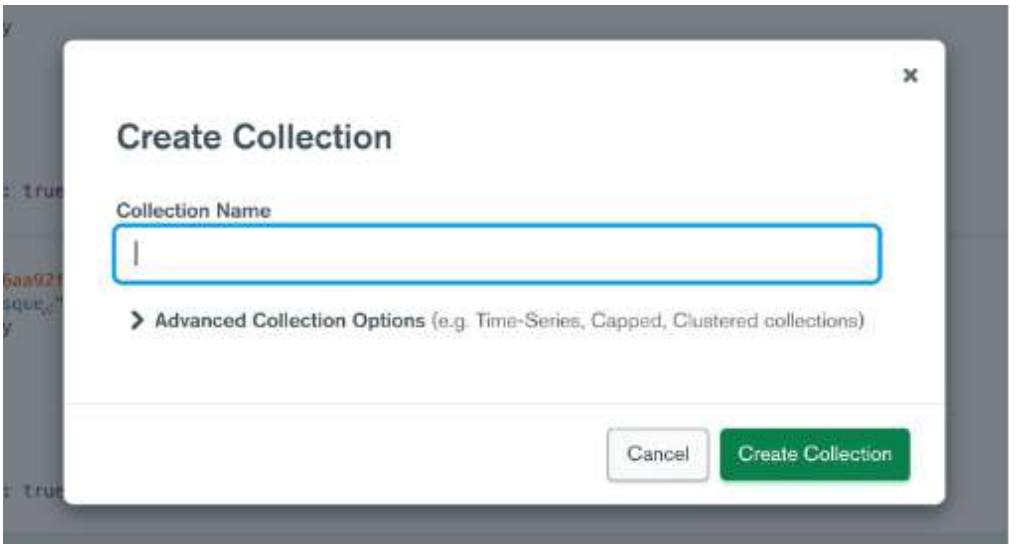


Figure 9.6: *Create Collection in MongoDB Compass*

As you can see in *Figure 9.6*, there are some advanced options and collection types available. We will discuss some of these very useful options in the next couple sections.

Capped Collections

As an alternative to TTL indexes, you can also "cap" your collection, so that it will only ever have a certain number of documents at a time, or to be constrained by a set size on disk. This can be very useful for data where the insertion order matters (as in the time the document was inserted) but where the documents themselves may be less useful, or needed over time, such as log data.

With a capped collection, you could make sure that you keep just the last 100,000 log entries or keep as many log entries as you can store in 1 gigabyte on disk. The size (as in the maximum size on disk) is always required when creating a capped collection, the maximum number of documents is optional. If either limit is reached, MongoDB will start to automatically delete documents from your collection, always starting with the oldest documents.

To create a capped collection, set the following options when creating the collection:

```
> db.createCollection("myCapped", {
  "capped": true, "size": 1073741824, "max": 100000
})
```

You can check if the collection you are working with is capped by using the following method:

```
> db.myCapped.isCapped()
true
```

However, there are some differences and constraints with capped collections. For example, if you plan on updating your documents, make sure to create an index on the field you will use to update, so that it will not require a collection scan. Moreover, you might consider using a TTL index to accomplish auto-deleting documents, as it does not have some of the constraints of a capped collection.

Time-Series collections

Using some of the same concepts, such as insertion order, time-series collections focus on one or more "unchanging parameters". This is most useful when collecting measurements over time. A common use case is sensor data where the unchanging parameter would be the sensor's id, and the changing value would be whatever that sensor is tracking. Alternatively, the unchanging factor could be a unique identifier of some sort, and at a defined interval, there could be multiple measurements, or it could be tracked.

For example, a smart outdoor grill with sensors might be able to take a measurement every 30 seconds of the outside air temperature, the grill's temperature and the temperature of the meat that is cooking.

Very roughly, to create this time-series collection, the statement might look like:

```
> db.createCollection("smartGrill",
{ "timeseries": {
    "timeField": "timestamp",
    "metaField": "metadata",
    "granularity": "seconds"
  }
})
```

The **`timeField`** is linked to a **`timestamp`** field, and the **`metaField`** is linked to the unique, unchanging values in **`metadata`**.

The documents with the gathered data might look like:

```
[
{
  "metadata": { "sensorId": 123, "type": "grill" },
  "timestamp": ISODate("2023-01-10T00:01:00.000Z"),
  "currentMeasures": {
    "outsideTempF": 70.2,
    "grillTempF": 217.0,
    "meatTempT": 101.3
  }
},
{
  "metadata": { "sensorId": 123, "type": "grill" },
  "timestamp": ISODate("2023-01-10T00:01:00.5000Z"),
  "currentMeasures": {
    "outsideTempF": 70.1,
    "grillTempF": 217.2,
    "meatTempT": 101.4
  }
}
]
```

MongoDB can then use this structured stream of data to analyze the changes (here, in the **`currentMeasures`** object) over time both efficiently, and very powerfully.

Storing files with GridFS

As covered earlier, a document has an upper limit in size for performance reasons, but that does not mean you cannot store large pieces of data, or whole files within MongoDB. To get around the document limit, MongoDB has a technology called GridFS, which will break up a file into smaller pieces (which can be stored in individual documents) and then stream back the pieces to you in the form of the original file. You can store images, various types of files, even videos.

We will cover more about GridFS later in this book.

Document schema validation

Another way to optimize a collection for a specific purpose is to use schema validation, which will enforce a schema on all documents inserted into the collection. As we have covered in depth, by default, MongoDB has a flexible model for document schemas, meaning you can have massive mix of types in a document, and each document in the collection does not need to follow the same schema.

However, there are use cases where you may want this flexibility, but also "lock" the schema design for all documents in the collection, so that there are no unintended changes to that schema. While it is common to not be concerned with document validation, for some use cases, it might be critical.

This validation could be on the field level; perhaps enforcing a field so it can only be a number and not a "number as a string". Alternatively, it could enforce that a field is always a positive number, or that a password field is a string of a certain minimum length. Another use case might be enforcing that a field can only be one of four possible values, and the combinations are almost as flexible as the document model itself.

Basic schema validation

As a brief example of how this validation works, assume we know our cookbook recipes will always have a **`title`** and a **`type`** field. To start off we could have a validation rule like so:

```
{
    "$jsonSchema": {
        "required": ["title", "type"],
    }
}
```

This will make sure that these fields are in each document but will not enforce anything about their contents. If we wanted to go a step further, we could make sure the **`title`** is always a string, and that **`type`** is only one of a valid array of "type" strings, by setting the **`properties`** of the schema:

```
{
    "$jsonSchema": {
        "required": ["title", "type"],
        "properties": {
            "title": {
                "bsonType": "string",
                "description": "Must be a string, and is required"
            },
            "type": {
                "enum": ["Breakfast", "Dinner", "Dessert"],
                "description": "Must be a valid type, and is required"
            }
        }
    }
}
```

Note here that we could set the BSON type, as we did for **`title,`** or we could use an **`enum`** as did for the **`type`** field, to make sure it can only be a value of Breakfast, Dinner or Dessert. For objects, you could supply a **`bsonType`** for each field in that object, and so and so on. Lastly, we were able to supply a descriptive error within the **`description`**, which the user will see if they try to insert or modify a document in an invalid way.

You can assign this validation to a collection when you create (or modify) it by using the **`validator`** option:

```
db.createCollection("myCollection", {
   "validator": {
      "$jsonSchema": { }
  }
})
```

The **`$jsonSchema`** takes a document, like the ones in the preceding examples.

Validation in MongoDB compass

MongoDB Compass provides a helpful UI for validation that will show you valid and invalid documents, as seen in *Figure 9.7*:

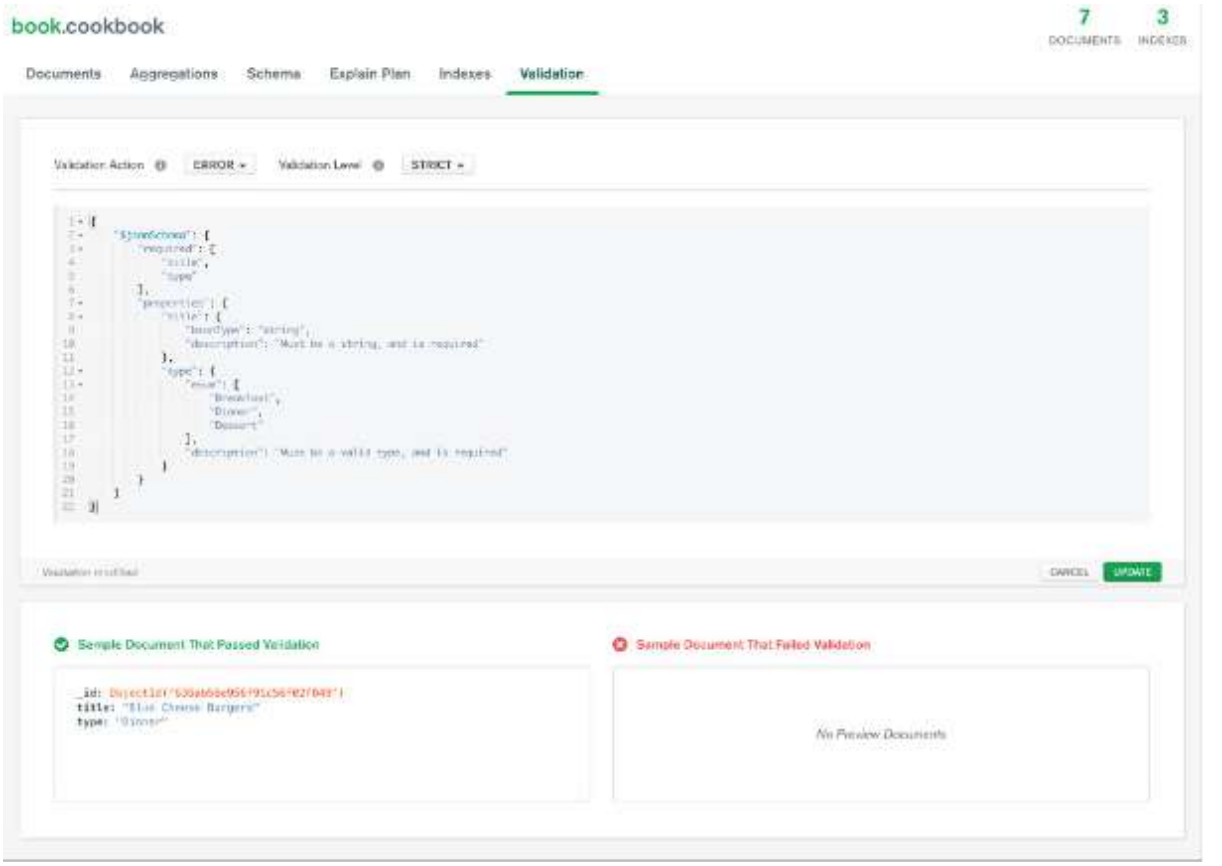


Figure 9.7: *Validation in MongoDB Compass*

If you change your validation to something that would make some documents invalid, MongoDB Compass will let you know. For example, in the following *Figure 9.8*, we changed the **`enum`** for the **`type`** field to not include "Breakfast" which would make a document invalid:

Figure 9.8: *Invalid Validation in MongoDB Compass*

Going further, this can be used to make sure an entire document fits an exact schema, and a lot more. Search the official MongoDB documentation on this topic for a lot more on this feature.

Collection maintenance

Beyond things like backups, imports, and exports, which we will cover in later chapters, there are a couple methods you should be aware of when administrating and maintaining collections. To get a list of all the collections in your database, you can run the following method:

```
> db.getCollectionNames()
[
  'cookbook',
  'examples',
  'test',
  'tacos'
]
```

Collection statistics

If you need more than just the name, you can use the **`getCollectionInfos()`** method which will return an object for each collection, with a summary of information:

```
{
    name: 'cookbook',
    type: 'collection',
    options: {
      validator: {
        '$jsonSchema': {
          required: [ 'title', 'type' ],
          ...
        }
      },
      validationLevel: 'strict',
      validationAction: 'error'
    },
```

```
    info: {
      readOnly: false,
      uuid: UUID("12e46d29-fe25-498c-887a-743ce431534c")
    },
    idIndex: { v: 2, key: { _id: 1 }, name: '_id_' }
  }
```

This will show us useful information just like the collection **`options`**. To get all the statistics for your collection, you can run the **`stats()`** method like so:

```
> db.cookbook.stats()
{
  ns: 'book.cookbook',
  size: 8571,
  count: 7,
  avgObjSize: 1224,
  numOrphanDocs: 0,
  storageSize: 36864,
  freeStorageSize: 16384,
  capped: false,
  wiredTiger: {
    ...
  },
  nindexes: 3,
  indexBuilds: [],
  totalIndexSize: 110592,
  totalSize: 147456,
  indexSizes: { _id_: 36864, ix_title: 36864, ix_ingredients_name: 36864
},
  scaleFactor: 1,
  ok: 1
}
```

Here, we have truncated the results slightly to highlight some useful stats, including being able to see the **`size`** of the collection, the **`count`** of documents, the number of

indexes (**`nindexes`**), their sizes, and so on. You can use this information to make decisions such as how you should backup your collection, or if you should split it up, or move it to another disk, and so on.

Deleting collections

If you need to delete (or drop) a collection, simply run the following command:

```
> db.myCollectionName.drop()
```

You might want to export or backup your collection before you do that, just in case! More on that in the next chapter.

Conclusion

Striking the correct balance between collection settings and indexes can take some trial and error. Each index has a performance impact of its own; for example, any time you insert or update documents every index, covering them also needs to be updated, which has a performance cost. On the other hand, having an index can also make an update run *considerably faster* since the document(s) can be found faster.

If the topic interests you, a good concept to read more on is the **Equality, Sort, Range** (**ESR**) Rule, which can help you construct your indexes. Either way, use the tools we learned about to constantly keep an eye on, and tweak your indexes for optimal performance!

In the next chapter, we will discuss different ways to import and export data from your collection and databases with MongoDB.

Chapter 10

Getting In and Getting Out – Data Migration

"Do not be so open-minded that your brains fall out."

— G.K. Chesterton

"Mr. Data, are you alright?"

"Yes, sir. I am attempting to fill a silent moment with non-relevant conversation."

"Ah. Small talk."

— Captain Picard and Lt. Commander Data

Introduction

Oftentimes running a query, either via the MongoDB Shell or via programming language, is enough to get data in or out of your MongoDB Server. However, there are times where you will need to import and export data via different methods, be it for backups, or bulk inserts, or because you have data you need to move around. We have covered very basic importing thus far, and now we will delve into more powerful and complex methods for importing and exporting.

Structure

In this chapter, we will discuss the following topics:

- Importing Data

    - Importing via MongoDB Compass
    - MongoDB Database Tools
    - Bulk Inserts
- Exporting Data
    - Exporting via MongoDB Compass
    - Using Database Export Tools
- Transferring Data
    - Transferring via the Aggregation Framework
    - Archiving Documents

Objectives

By the end of this chapter, you should feel comfortable with importing and exporting in MongoDB using MongoDB Compass, MongoDB Database Tools as well as scripting methods. Additionally, you will have a good idea of how to transfer data between collections and databases.

Importing data

We previously covered the basics of importing data into MongoDB, but there are many other options and ways to get data into your database.

Importing via MongoDB Compass

As a reminder, you can use the import functionality of MongoDB Compass by pressing the **ADD DATA** button, as seen in *Figure 10.1*:

Figure 10.1: *MongoDB Compass Add Data Button*

Then choose either to import via a file, as seen in *Figure 10.2*:

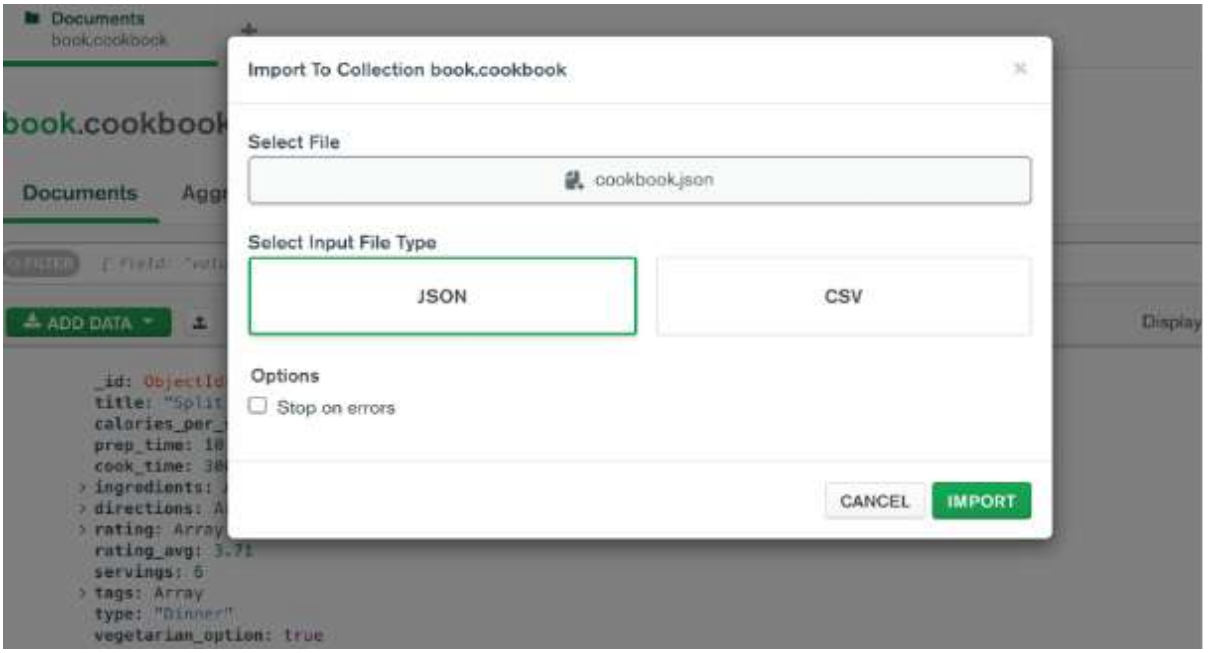


Figure 10.2: *MongoDB Compass Import Modal*

Alternatively, you can also insert documents inline, by using Insert Document, as seen in *Figure 10.3*:

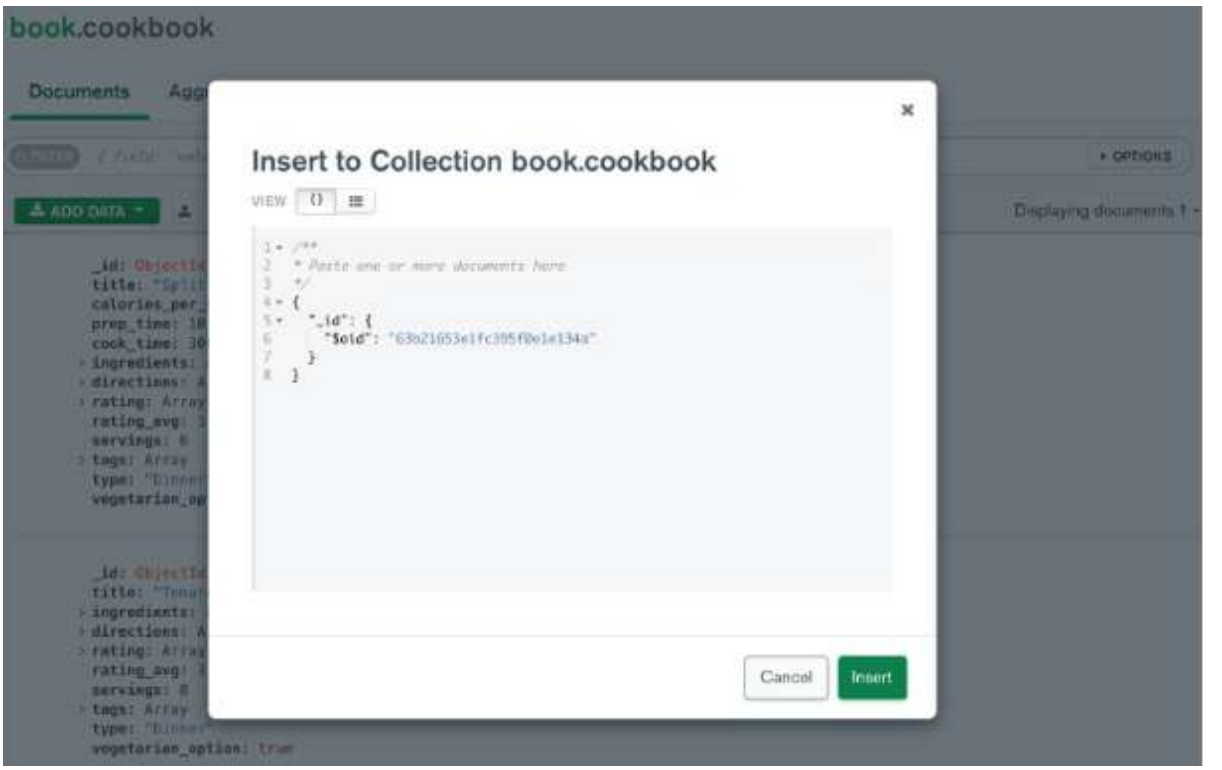


Figure 10.3: *MongoDB Compass Insert Modal*

You can insert more than one document at a time using this method as well, simply wrap your documents in an array like so:

```
[
  { "_id": 1, "field": "value" },
  { "_id": 2, "field": "value" }
]
```

For one off, or smaller inserts, MongoDB Compass' tools might be enough. For more complicated, or bigger imports, you will want to use the command line *MongoDB Database Tools*. You may have already installed these (optionally) at the beginning of the book. However, if you have not, now would be a good time to do so. We will cover these tools in the next section.

Using database import tools

MongoDB has some powerful tools for interacting and monitoring your database, as well as importing and exporting data. These tools have been bundled together as the *MongoDB Database Tools* and are available for free from the MongoDB website. These tools run on all major operating systems.

MongoDB database tools

You can read more about the tools here:

https://www.mongodb.com/docs/database-tools/

If you have not yet done so, download and install the *MongoDB Database Tools* from the MongoDB website:

https://www.mongodb.com/try/download/database-tools

To run any of the tool commands, use a command line and type out their name. Depending on your operating system or install method, you may need to run the commands slightly differently.

On Mac or Linux, you can run the commands like so:

```
$ mongoimport
```

For Windows, you may need to browse to the folder you have installed the tools to, but then you can also run the command as follows:

```
C:\Program Files\...bin\> mongoimport.exe
```

For simplicity, we will refer to commands with the standard **$** prompt, as we have been using **>** to indicate the **mongosh** command is in use. So, if you are using Windows or have some other setup, keep that in mind.

You can also use the *Windows Subsystem for Linux*, which will make things easier.

Using the mongoimport Command

If you are running MongoDB locally, on the default port, you can simply run the **mongoimport** command directly:

```
$ mongoimport
```

Or you can specify a connection string. If you are connecting to a remote server, or a MongoDB Atlas cluster, you will need to adjust your hostname and/or connection string (see MongoDB Atlas for your query string, if using that service). To connect to your local MongoDB server, you can use a connection string like this:

```
$ mongoimport --uri "mongodb://localhost:20701"
```

However, either way, you will get an error:

```
...    no collection specified
...    using filename '' as collection
...    error validating settings: invalid collection name: collection
name cannot be an empty string
```

To fix this, we will, at the least, need to include a file to import. We will be testing importing using a file called **test1.json** which looks like this:

```
{
  "title": "Document, Party of 1"
}
```

You can find a copy of this file in the **chapters/10** folder in the book's git repository. Using this file and the **test** database, we can perform our first import.

Make sure you have either moved the **test1.json** file to the same director you are in, or browse to that **chapters/10** directory and run:

```
$ mongoimport test1.json
```

This should successfully import one document into a collection named **test1** in the **test** database:

```
...    no collection specified
...    using filename 'test1' as collection
...    connected to: mongodb://localhost
...    1 document(s) imported successfully. 0 document(s) failed to import.
```

Since we are importing to a local MongoDB install, the connection string is not necessary.

By default, the **`test`** database will be assumed unless you set another one. Moreover, as the output shows, MongoDB was able to use the file's name as the collection name. To set either of these manually, adjust the connection string and collection option:

```
$ mongoimport --uri "mongodb://localhost:27017/database1"
--collection="myCollection" test1.json
```

This will import into a database named **`database1`** and a collection called **`myCollection`**. You can also specify a username and password, either in the connection string, or by using the **`--username`** and **`--password`** options.

Next, we can import multiple documents, using the **`test2.json`** file:

```
$ mongoimport test2.json


...    Failed: cannot decode array into a primitive.D
...    0 document(s) imported successfully. 0 document(s) failed to import.
```

But here, something went wrong, and you might be wondering what it was. This error happened because of the format of the JSON file, specifically that the documents are in an array which looks like this:

```
[{
  "_id": { "$oid": "63b22823e1fc395f0e1e134d" },
  "title": "Document 1"
},{
  "_id": { "$oid": "63b22838e1fc395f0e1e134e" },
  "title": "Document 2"
},{
  "_id": { "$oid": "63b22840e1fc395f0e1e134f" },
  "title": "Document 3"
}]
```

This was done to make the file valid JSON, but it is not required for **`mongoimport`**. To fix this error, we need to add the **`--jsonArray`** option:

```
$ mongoimport --jsonArray test2.json


...    3 document(s) imported successfully. 0 document(s) failed to import.
```

Now three documents should have been imported into the **`test2`** collection in your **`test`** database.

Importing Into Non-empty Collections

If you run the same import command with **`test2.json`** again, you will get an error:

```
$ mongoimport --jsonArray test2.json


...    continuing through error: E11000 duplicate key
error collection: test.test2 index: _id_ dup key: { _id:
ObjectId('63b22838e1fc395f0e1e134e') }

...    0 document(s) imported successfully. 3 document(s) failed to import.
```

This is because unlike **`test1.json`** which has no **`_id`**, the documents in **`test2.json`** each have an **`_id`**. Since the **`_id`** must be unique if you are importing into an existing collection, you need to make sure that **`_id`** does not already exist. If it does, **`mongoimport`** will skip importing. If you want to perform an "upsert" that will insert or update documents, you can provide the **`--mode`** option and use **`test3.json`** like so:

```
$ mongoimport --jsonArray --collection=test2 --mode=upsert test3.json
```

This will upsert four documents, updating three (each of these have a new **`title`**) and inserting one new document into the **`test2`** collection. If you run the same command again, *zero* documents will be affected as **`mongoimport`** will see there were no changes.

For more on the **`mongoimport`** tool and its many other command options, see the official documentation: **https://www.mongodb.com/docs/database-tools/mongoimport/**

Importing JSON from an API

You can also use JSON directly. For example, we can use JSON pretty much directly from an API.

To illustrate this, we can use a free API from NASA which provides data on **Near Earth Objects** (**NEO**), which are things like asteroids. The following URL will return a JSON object with several NEOs on a particular date:

https://api.nasa.gov/neo/rest/v1/feed?start_date=2023-09-07&end_date=2023-09-07&api_key=DEMO_KEY

You can download the results, or use the command line and a command such as **`curl`**, to save them to a file like this:

```
$ curl 'https://api.nasa.gov/neo/rest/v1/feed?start_date=2023-09-07&end_
date=2023-09-07&api_key=DEMO_KEY' > neo.json
```

We have included a **`neo.json`** file in the **`chapters/10`** folder, so you do not need to perform this step to continue. However, you can still do it if you wish! Using this JSON file, you can run the following set of commands to directly import the API results (the NEOs) into MongoDB, as documents:

```
$ cat neo.json \
 | jq '.near_earth_objects["2023-09-07"]' \
 | mongoimport --jsonArray --collection="neo"
```

To make things a little clearer, let us break each command down into a step. First, we output the contents of the JSON file with **`cat`**:

```
$ cat neo.json
```

The API response, **`neo.json`**, has an object named **`near_earth_objects`** that has an array inside it called **`2023-09-07`** which has all the *Near Earth Objects* on that date. These are stored as normal JSON objects, which looks something like this:

```
{
  "near_earth_objects": {
    "2023-09-07": [
      { "name": "416151 (2002 RQ25)", ... },
      { "name": "523685 (2014 DN112)", ... },
    ]
  }
}
```

We can use another command called **`jq`**, to target this array, like so:

```
$ jq '.near_earth_objects["2023-09-07"]'
```

This JSON array can now be "piped" directly to **`mongoimport`** which accepts Standard Input:

```
$ mongoimport --jsonArray --collection="neo"
```

Bringing all the commands back together (which you can see in the **`chapters/10/import_neos.sh`** file), our process looks like this:

```
$ cat neo.json | jq '.near_earth_objects["2023-09-07"]' | mongoimport
--jsonArray --collection="neo"
```

Running this command will import the eleven *Near Earth Object* "objects", directly into MongoDB documents and we did not have to do anything else!

```
...    connected to: mongodb://localhost

...    11 document(s) imported successfully. 0 document(s) failed to
import.
```

After importing, you see them in MongoDB Compass, complete with all their complex structure. MongoDB also added a **`_id`** for us, as seen in the following *Figure 10.4*:

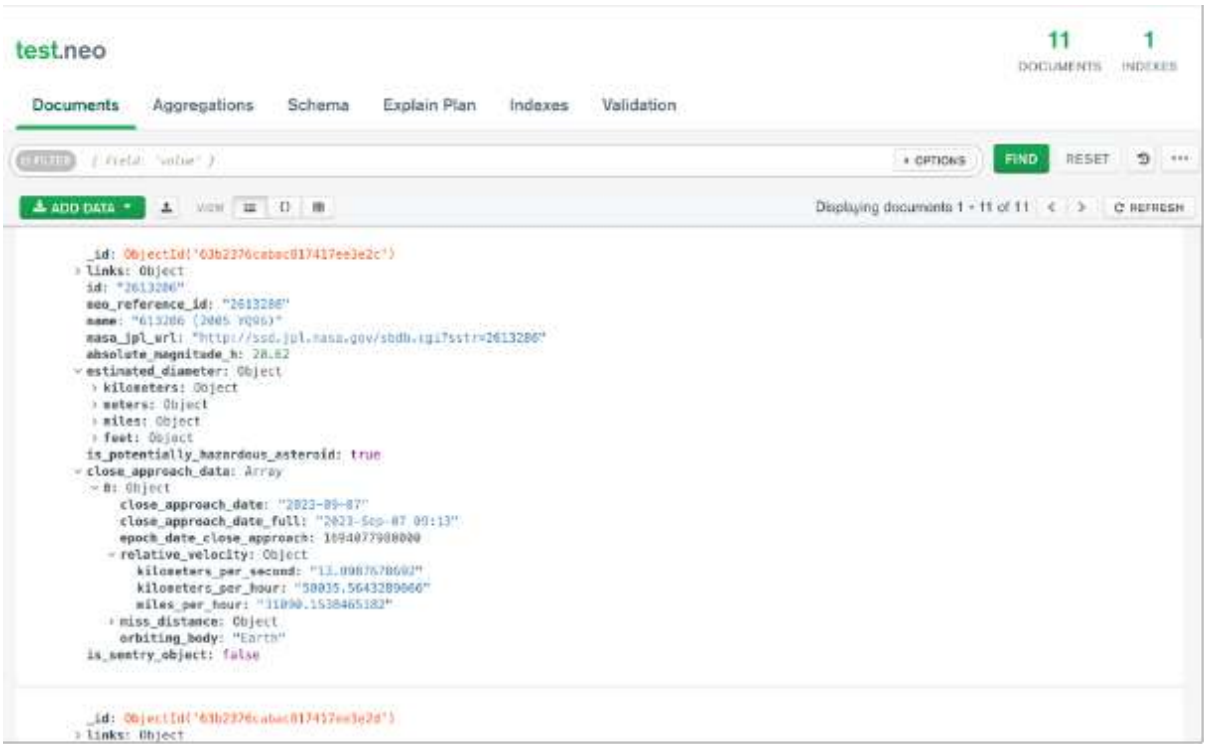


Figure 10.4: *Viewing NEO Documents in MongoDB Compass*

Bulk Inserts

Many databases provide a way to do "bulk", or mass inserts and MongoDB is no different. There are several methods available for this, and you can perform these bulk inserts via **`mongosh`**, a script or via your favorite programming language.

As a simple example, you can run a series of bulk inserts using a set of commands like this:

```
var bulk = db.test.initializeUnorderedBulkOp();
bulk.insert( { "title": "Doc 1" } );
bulk.insert( { "title": "Doc 2" } );
bulk.insert( { "title": "Doc 3" } );
bulk.execute();
```

In essence, you define a bulk operator (which can be ordered, or unordered) and then run the **`insert()`** method on it for however many times you need to, ending with an execute which will insert the actual documents.

Exporting data

To compliment the importing tools, there are also several ways to export data out of your MongoDB database, such as more MongoDB Database Tools and MongoDB Compass.

Exporting via MongoDB Compass

You can export documents directly from MongoDB compass by pressing the export button as seen in *Figure 10.5*:

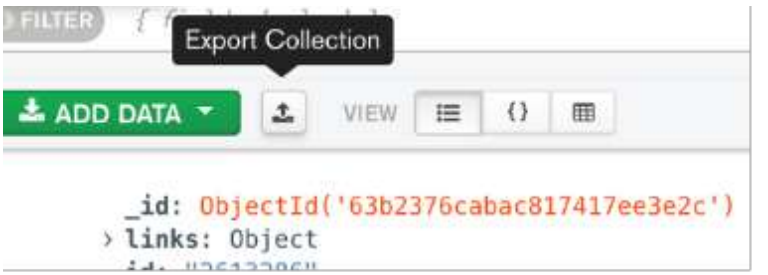


Figure 10.5: *MongoDB Compass Export Button*

This will launch a modal which allows you to write a query for what documents you want to export. For example, maybe you only want to export documents that are older than a year or have a certain value in the **`type`** field. By default, the query will export all documents, as seen in *Figure 10.6*:

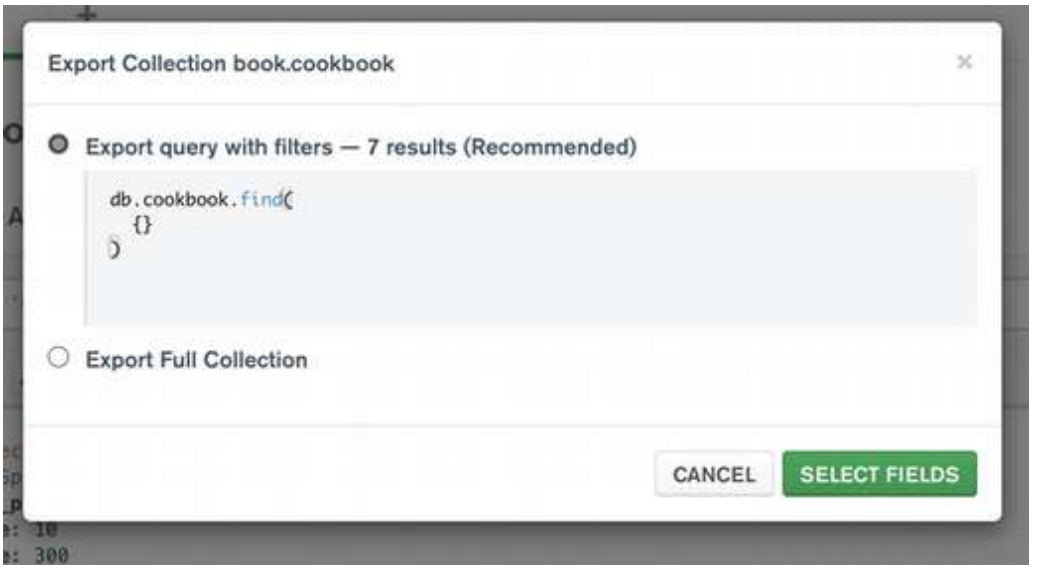


Figure 10.6: *MongoDB Compass Export Modal*

Using database export tools

If you prefer to use the command line, or you have more complex export needs there are a handful of tools for that in the *MongoDB Database Tools*.

Using the mongoexport Command

As a counterpart to the **`mongoimport`** command, **`mongoexport`** will export documents from your collection, as JSON. It has similar options as well:

```
$ mongoexport --db="book" --collection="cookbook"

...    connected to: mongodb://localhost/
 { ... }
 { ... }
 { ... }
...    exported 7 records
```

This outputs all the documents in the collection to Standard Out; you can send this output to a file or pipe it to another program:

```
$ mongoexport --db="book" --collection="cookbook" > backupfile.json
```

You can also use the **`--out`** option to export to a file:

```
$ mongoexport --db="book" --collection="cookbook" --out="backupfile.json"
```

Keep in mind that this will output each document on a new line, with no array wrapping them or commas. This format is the default for **`mongoimport`** but is not technically valid JSON. Just like the example for import, you can add the **`--jsonArray`** option:

```
$ mongoexport --db="book" --collection="cookbook" --jsonArray
--out="backupfile.json"
```

You can also provide a query, to just get certain documents back:

```
$ mongoexport --jsonArray --query='{ "type": "Dinner" }'
```

This can be pretty much any MongoDB query you need, but do note the single quotes around the query, to avoid possible quote related issues.

There are many more options such as **`--limit`**, **`--sort`**, as well as the **`--fields`** option which will just export those fields (plus the **`_id`**) instead of the whole document:

```
$ mongoexport --jsonArray --fields="title,ingredients"
```

Using the mongodump Command

If you are exporting for backup reasons, while you can use **`mongoexport`**, you probably will want to use **`mongodump`** which outputs a "binary" export, in BSON instead of JSON. This not only makes importing quicker, but it can also make sure that some niche issues with JSON to BSON conversion, are avoided.

To dump all the databases and collections on your server you can run the following command:

```
$ mongodump
```

This will dump all the databases (each inside their own folder) and all collections (each in their own **`.bson`** files) within a folder named **`dump`**, in the location you ran the command. To just backup certain databases or collections you can run:

```
$ mongodump --db="book" --collection="cookbook"
```

You can also provide a **`--uri`** connection string if not connecting to a local server, as well as many of the same options as the other commands, such as, to change the path (folder) where the dump goes to use the **`--out`** option. There is also a **`--gzip`** option to automatically compress the dump for you.

Using the mongorestore command

We will be considering backups in more detail in C*hapter 13, Being Proactive – Security and Backups*. However, as a quick preview, you can use the **`mongorestore`** command to import/restore the dumps from **`mongodump`**.

Transferring data

More than likely, eventually, you will need to export and import data from MongoDB *within* MongoDB. For example, what if you need to move a collection to another database? Or move a subset of documents from one collection to another?

There used to be a couple different commands for this in MongoDB, but now, with the *Aggregation Framework*, this can be done with two special stages, **`$out`** and **`$merge`**.

Transferring via the Aggregation Framework

With either the **`$out`** or **`$merge`** stage, you can "copy" the results of a pipeline into a collection:

```
> db.cookbook.aggregate([
  { "$match": { "type": "Breakfast"} },
  { "$out": { "db": "recipes", "coll": "breakfast" } }
])
```

This will take all the matching documents in the **`cookbook`** collection and copy them into the *new* **`breakfast`** collection in the **`recipes`** database. You could omit the **`$match`** stage and all documents would be copied.

This will not *delete* the documents from the **`cookbook`** collection, but will instead *create or replace* the target collection, with your pipeline's results. If you do not want to replace the target collection you can instead use **`$merge`**:

```
> db.cookbook.aggregate([
  { "$match": { "type": "Breakfast"} },
  { "$merge": {
    "into": { "db": "recipes", "coll": "breakfast" }
  } }
])
```

The **`$merge`** stage has several options for how this merge behaves, but in short, it can be safely used to transfer documents into an existing collection.

Archiving documents

You can use either stage along with any other stages to construct a complex pipeline of "cleaning" or "transforming" of documents before they get inserted into the final target collection. As a simple example, this pipeline would take all the documents older than *January 1st, 2023*, modify them by adding two new archive fields, and then copy them to the **`archive`** database's **`cookbook`** collection:

```
> db.myCollection.aggregate([
  { "$match": {
    "createDate": { "$lt": ISODate("2023-01-01") }
  }},
  { "$addFields": {
    "archived": true,
    "archivedDate": new Date()
  }},
  { "$merge": { "into": { "db": "archive", "coll": "cookbook" } } }
])
```

After this transfer completes, you can confirm if everything looks fine by querying the archive database's collection. When you confirm the documents have been properly moved, you can then delete those documents in the source collection using the same query filter as the **`$match`**:

```
> db.myCollection.deleteMany({"createDate": { "$lt":
ISODate("2023-01-01") }})
```

An important thing to note is that both **`$out`** and **`$merge`** need to be the *last stage* in a pipeline.

Conclusion

You should now be empowered to do all sorts of import and export operations on your MongoDB database. We covered using the MongoDB Compass UI, the MongoDB Database Tools, and the Aggregation Framework to import, export and transfer documents in and around your databases and collections.

In the next chapter, we will dive deeper into how to configure, monitor, and troubleshoot MongoDB.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 11

Make It Great – Configuration and Monitoring

"A sum can be put right: but only by going back till you find the error and working it afresh from that point, never by simply going on."

— C.S. Lewis

"I believe in having each device secured and monitoring each device, rather than just monitoring holistically on the network, and then responding in short enough time for damage control."

— Kevin Mitnick

Introduction

As with any server implementation, being able to properly configure, monitor and tweak your setup is key to long term success. How do you change where your databases' files are stored on disk? How do you change the port MongoDB uses? Is there a way to know how many updates happened in the last 5 minutes? What are your server's biggest bottlenecks? Can you chart your databases' usage? How can you change your network settings? We will cover all this and more in the following chapter.

Structure

In this chapter, we will discuss the following topics:

- MongoDB Server Operations
- Configuration
- Monitoring MongoDB

Objectives

More than likely, you have set up your MongoDB server to automatically start up, and you have not thought about it since. That is a great thing about MongoDB in general; a lot of the time, it should "just work". However, it is important to know how the server works, so that you can best configure and generally troubleshoot your server.

In this chapter, we will consider how the server's start, stop and restart processes work, as well as how to configure your server. We will focus on important settings that you will need to know, in order to properly administrate your server, as well as various monitoring tools and tips for troubleshooting.

MongoDB server operations

We have covered some start/stop operations in the opening chapters, but as a review let us go over them once again.

Starting the MongoDB server

Depending on your operating system, you may need to use different commands to start your MongoDB Server.

For most Linux based installs, you should be able to run start or restart using the following:

```
$ sudo systemctl start mongod
$ sudo systemctl restart mongod
```

For macOS, using brew you can run the following:

```
$ brew services start mongodb-community@6.0
```

Lastly, for Windows, go to the *Services Console,* find the MongoDB service, right click on the MongoDB service and click "Start".

Stopping the MongoDB Server

No matter what your operating system is, you can shut down MongoDB, assuming you have access via the MongoDB Shell:

```
> use admin
> db.shutdownServer()
```

Note that we switched to the **`admin`** database. While this may not be necessary in your exact setup or connection, in most cases, not using the **`admin`** database will result in an error. If you are using a *Replica Set*, the operation might be a bit different, but we will cover that in *Chapter 12, Seamless Scaling – Replication and Sharding.*

If you are using an operating system such as Linux, you will most likely be able to stop with a command like one of these two:

```
$ sudo systemctl stop mongod
$ sudo service mongod stop
```

If you are using macOS, using brew, you can run the following command:

```
$ brew services stop mongodb-community@6.0
```

For Windows, go to the *Services Console*, find the MongoDB service, right click on the MongoDB service and click "Stop", or "Pause".

MongoDB Server binary

In each of these cases, the start/stop operations are telling the MongoDB Server program, known as a "binary", to start or stop running.

For most operating systems, this binary is called **`mongod`** and on Windows, it is **`mongod.exe`**. This **`mongod`** will start the MongoDB Server "process" and "serve" the database. The database process will listen on whatever port you assigned it to (**`27017`** by default) and control the reading and writing of the *MongoDB Data Files*, which are the data files for your database, on disk.

The importance of ports

If you are unclear what a port is, it is like a particular channel that can be "tuned into" on a particular address, on a network, to send and receive commands and data. You actually use them all the time, but they are often hidden away from you. For example, a typical web server runs on ports **`80`** and **`443`**, for **`http`** and **`https`** respectively. Any time you browse to a **`https://`** website in your browser (by default), that will use port **`443`**. For simplicity, the browser does not require you to know or type out this port number, since it can be assumed.

For MongoDB, this default port is **`27017`**, which you can change, or possibly your MongoDB setup may use multiple ports as part of a Replica Set, which we will cover in *Chapter 12, Seamless Scaling*.

Other MongoDB binaries

There are other binaries that may be in use, depending on your given setup, if you are using Sharding, or Replication, and so on.

Configuration

In the next section, we will examine how to configure your MongoDB Server.

Server binary and data

You may be wondering where exactly the server "lives" on your system. This was briefly touched when we installed MongoDB. It is different depending on your operating system, or sometimes due to other factors as well. Generally speaking, if you have used the recommend install processes, you can find your server binary, configuration file as well as data and log files, in predictable places.

The following tables show the default location for both the database "binary" or the actual program that runs the server, and the location where that binary stores its data:

Linux/Docker

`Binary`	`/usr/local/bin/mongod`
`Config`	`/etc/mongod.conf`
`Data`	`/usr/local/var/mongodb`
`Logs`	`/usr/local/var/log/mongodb`

Table 11.1: *Location for Linux/Docker*

MacOS (Intel)

`Binary`	`/usr/local/opt/mongodb-community/bin/mongod`
`Config`	`/usr/local/etc/mongod.conf`
`Data`	`/usr/local/var/mongodb`
`Logs`	`/usr/local/var/log/mongodb`

Table 11.2: *Location for MacOS (Intel)*

MacOS (Apple Silicon)

Binary	`/opt/homebrew/bin/mongod`
Config	`/opt/homebrew/etc/mongod.conf`
Data	`/opt/homebrew/var/mongodb`
Logs	`/opt/homebrew/var/mongodb`

Table 11.3: *Location for MacOS (Apple Silicon)*

Windows

Binary	`C:\...[install directory]\bin\mongod.exe`
Config	`C:\...[install directory]\bin\mongod.cfg`
Data	`C:\...[install directory]\data`
Logs	`C:\...[install directory]\logs`

Table 11.4: *Location for Windows*

MongoDB Data Files

Inside the folder assigned to **`dbpath`**, MongoDB will store its data files. This will generally include a number of **`.wt`** files, which includes the database and collection data, as well as separate index files, a **`journal`** directory and various **`.lock`** files.

It is important that the location of your **`dbpath`** is somewhere that has plenty of disk space to store all these files, and ideally will have some sort of RAID or redundancy. A lot of MongoDB runs in RAM, but making sure you have a good disk setup is still essential.

Command line

One of the simplest ways to control the options of the MongoDB Server is to use "flag" options when you start **`mongod`**. For example, to change port and the location of where the data files are stored, you could run MongoDB, as shown:

```
$ mongod --port 28018 --dbpath /store/mongodb
```

Now when MongoDB starts up, it will "bind" to the port **`28018`** and either look for, or if they are not there, create the data files for your database in the **`/store/mongodb`** directory.

You can set nearly every possible option in this fashion, but it can become unwieldy and error prone. For most configurations, it is suggested you use the MongoDB configuration file. You can find the default location of the configuration file in the

tables mentioned above, or you can specify your own when starting the MongoDB server by passing the config option:

```
$ mongod --config=/path/to/file
```

The exact path formatting will depend on your operating system, and generally speaking, it is best practice to edit and use the file in its default location. Editing the file in the default location should mean you do not need to tell MongoDB where to find its configuration file.

If you download and run the binary of MongoDB directly, it has certain defaults such as port **`27017`** and a **`dbpath`** of **`/data/db`** or **`C:\data\db`** even if no configuration file is specified.

Configuration file

After finding and opening the MongoDB configuration file, you should see a number of defaults already set, but these are not the only options you can set. There are many more. If you change any options in the configure file, you will need to stop and restart your MongoDB Server before any changes will take effect.

File Format

The configuration file itself uses a format called **`YAML`** which uses indention to organize options and values. It is actually a lot like a large, nested **`JSON`** object, but without any brackets or other such formatting. An example of **`YAML`** might look like this:

```
category:
  option: "value"
  another: true
somethingElse:
  sub:
    enabled: false
yetAnother:
  lookHere: "/var/lib"
```

Note: the YAML format uses spaces, not tabs for indentions.

Server defaults

As previously stated, even without a config file, certain options will be defaulted and depending on your install, certain options will also be set in your default config file. A couple examples might be like the following:

```
storage:
  dbPath: /var/lib/mongodb
  journal:
    enabled: true
systemLog:
  destination: file
  logAppend: true
  path: /var/log/mongodb/mongod.log
net:
  port: 27017
  bindIp: 127.0.0.1
processManagement:
  fork: true
  timeZoneInfo: /usr/share/zoneinfo
```

Common Options

In the next couple of sections, we will highlight some of the most common options. For a deeper dive, consult the official MongoDB documentation:

https://www.mongodb.com/docs/manual/reference/configuration-options/

Storage Options

As shown in the previous example, the config file groups options in categories to make this management easier, and sometimes those categories have subgroupings of their own. One such category is **`storage`** which we will look at first:

```
storage:
  dbPath: /var/lib/mongodb
  journal:
    enabled: true
  directoryPerDB: true
```

This will not only set the **`dbpath`**, but also make sure "journaling" is enabled. In simple terms, journaling is the enabling of a special log that lives separately from your databases' data files, and can be used to keep those files from becoming invalid, or can be used to "recover" them if they do become corrupt. This option is on by default for MongoDB, and you would only want to turn it off in special cases.

After that, you can see an option called **directoryPerDB**; this option (which is off, or **false** by default) will group all of a database's data files in a folder of their own. Otherwise, MongoDB will write all the database files, for all databases and collections to the **dbpath** folder.

Setting the **directoryPerDB** can be useful if you want to make sure to separate out your data by database, and perhaps even mount different folders to different locations on disk. For example, assume this setup:

```
/var/lib/mongodb
- admin
- archive
- book
- local
- test
```

Here we have a **/var/lib/mongodb** folder with five folders inside it, each named for a database. Inside each of those database folders are the data files for that database and its collections. We could technically, using something like a *symlink*, or other method, and make it so that those folders were actually somewhere else, or on another disk like so:

```
/var/lib/mongodb
- admin
- archive  -> /disk2/archives/mongodb
- book     -> /store/db/book
- local
- test
```

Now the **archive** database folder, and all its contents are actually on **disk2** (perhaps less expensive, long term storage) and the **book** database folder and all its contents are using **/store/db/book**, which might be on a more expensive, faster piece of storage. The rest of the folders will stay on whatever disk the **dbpath** is on.

There are many other storage options, especially storage engine options you can tweak in advanced use cases.

Network options

Another commonly adjusted section of options is the **net**, or networking category:

```
net:
  port: 27017
```

```
  bindIp: 127.0.0.1
  maxIncomingConnections: 65536
```

The **`bindIp`** option is of particular interest here, as it tells MongoDB which network address to "attach" itself to and listen for requests on. This is very important because if it is not set up correctly, you might not be able to connect to your MongoDB at all; or if set too liberally, it may leave your server "too open" and therefore, allowing services, or people that you do not want to be able to connect to your database, access to your database.

As set in this configuration, MongoDB will listen only on the "localhost" IP or **`127.0.0.1`**, meaning if you, or whatever service or program you are running are on the same machine as MongoDB, you can connect. If you are on another machine, even on the same internal network you will not be able to connect to MongoDB.

You can set multiple values for **`bindIp`** like this example which will allow connecting from the localhost, or an internal IP **`192.168.2.112`**:

```
net:
  bindIp: localhost, 192.168.2.112
```

Now, if we have a webserver running code in the same internal network, it can connect to MongoDB using the server's internal IP. You can add various addresses to this comma separated list if you wish, even external IP addresses. However, to keep things secure, you will want to avoid this in most cases.

Another choice is to use the **`bindIpAll`** or **`--bind_ip_all`** (on the command line) option, which will, as the name implies, start to "listen" on any IPv4 addresses. This is how MongoDB inside Docker runs by default because you explicitly "expose" ports running on a Docker container to the outside. For a process running on a typical server, you will generally want to specify what IPs MongoDB will listen on.

```
net:
  maxIncomingConnections: 65536
```

The **`maxIncomingConnections`** option is another one you might need to tweak every so often as it lets you limit (or increase) the number of connections the database will accept. Generally, it is fine not to set this, and assume the default.

Logging options

An often overlooked aspect of running a database is its logs, however if, *and really when* you eventually run into a problem, logs can be priceless. A typical log configuration might look like this:

```
systemLog:
  destination: file
```

```
  logAppend: true
  path: /var/log/mongodb/mongod.log
```

Here, we are saying the "type" of logging, the **`destination`**, will be a MongoDB log **`file`**. We will also need to let MongoDB know where it should store this file, that is, the **`path`**. An important factor you may want to consider is, if your log file is on the same disk as your database files. In certain cases, this can be a problem.

For example, if either your data files or log files get too large, they may start to complete with each other as you quickly run out of space on disk! If possible, it is recommended to store these on two different disks to avoid contention. You can also set **`destination: syslog`** if you wish to use your systems' log, but this is not generally recommended.

Another option of note is the **`logAppend`** option. If set to **`true`**, when **`mongod`** restarts, this will "reuse" the log file by adding new entries to the end of the file. If set to **`false`**, it will use a new file, keeping, and essentially backing up the old log file. You will want to consider which behavior is ideal for your server setup.

Process management options

The configuration category of **`processManagement`** configures how the binary program, or "process" functions:

```
processManagement:
  fork: true
  timeZoneInfo: /usr/share/zoneinfo
```

The most important option here is **`fork`**, which will run MongoDB as a "background process" or daemon. Running as a background process is the typical way to run MongoDB as it does not require a logged in user to start the MongoDB binary, and also makes sure any output does not get written out to the screen, but rather only to the log files. The **`fork`** option is not supported on Windows but running as a Windows Service has a similar effect.

The **`timeZoneInfo`** option will let you ensure your database has all the latest time zone information, which surprisingly can change rather often. In this case, we have linked it to our operating systems time zone information, which should update itself fairly regularly. Otherwise, MongoDB will use its own built-in database of this information that is updated on each MongoDB release (as in each new version, as you upgrade to it).

Replication

We will cover Replication and Sharding (the concept of breaking up your documents, across servers) in the next chapter, *Chapter 12, Seamless Scaling – Replication and*

Sharding, but as a reference, there are special groups for these features as well. For examples, you might see something like this in a Replication setup:

```
replication:
   replSetName: myReplicaSet
```

Security Options

While this we will not cover security in depth in this chapter (we will do that in *Chapter 13, Being Proactive – Security and Backups*), there are a number of security related configuration options. Here is a small example:

```
security:
  javascriptEnabled: false
  authorization: true
```

Both of these examples are *overrides* of the default configuration. Out of the box, MongoDB will allow sever-side JavaScript execution, like the JavaScript we have used via the MongoDB Shell, as part of a query. This can be super helpful at times, but it also carries some security risks. If you know you will not be using JavaScript server-side, it is a good idea to disable this feature via the **`javascriptEnabled`** option.

Along the same lines, *User Access Control*, also referred to as *Role-Based Access Control*, is not enabled by default for MongoDB. *User Access Control* allows you to restrict access, or control levels of access to a database by user account. The authorization option will enable this feature, and more than likely, you will want to do this so you can create different user accounts for different databases, such as read only accounts.

As we will discuss in *Chapter 13, Being Proactive*, authorization and authentication are not the same thing. Authentication is a challenge to prove you are who you say you are, like being asked for a correct password. If you can provide the correct password, your user account has been authenticated. Authorization is controlling what things you can do, or your "privileges", after your identity is confirmed. These privileges may be actions like performing insertions or deletions on a database.

Windows service options

If you are using the Windows operating system, there are some options exclusively for configuring the MongoDB Windows Service. These will be set for you automatically via the install wizard, but if you need to adjust this manually, you can do so with the following options:

```
processManagement:
   windowsService:
```

```
serviceName: <string>
displayName: <string>
description: <string>
serviceUser: <string>
servicePassword: <string>
```

Externally sourced config

A really useful feature of MongoDB is its ability to get its configuration values from an "external source" such as a script or network request. This is not running a script to startup your server, but rather allowing you to get particular configuration values from a script or by calling a web service.

You can even get the entire config settings from an external location. To do so, you will need your config file to have this (and only this) inside it:

```
__rest: "https://yourdomain.com/serverstuff/mongodb/config"
type: "yaml"
```

The **__rest** will connect to a URL and download the contents as the value. Here the **type** is set to **yaml** since it has multiple values. Similarly, you can get the config contents from a locally executed script with **__exec**:

```
__exec: "python /store/scripts/mongodbConfig.py"
type: "yaml"
```

These same options can be use more granularly for individual config values as well:

```
net:
  port:
    __rest: "https://yourdomain.com/serverstuff/getmyport"
    type: string
```

Here, the **type** is **string**, as it is a singular value. The same sort of syntax can be used with **__exec** to get the value from a script.

Monitoring MongoDB

MongoDB provides a number of database commands as well as command line tools for monitoring your MongoDB server instance. There are many monitoring options, and so, always make sure to check out the official documentation to see everything that is available. In the next section, we will highlight some of the most useful commands.

Database statistics

Reviewing and tracking statistics and metrics in your database is extremely useful not only to know what is going on right now in your database, but what is going on long term or if there are outliers you should investigate.

The dbStats command

To get a quick snap shot of what is in your database, you can use the **`dbStats`** command:

```
> db.runCommand( { dbStats: 1 } )
{
  db: 'book',
  collections: 8,
  views: 0,
  objects: 38,
  avgObjSize: 493.57894736842104,
  dataSize: 18756,
  storageSize: 262144,
  indexes: 10,
  indexSize: 319488,
  totalSize: 581632,
  scaleFactor: 1,
  fsUsedSize: 28314783744,
  fsTotalSize: 62671097856,
  ok: 1
}
```

You can use these stats to see if you need to adjust your disk size; or if tracked over time, see changes in document size, or number of documents, and so on.

The serverStatus command

For a robust report of just about everything happening on your MongoDB server, use the **`serverStatus`** command:

```
> db.runCommand( { serverStatus: 1 })
```

The output alone, which is a large document, would take up a few chapters of this book! So instead, we will focus on a couple areas of interest. Since it is a document, we can use dot notation to get back just certain portions of the output. Let us start off with some of the high-level stats, such as **`version`** and **`uptime`**:

```
> db.runCommand( { serverStatus: 1 }).version
6.0.3


> db.runCommand( { serverStatus: 1 }).uptime
452400
```

The **`uptime`** is the number of seconds that the **`mongod`** binary has been running. Next, we can check the stats on database connections:

```
> db.runCommand( { serverStatus: 1 }).connections
{
  current: 17,
  available: 838843,
  totalCreated: 128,
  active: 4,
  threaded: 17,
  exhaustIsMaster: 0,
  exhaustHello: 3,
  awaitingTopologyChanges: 3
}
```

Or perhaps you are a fan of metrics? Good news, **`serverStatus`** has *a lot* of those! Let us look at a couple of them, such as how many times different database commands have been run:

```
> db.runCommand( { serverStatus: 1 }).metrics.commands
{
  ...
  aggregate: { failed: Long("0"), total: Long("33") },
  ...
  delete: { failed: Long("0"), total: Long("2") },
  ...
  find: { failed: Long("0"), total: Long("1084") },
```

```
  ...
  insert: { failed: Long("0"), total: Long("1") },
  ...
  update: {
    arrayFilters: Long("0"),
    failed: Long("0"),
    pipeline: Long("88"),
    total: Long("18")
  },
  ...
}
```

In this example, the output has been abbreviated (as it would take up a couple pages), but this will return metrics on just about any possible command you can run on your database. You can also target just a particular command, such as **`find()`**:

```
> db.runCommand( { serverStatus: 1 }).metrics.commands.find

{ failed: Long("0"), total: Long("1086") }
```

You can get a similar accumulation for just actions on documents using **`metrics.document`**:

```
> db.runCommand( { serverStatus: 1 }).metrics.document
{
  deleted: Long("3"),
  inserted: Long("11"),
  returned: Long("165"),
  updated: Long("14")
}
```

Both of these can be useful in troubleshooting what sort of operations are happening most often on your database. Or, if gathered and plotted over time, allow you to see spikes of usage. There are various monitoring tools that will do this for you; more on that later in this chapter.

Another metric to keep track of is collection scans, which as we discussed in *Chapter 9, Planning for Performance – Collections and Indexes*, means a query was unable to use an index :

```
> db.runCommand( { serverStatus: 1 }).metrics.queryExecutor
```

```
{
  scanned: Long("125"),
  scannedObjects: Long("221"),
  collectionScans: { nonTailable: Long("29"), total: Long("29") }
}
```

You can use these stats, along with other query metrics and index stats to see if you can improve the situation with proper indexes.

Command line tools

If you love the command line, you are in luck. There are two programs which are available as part of the *MongoDB Database Tools,* that complement the database commands we have been working with.

The mongotop command

To get a snapshot of what is going on in your database from the command line, you can use **`mongotop`**. By default, it will return results every second. To adjust the frequency, pass a number of seconds; for example, pass **`300`** for results every five minutes:

```
$ mongotop 300
2023-01-08T23:52:27.021+0000    connected to: mongodb://localhost/

                    ns    total     read    write    2023-01-08T23:57:27Z
          book.cookbook    234ms    234ms      0ms
              test.neo     45ms     45ms      0ms
             book.items     35ms     35ms      0ms
```

You can see an example of the default one second frequency in the following *Figure 11.1*:

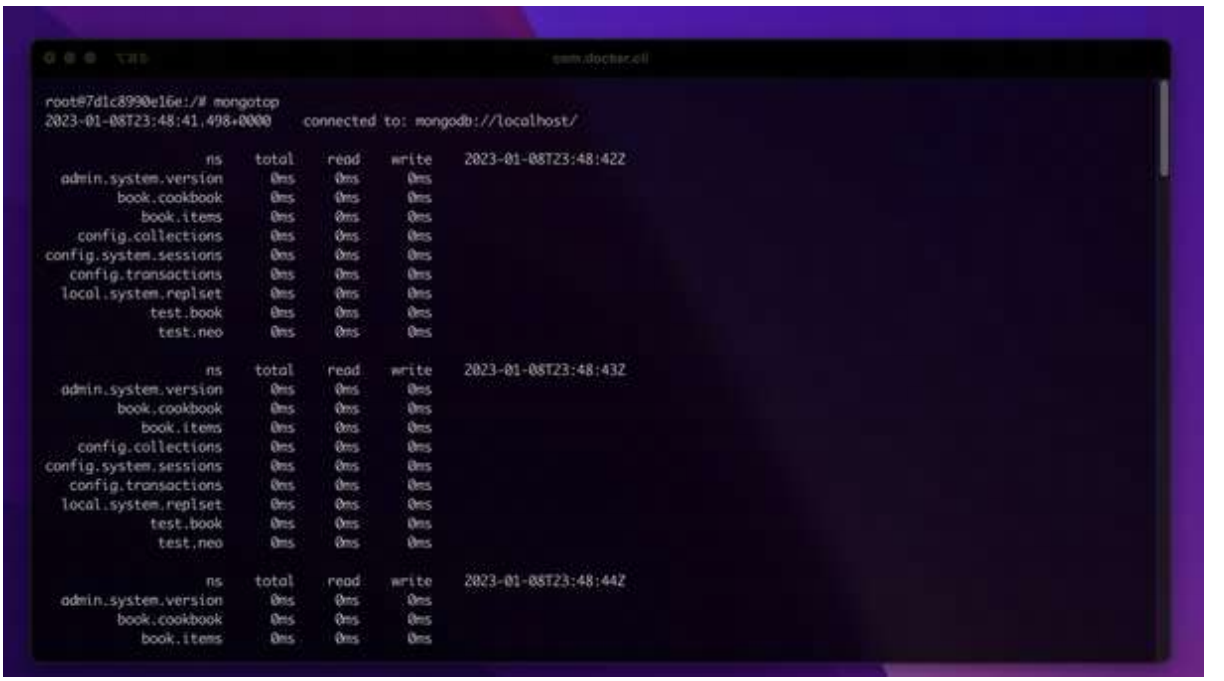


Figure 11.1: *The mongotop Command*

You can also pass the **`--json`** option to get back the results in **`JSON`**.

Note: They will not be formatted. As with most of these programs, you can pass the --uri option to set a connection string for your specific server.

The mongostats command

For a condensed, regularly updated subset of the **`serverStatus`** metrics, the **`mongostats`** command is available via the command line. It works a lot like **`vmstat`** if you are familiar with that from Linux. It also takes the option of seconds, as well as **`--uri`** if you need to pass a connection string.

If no frequency is passed, it will gather and output stats, such as inserts, queries, updates and deletes, every second, occasionally outputting the headers again as seen in *Figure 11.2*:

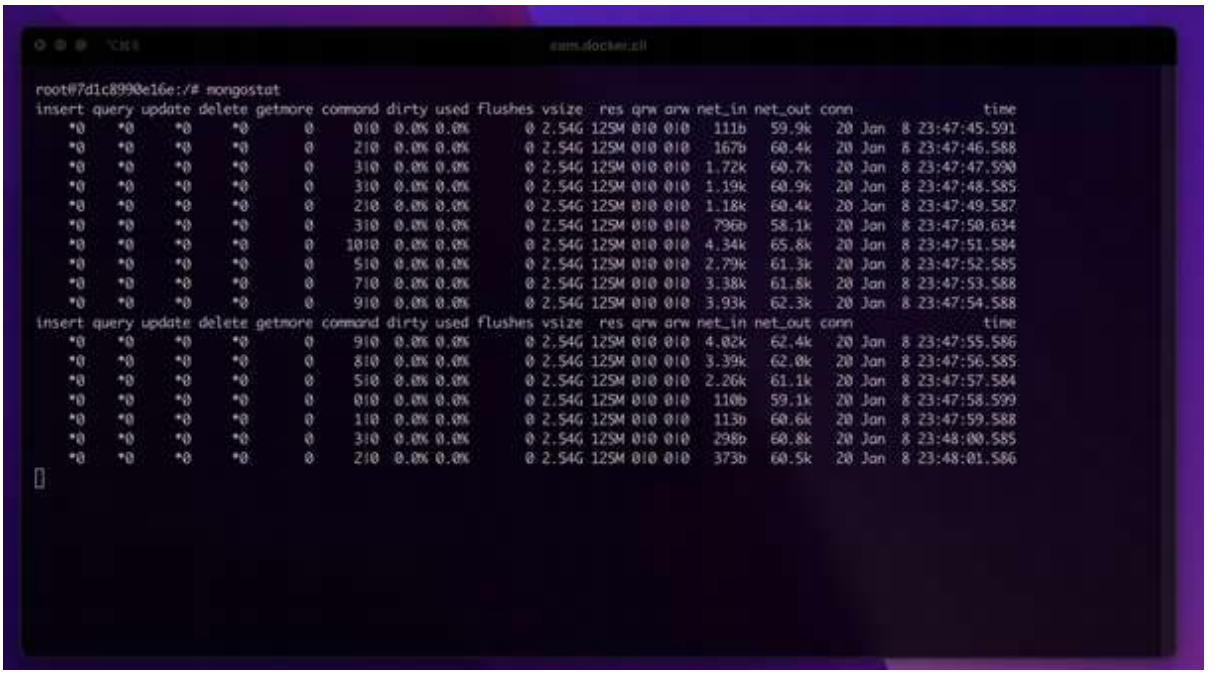


Figure 11.2: *The mongostat Command*

You can open a terminal, set a frequency and watch the results as you perform different actions on your database, or watch live to see how much load your server is receiving.

Monitoring software

There are a number of free or paid tools for server monitoring that support MongoDB. One free tool built into MongoDB is *Atlas Free Monitoring*. The first time you login to the mongosh shell, you should see a message letting you know how to set up this monitoring, or you can run this command:

```
> db.enableFreeMonitoring()
```

You can learn more details about this free MongoDB service here:

`https://www.mongodb.com/docs/manual/administration/free-monitoring/`

MongoDB's Free Monitoring will use the same data we have been reviewing but makes it a little more useful by providing helpful charts, and trends over time. The free version keeps this data for 24 hours. You can get an idea of what these charts look like in the following *Figure 11.3*:

Figure 11.3: *MongoDB Free Monitoring*

There are many free, open source or paid monitoring services that you can install yourself or use a software as a service. Some suggestions are Nagios, Prometheus, New Relic, Datadog and Cacti.

Conclusion

In this chapter, we have learned about how to operate and configure your MongoDB Server as well as how to monitor its operations and performance.

In the next chapter, we will use this knowledge to expand beyond a single server into a *Replica Set*, a set of servers that all duplicate each other for better redundancy and performance. We will also learn about the concept of "sharding", which can automatically balance our data across multiple servers based on a specific field in each document.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 12

Seamless Scaling – Replication and Sharding

"Failing well means ending something that is not working and choosing to do something else better."

— Henry Cloud

"Gelfling, you must find the shard ... before the three suns meet. If not, Skeksis rule forever."

— UrSu, The Dark Crystal

"Better to have, and not need, than to need, and not have."

— Franz Kafka

Introduction

The sign of a true professional is knowing that even in the best planned systems, something will fail, at some point. Perhaps it is a simple issue of a log filling up a server's disk, or maybe it is something much more serious. The best way to plan for failure is by not having a "single point of failure", so that even if one thing fails, something else can take its place.

Perhaps your problem is database load. Your application is growing, but upgrading to a new and larger server is very costly. Is there any way to scale your database without constantly upgrading to more and more expensive servers? In this chapter,

we will discuss two core concepts for scaling MongoDB, both for better resilience and for better performance.

Structure

In this chapter, we will discuss the following topics:

- Reducing risk with replication
    - Replica sets
    - Initiation and configuration
    - Initiating a replica set
    - Member roles and types
    - Administration
- Scaling with sharding

Objectives

In this chapter, we will cover the core concepts of replication in MongoDB, discuss how to setup a replica set and how to leverage the concept of "sharding" for horizontal scaling, using MongoDB. By the end of this chapter, you should be comfortable setting up a basic replica set and have a knowledge of how to administrate that replica set. You will also have a solid idea of how sharding works in MongoDB and how and why you would use it for your application.

Reducing risk with replication

Thus far, we have been working with a single MongoDB Server, which is also known as a "standalone" server setup. This sort of setup is reasonable and acceptable for local development or testing things out, although for production environments, it represents an unacceptable "single point of failure".

If anything happens to your single server, be it a networking issue, some sort of crash, or some other unexpected event, the application or website will experience downtime. Worse yet, getting your single server back online may take an unknown amount of time, depending on what caused the outage in the first place.

To avoid this *single point of failure*, MongoDB uses a concept called database replication. You may be familiar with this concept from other database systems. However, if you are not acquainted with replication, it is the concept of copying the contents of a database between multiple other servers. Then, if a single server goes down, there is another server available to take its place with the exact same setup and data. In MongoDB, the best way to accomplish replication is by using what are called Replica Sets.

Replica sets

The basic idea of a *Replica Set* starts with a *Primary* server. The primary acts as a single conduit for database writes and reads. Moreover, it also constantly replicates, or copies its data to one or more other servers, called secondaries. You can see a simple example of this setup in *Figure 12.1*:

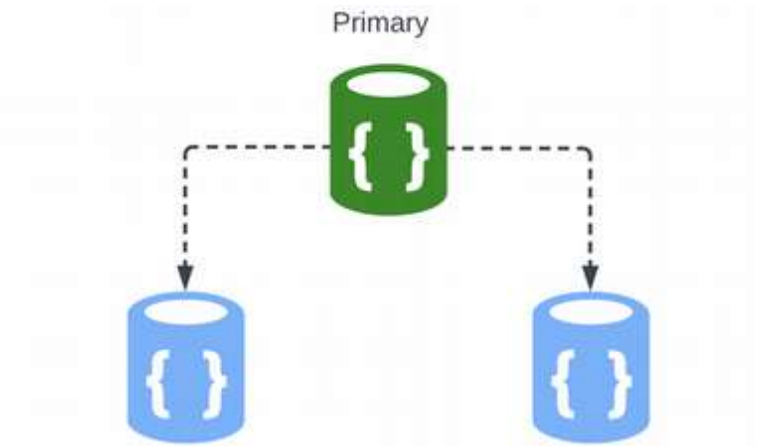


Figure 12.1: *Basic Replica Set*

While there are various ways to setup a replica set, the most common one is to start with at least three **`mongod`** instances, or nodes. You can run each node on the same physical server, but that is still a single point of failure. So ideally, each node will run on a different physical server. We will discuss setting up a replica set later in this chapter. For now, let us cover how a replica set works.

Primaries, secondaries and elections

When the replica set starts up, each node will communicate with all the other nodes, exchanging data on their configurations, and so on. Following that, amongst themselves the nodes will immediately perform what is called an "election". This election will determine which node will be the *primary*. We will discuss more in a later section about how this election works but suffice it to say, in our three node configuration, one of the three nodes will be elected the *primary* and the other two will become *secondaries*.

We now have a three-node replica set, with the data from the primary constantly being duplicated to the secondaries.

Automatic failover

You might now be wondering what happens if a primary node goes down for some reason. Firstly, this could happen because there is a problem on the physical server, or maybe it was on purpose for maintenance, or a rolling upgrade.

In the three-node configuration we have been discussing, if the primary encounters a problem and does not communicate to the other members of the set within the **`electionTimeoutMillis`** value (which is **`10`** seconds, by default), a new election will be called. Each node still running will take an account of various factors such as the last time it synced with the primary, any configuration priority you have set, and so on. It will either vote for itself to be the new primary or vote for another member. In the case of a tie, the election will be re-run until a node wins, and becomes the new primary.

After the new primary is elected, the remaining nodes become, or remain, secondaries, replicating off the new primary as shown in *Figure 12.2*:

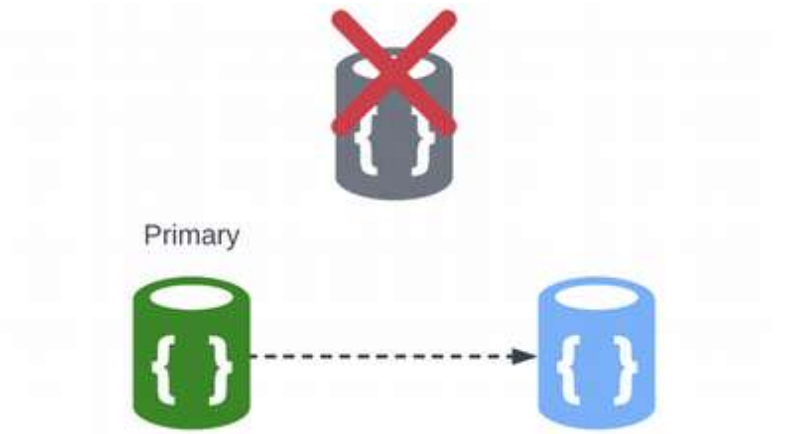


Figure 12.2: *Failure of a Primary, and Election of a New Primary*

To become a primary node, a node must receive a majority of votes. In this example, there are 3 nodes, and so for any node to become a primary, it will need at least two votes to have a majority. For this reason, it is a best practice to always have an odd number of nodes in your set. For an illustration of why this matters, assume a four node set, as seen in *Figure 12.3*:

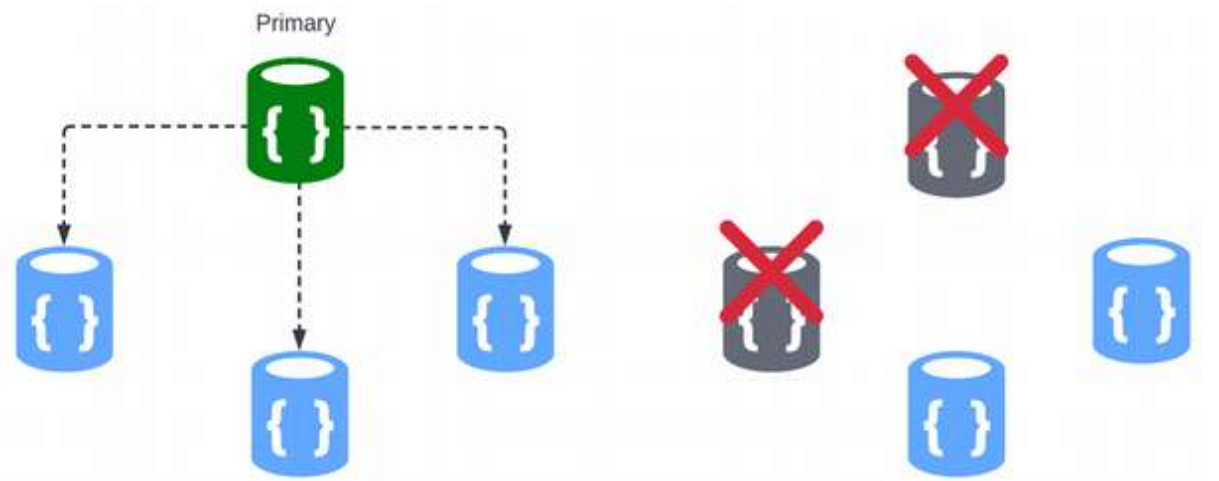


Figure 12.3: *Four Node Replica Set*

You may be thinking that by adding another node, this replica set is more resilient. However, it provides the same amount of protection as three nodes, as seen on the right side of *Figure 12.3*.

With either three nodes or four nodes, *you can only lose one node* and still have a majority. If you have four nodes and lose two, a majority is not possible, since it will be two votes out of four.

If you have five nodes, a majority is possible, and you could lose up to two nodes, and so on.

You can have as many as 50 nodes in a replica set, but only up to 7 of those nodes can vote.

Initiation and configuration

To setup a replica set, you will need to fill out a couple of extra options in the MongoDB configuration file, and run multiple **`mongod`** binaries at the same time. Each **`mongod`** will be a node, and you can use the configuration file of each, in order to control various aspects of your replica set.

In this section, we will setup an example replica set. Feel free to follow along on your local machine, in Docker, or however you choose.

Note: You will need to be able to run the `mongod` binary from the command line to follow these steps. Refer back to Chapter 11, *Make It Great – Configuration and Monitoring* to learn how to do this, as well as to know how to find the binary's location on your system.

Replica set configuration

Here we have an example configuration file we will call **`node1.conf`**, with some extra options for a replica set:

```
net:
  port: 27021
  bindIp: localhost
processManagement:
  fork: true
storage:
  dbPath: /store/mongodb/node1
systemLog:
  destination: file
```

```
  path: /store/logs/node1/mongod.log
  logAppend: true
replication:
  replSetName: myReplSet
```

We will use this for the first node in our set. A few of the options to focus on, are the **`port`** as well as the **`dbPath`** and **`path`** for the logs, which is where the node's files will be stored. Since we are running these all on our local machine, all of these *should be unique* for each node. You can adjust the paths to make sense for your server setup, and of course, if you are running your nodes on *separate physical servers,* you can use the same paths.

The final option is the **`replSetName`**, which is very important and *must be the same* for each node. You can name this whatever you would like, but here we are calling our set **`myReplSet`**.

We can start up this node as follows:

```
$ mongod --config=/store/mongodb/node1.conf
```

The exact path for your config file will depend on your system, but here we are assuming it is in a folder called **`mongodb`** inside a **`store`** folder. This will launch a MongoDB server running on port **`27021`**, which you will recall is not the default MongoDB port. This is because we will be running two more servers, and they cannot all share the same port on a local server.

Note: If you run each node on a separate server, which you will want to do in a production setup, using different ports is not necessary. However, for our local example, this is required.

Next, we will repeat this process two more times, with slight changes to our configuration files. For example, **`node2.conf`** will look slightly different:

```
net:
  port: 27022
  bindIp: localhost
processManagement:
  fork: true
storage:
  dbPath: /store/mongodb/node2
systemLog:
  destination: file
```

```
  path: /store/logs/node2/mongod.log
  logAppend: true
replication:
  replSetName: myReplSet
```

For **node3.conf**, using the same convention, we will set the port to **27023**, and use **node3** folders and so on. You can then start each **mongod** with its own config file, as shown here:

```
$ mongod --config=/store/mongodb/node2.conf
$ mongod --config=/store/mongodb/node3.conf
```

This should output something like this:

```
about to fork child process, waiting until server is ready for connections.
forked process: 2809
child process started successfully, parent exiting
```

You should now have three MongoDB servers running on ports **27021**, **27022** and **27023**, respectively. You can see these processes by running a program like **top** or **htop** on the command line, as shown in *Figure 12.4*, or another process management program:

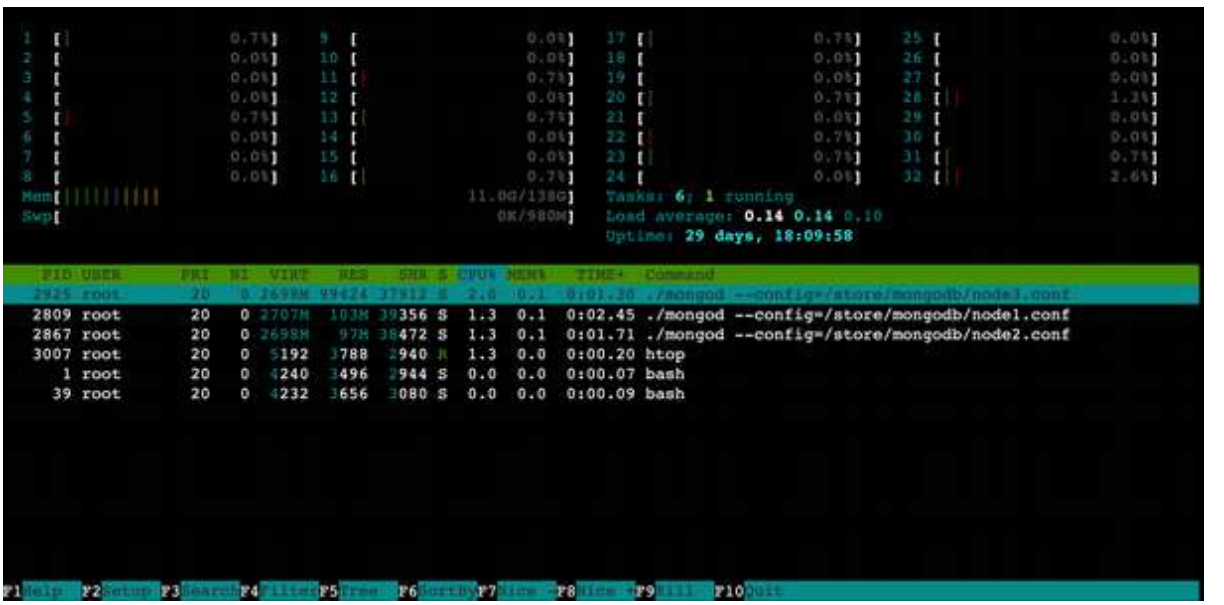


Figure 12.4: *Viewing mongod Processes in htop*

Initiating a replica set

Now that we have our three nodes running, we need to tell MongoDB to treat them as a set. To do so, connect to one of your nodes (it does not matter which), and run the **`rs.initiate()`** command with a replica set configure document like this:

```
> rs.initiate({
  "_id": "myReplSet",
  "version": 1,
  "members": [
    { "_id": 1, "host": "localhost:27021" },
    { "_id": 2, "host": "localhost:27022" },
    { "_id": 3, "host": "localhost:27023" }
  ]
})
```

Here, we have given our set a name **`myReplSet`**, a **`version`** (which we will update if we change this configuration) and an array of **`members`**. These members are represented by a document with a unique **`_id`** and the host and port the node is running on. If this command succeeds, you can check the status using the following command:

```
> rs.status()
```

We will discuss the output of this command more in the next section. For now, you can disconnect from your node, and prepare to connect to the replica set proper.

Connecting to a replica set

There are various ways to connect to your new replica set. However, the most straightforward way is using a connection string. The following example shows connecting to the replica set via **`mongosh`**:

```
$ mongosh "mongodb://localhost:27021, localhost:27022,
localhost:27023/?replicaSet=myReplSet"
```

Using a comma separated list, we have specified each member of the set, which in this case, is all on **`localhost`** with their unique ports. For a production replica set, this would be whatever hostname/server that each node is running on. Once you connect, the prompt will change to something like this:

```
myReplSet [primary] test>
```

This lets you know that you are connected to the **`myReplSet`** replica set, while also being on the *primary* node. You will be automatically connected to the **`primary`** node, since this is the only one that can receive reads and writes. Alternatively, via MongoDB Compass, input the connection string as shown in *Figure 12.5*:

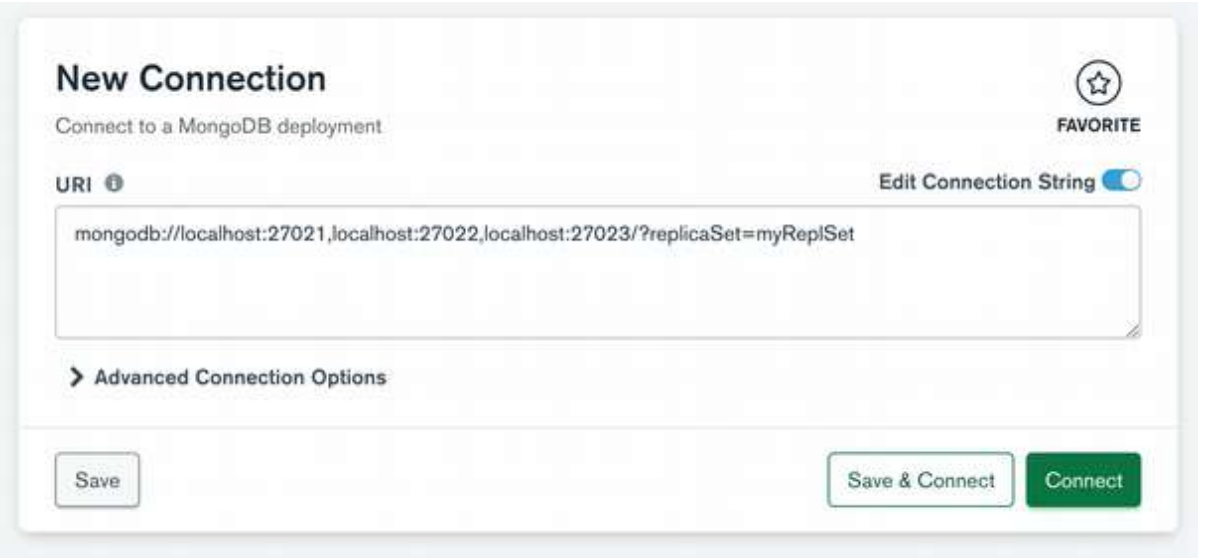


Figure 12.5: *Connecting to a Replica Set in MongoDB Compass*

Once connected, you can run some commands to get information about the set. As previously mentioned, one very useful command is **`rs.status()`**. Here is a simplified version of its output:

```
> rs.status()
{
  set: 'myReplSet',
  heartbeatIntervalMillis: Long("2000"),
  majorityVoteCount: 2,
  votingMembersCount: 3,
  optimes: {
     lastCommittedOpTime: { ts: Timestamp({ t: 1673926363, i: 1 }), t:
Long("1") },
  },
  members: [
    {
      _id: 1,
      name: 'localhost:27021',
```

```
      stateStr: 'PRIMARY',
    },
    {
      _id: 2,
      name: 'localhost:27022',
      stateStr: 'SECONDARY',
    },
    {
      _id: 3,
      name: 'localhost:27023',
      stateStr: 'SECONDARY',
    }
  ]
}
```

A couple of things of note here: firstly, the **`heartbeatIntervalMillis`**, which we briefly mentioned earlier, is the max amount of time between communications or "heartbeats" amongst nodes. While a replica set is running, not only are changes pushed from the primary to the secondaries, but each node will also constantly be "talking" back and forth between each other. If a node's heartbeat stops or is taking too long, the replica set will respond. We will cover this in more details in the section on fail-safes and disaster recovery.

Two other data points of interest are the **`majorityVoteCount`** and **`votingMembersCount`**. Since we have three voting members, the majority needs to be two in any election. The **`optimes`** object reflects various operations on the server and stats on the **`oplog`**, which we will discuss in the next section.

Lastly, we also have to consider the array of set **`members`**, which will let us know which node is the primary and some statistics on each node. We can use this information to connect directly to the primary if we wish, using the following snippet:

```
$ mongosh --port 27021
```

If you do so, you will get a slightly different prompt:

```
myReplSet [direct: primary] test>
```

Notice that the **`direct`** specifier is now in the prompt. This is an important distinction. If you connect to the set via a connection string, and the primary node goes down, or

switches nodes, queries will still work. If you connect *directly* and there is a change, you will need to either switch to connect to the set, or find out the new primary and connect directly. For this reason, in most use cases, it is recommended to always connect to the full set, using a connection string.

The local database

Once a replica set is set up and initiated, all the databases and collections on the primary database will be replicated to the secondaries except one, that is, the **`local`** database. Each **`mongod`** instance has its own unique copy of the local database, which contains collections storing various information about that instance and replica set.

Some collections such as the **`startup_log`**, stores information about how the server was started up, including the **`startTime`**, **`buildInfo`** as well as **`cmdLine`** object, which stores the configuration for the server as JSON instead of the YAML format:

```
"cmdLine": {
  "config": "/store/mongodb/node2.conf",
  "net": {
    "bindIp": "localhost",
    "port": 27022
  },
  "processManagement": {
    "fork": true
  },
  "replication": {
    "replSetName": "myReplSet"
  },
  "storage": {
    "dbPath": "/store/mongodb/node2"
  },
  "systemLog": {
    "destination": "file",
    "logAppend": true,
    "path": "/store/logs/node2/mongod.log"
  }
},
```

The oplog collection

One of the most important collections in the **`local`** database, especially when it comes to replica sets, is the **`oplog.rs`** collection, or simply known as the "oplog". It is a capped collection that contains each write operation to your MongoDB instance in a document. By default, the collection is *capped to 5% of your disk*, but you can adjust this value by setting the **`oplogSizeMB`** option in your configuration.

MongoDB uses these oplog entries as a basis for replication. Since they are stored in order, a secondary can simply "replay" each entry, to replicate all the actions the primary has performed. When the max size is reached, the oldest operation documents will be deleted, just like any other capped collection.

You can query this collection, to see what actions have happened. However, you cannot modify the collection manually. This is a fail-safe to make sure that your replica set keeps running smoothly. For example, you can run the following **`find()`** query, from within the **`local`** database against the **`oplog.rs`** collection to see inserts or updates:

```
> db.oplog.rs.find({ op: { $in: ["i","u"] }, "ns": "testo.replo" })
```

This will return documents that look something like the following example:

```
{
  lsid: {
    id: new UUID("605abe18-4f00-4ec2-bfb0-6c13166fdf18"),
    uid: ...
  },
  txnNumber: Long("1"),
  op: 'i',
  ns: 'testo.replo',
  ui: new UUID("d3184a0b-072b-4dbc-a95d-3de0961b57bb"),
  o: { _id: ObjectId("63c5f5e326fcbb455630de6a") },
  o2: { _id: ObjectId("63c5f5e326fcbb455630de6a") },
  stmtId: 0,
  ts: Timestamp({ t: 1673917925, i: 1 }),
  t: Long("1"),
  v: Long("2"),
  wall: ISODate("2023-01-17T01:12:05.807Z"),
```

```
  prevOpTime: { ts: Timestamp({ t: 0, i: 0 }), t: Long("-1") }
}
```

Each document represents a different kind of action that the server has made or performed, as well as precise data about the time and any change in **_id**, and so on. The **op** field helps identify what that action was; you can look up what each stands for in the documentation, or maybe try to see if you can figure it out yourself based on the context!

The difference between the oldest oplog entry and the newest is known as the "oplog window. If a node is disconnected, it can use this window to "match" and then sync up changes to "catch up". If the window is too old, the secondary node will have to restart from scratch, to make sure no changes are ever missed.

Changing a primary to a secondary

You might be wondering what happens if you need to *purposely* take your primary offline. Perhaps you need to perform some sort of maintenance on the server, or maybe you wish to upgrade the MongoDB version and perform some backups. The first step in taking your primary offline, or simply forcing it to be a secondary, is making sure it "steps down" properly. To determine which node is your primary, you can run the **db.hello()** command and pick out the **primary** field value, as shown here:

```
> rs.hello().primary
localhost:27022
```

The step-down process will gracefully finish any long running tasks, such as write operations, or building an index, and so on. Then it will communicate with the available secondaries to make sure that at least one of them is "caught up" to the primary's **oplog**. You can initiate this process by connecting to your primary and running the **rs.stepDown()** command:

```
> rs.stepDown()
{
  ok: 1,
  '$clusterTime': {
    clusterTime: Timestamp({ t: 1673935033, i: 1 }),
    signature: {
      hash: ...,
      keyId: Long("0")
    }
```

```
  },
  operationTime: Timestamp({ t: 1673935033, i: 1 })
}
```

If you wish, you can also configure two options when you setup down:

```
> rs.stepDown(stepDownSecs, secondaryCatchUpPeriodSecs)
```

Once a primary steps down, it will not be able to become a primary again for the extent of the **`stepDownSecs`**, 60 seconds by default. The second option, **`secondaryCatchUpPeriodSecs`** has a default of 10 seconds, and will control how "caught up" any secondary needs to be, in order to be eligible to become a primary. This value is *purposely low* as any newly requested writes *will fail* during this process. If no secondary is appropriately caught up, the primary *will not* step down.

After successfully stepping down, a new election will be called and a new primary elected.

Member roles and types

There are many other configuration options for replica sets, which are out of scope for this chapter. However, make sure to check out the official documentation. Nonetheless, we will cover a few important replica set **`members`** options, including **`priority`**, **`votes`** and **`hidden`**. Using these options, you can create different kinds of member "roles" or node types.

Priority

Our basic setup had the following configuration for the members:

```
"members": [
  { "_id": 1, "host": "localhost:27021" },
  { "_id": 2, "host": "localhost:27022" },
  { "_id": 3, "host": "localhost:27023" }
]
```

This created three identical nodes, which then voted on their own and essentially picked a primary at random. If we know a node, or nodes that we would ideally like to be a primary, or a secondary that is most likely to be elected primary in the event of the current primary going down, we need to add some information for that member. Following is an example setup with weighted nodes:

```
"members": [
  { "_id": 1, "host": "localhost:27021", "priority": 1000 },
  { "_id": 2, "host": "localhost:27022", "priority": 1 },
  { "_id": 3, "host": "localhost:27023", "priority": 0 }
]
```

In this case, *node 1* has the highest **`priority`** possible (**`1000`**); *node 2* has the most basic priority of **`1`**, and *node 3* has a priority of **`0`**. This means that unless there is some issue that comes up nearly all the time, *node 1* will be elected the primary, *node 2* can become primary if needed, but *node 3*, with a priority of **`0`** cannot become a primary.

That does not mean that *node 3* is not a fully-fledged secondary, with a complete copy of the primary. Rather, it is *explicitly set* as a secondary and cannot be elected primary. If you are wondering what would happen in the case where *node 3* was the only node left online, we will discuss that in the section on replica set fail-safes.

Voting nodes

You can set the **`votes`** property of each node to either **`1`** or **`0`**; this would allow you to make a node "non-voting" by setting its **`votes`** to **`0`**. If you do so, it also needs a **`priority`** of **`0`** because nodes with a **`priority`** of more than **`0`** must be able to vote. Basically, if you are giving a node the power and responsibility of being a possible primary, it must also be able to vote in an election.

As a recap, you can have up to **`50`** members in your replica set, but only up to **`7`** can "vote". So, if you need *more than 7 members*, you will need to make sure for each of those nodes that their **`votes`** and **`priority`** are set to **`0`**.

Hidden nodes

A special type of node that has a **`priority`** of **`0`** but can vote, is known as a "hidden" node. These nodes receive a copy of the primary's data, just like any other secondary. However, they are otherwise not actively part of the replica set, with the exception of voting in an election. When configuring the members of your set, you can specify a hidden node as follows:

```
{
  "_id" : 123
  "host" : "hostname:port",
  "priority" : 0,
  "hidden" : true
}
```

Hidden nodes are ideal for things such as facilitating backups, or read only reporting tasks, since they have a copy of the primary's data. Otherwise, they have no load and can be taken offline without affecting the rest of the set.

Delayed nodes

A distinct type of **`hidden`** node is also possible, that *purposely delays* the running of operations from the primary, based on a specified delay in seconds. For example, you can set the **`secondaryDelaySecs`** option of a node to **`3600`**, and you will essentially have a "running" copy of your primary from one hour ago.

A node like this can have many purposes. Since all the operations are an hour old, you could use it to restore from some human error, such as accidental deletions. You can set up the member document of a delayed node, as we have in this example:

```
{
  "_id" : 123
  "host" : "hostname:port",
  "priority" : 0,
  "secondaryDelaySecs" : 3600,
  "hidden" : true
}
```

While they are allowed to, it is generally suggested to *not allow* delayed nodes to vote, for performance reasons.

Arbiters

As discussed earlier, a majority is required in elections. However, for various reasons, having an entire extra node, with all the data and responsibilities of a secondary just to maintain a majority, may be cost prohibitive. For these cases, there is the option of a "arbiter" node, which can vote, but *does not* have a copy of the primary's data and *cannot* become a primary.

You setup an arbiter the same way you setup a secondary. But make sure to add **`arbiterOnly: true`** to the member document, or add it to the set with a special command:

```
> rs.addArb("hostname.example.com:27017")
```

Generally speaking, arbiters are not recommended, but in some cases, they can be helpful to save costs, especially if you have your database spread across data centers. If you do use arbiters, make sure to limit to their use to one arbiter per replica set.

Administration

After your set is all up and running, a lot of the time you can just let it run, monitoring it as you would any other MongoDB server. However, there likely will be times when you need to slightly tweak your set, monitor it more closely, or recover from a problem. In this section, we will discuss making changes to your replica set, how fail-safe and disaster recovery features work, and some aspects of replica set monitoring.

Changing an existing replica set

If you want to make an adjustment to your existing replica set, you will need to "reconfigure" it and let the primary know about the changes. To do this, you can either create a new configuration object or use your existing configuration and just *adjust* it. For example, you can use the **`rs.conf()`** command to get the existing configuration object and then change the third member (at index 2) to a delayed node using **`rs.reconfig()`**, as shown in the following snippet:

```
> cfg = rs.conf()
> cfg.members[2].hidden = true
> cfg.members[2].priority = 0
> cfg.members[2].secondaryDelaySecs = 3600
> rs.reconfig(cfg)
```

Notice how instead of building up a whole new configuration object, we have put the existing configuration into a variable and modified it inline, via **`mongosh`**. In this case, *node 3* was a secondary, and so it should seamlessly reconfigure, and your set will have a new hidden, delayed node.

To reiterate, you will need to run this from the *primary* node. If *node 3* happened to be the primary, this would force it to stand down (assuming it can) and force a new election. Forcing an election will cause at least some amount of downtime, so be sure to do this during a planned maintenance window.

Fail-safes

There are a number of "fail-safe" features in place when a replica set is running, in order to make sure that there are no issues in the overall operation. One of these is that once a replica set is running, you cannot perform write or read queries on a secondary node. While not allowing writes to a secondary is probably obvious, you may be surprised about reads. For example, if we connect *directly* to one of our set's secondaries and then try to run a **`find()`** query as we have in the following snippet:

```
myReplSet [direct: secondary] test> db.example.find()
```

We will get an error similar to this:

```
MongoServerError: not primary and secondaryOk=false - consider using
db.getMongo().setReadPref() or readPreference in the connection string
```

This is first and foremost a fail-safe to make sure you know that you are not connected to the primary. By default, any queries issued to a replica set will go to the primary, unless you change the so called "read concern". As the error points out, there are a couple of ways around this fail-safe, one of which is changing the *Read Preferenc*e in your query string, by adding the **`readPreference`** parameter:

```
mongodb://localhost:27021,localhost:27022,localhost:27023/?replica-
Set=myReplSet&readPreference=primaryPreferred
```

You can also set the read concern on the command line, by running the **`setReadPref()`** from the MongoDB shell, as shown:

```
> db.getMongo().setReadPref("primaryPreferred")
```

You can get the current read concern by executing the following command:

```
> db.getMongo().getReadPrefMode()
primary
```

In older versions of MongoDB, you would run the **`rs.secondaryOk()`** or **`rs.slaveOk()`** commands to change this. In more recent versions, you can set your preference as part of your query, by appending it to the end:

```
> db.example.find().readPref("primaryPreferred")
```

Or, again, set the preference *for the duration of your connection* like so:

```
> db.getMongo().setReadPref("primaryPreferred")
```

In both these cases, we have now enabled reading from secondaries. However, as the name implies, the primary is still preferred. There are a couple different read preferences: **`primary`**, **`primaryPreferred`**, **`secondary`**, **`secondaryPreferred`** and **`nearest`**. Most of those are fairly self-explanatory, while the last one, **`nearest`**, is a bit more novel, in that it will pick the node with the *least network latency*, or how long a network request to that node it takes to come back.

The **`nearest`** read concern is useful in cases where an application's servers are spread across geographic areas. If the request comes from Edinburgh, Scotland, then a node in London or Paris will be preferred over a node in Northern Virginia, or Sydney, Australia, even if the *primary* is actually located in one of those places, as illustrated in *Figure 12.6*:

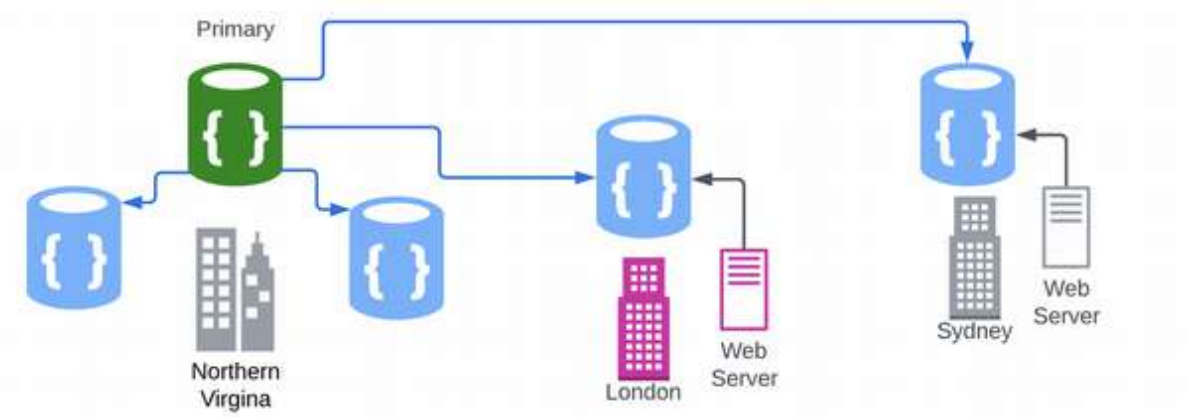


Figure 12.6: *Using nearest as the Read Preference*

Another "fail-safe" happens in the event of multiple node failures where a primary fails, or steps down, and a new primary cannot be elected. In this case, you will *no longer be able to write* to your replica set. For example, if you had three node set and two of the three nodes go down, the third *cannot be elected* primary and will remain a secondary.

However, you can still *read* from the remaining node, and so, you can keep that in mind when designing your application for handling outages. To restore *writing* to your set, you will need to get a majority of nodes back online, which we will discuss in the next section.

Maintenance and disaster recovery

Although we will cover disaster recovery in more detail in *Chapter 13, Being Proactive – Security and Backups,* we can recall that replica sets are key to better recovery, as well as smoother planned maintenance. In a replica set, by using the "step down" process, you can take a primary offline for maintenance without much or any downtime, since a secondary can take its place.

You can also use this same process to backup files on the primary, or a secondary. Furthermore, you can take advantage of concepts such as "rolling upgrades", where you take a single member of a set offline and upgrade its operating system; patch its software, or even its MongoDB version. You can then bring that node back online, test it, and move on to the next node, without as much risk as upgrading a single, non-replicated database.

However, in the case where you need to do an emergency reconfiguration of a replica set, because of some sort of failure, you can force a reconfiguration and election with an additional **`force`** option on **`rs.reconfig()`**:

```
> rs.reconfig(cfg, {force : true})
```

This `force` can allow you to, for example, force an election with a new array of members, and therefore a different majority.

Note: An important first step would be to back up a functioning node.

You can also remove nodes directly, using the following code snippet:

```
> rs.remove("hostname:27017")
```

You will need to run this command from the primary. Similarly, you can add a new member by configuring a member document, including things such as priority, voting, and so on:

```
> rs.add( { host: " hostname:27017", priority: 0 } )
```

Now you can rest a little more easily, knowing that your data is a good deal better protected!

Monitoring

There are a couple of replica set specific monitoring commands you should know about. Try them on your local replica set if you set one up:

```
> rs.status()
> rs.hello()
> db.serverStatus()['repl']
> rs.printReplicationInfo()
```

Each command will give you information and statistics about your set, with different levels of precision. You can also use these to monitor your replica set, such as running a script regularly to check and keep track of which server your primary node is running on.

Scaling with Sharding

Replication helps you spread things such as risk, across a set of servers, but when it comes to the actual data and load of the database, that remains essentially the same. A replica set does not inherently increase how much your database can handle. There are basically two ways to *scale* your database: you can scale "up" by buying or using servers with more RAM, disk and CPU, or you can scale "out" by spreading the load of your database across multiple servers. In MongoDB, you can "scale out" by using the concept of *sharding*.

Sharding data and Shard keys

Instead of getting bigger, and more expensive servers to scale your growing data, you can split up, or *shard* your data across multiple, less expensive servers. This will still use the concept of replication, but instead of copying the entire contents of your database to each node, sharding will copy just a subset of your database to each replica set.

Consider the example of a collection of recipes. We could choose to "break up" our collection by the recipes' title in alphabetical order, across replica sets, as seen in *Figure 12.7*:

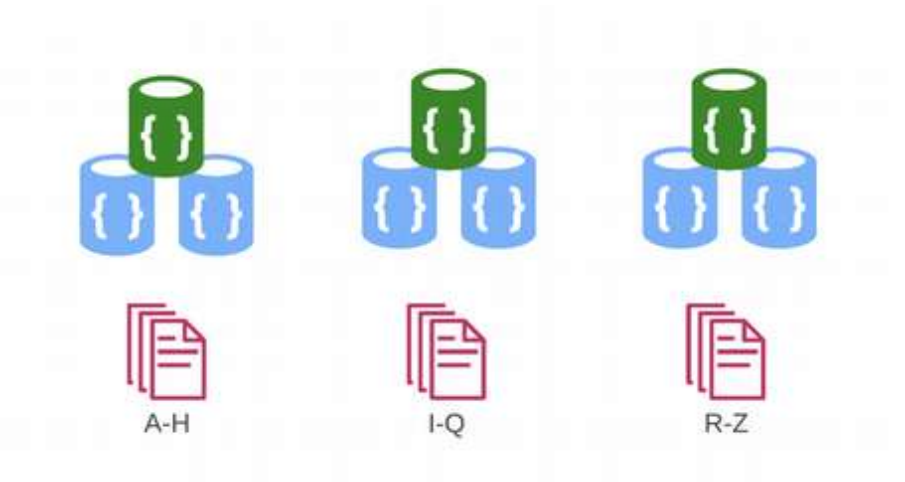


Figure 12.7: *Sharding in MongoDB*

All recipes with a title starting with an **`A-H`** will be in *replica set 1*, **`I-Q`** in *replica set 2*, and so on. MongoDB will automatically keep each replica set "balanced" by making sure no one replica set has an outsized amount of data. So, if at some point we add a lot of recipes starting with **`R`** and **`Z`**, the nodes may be "rebalanced" by moving all the **`R`** recipes from *replica set 3* to *replica set 2*, and so on.

The field (or fields) by which you instruct MongoDB to break up your data across replica sets, is referred to as the *shard key*. In this example, the shard key is the **`title`**.

Config Servers and the mongos process

Since the sharding of your data is handled automatically, your application will not know which replica set has the document you are looking for, and therefore will not know where to connect. Instead, your application will connect to a "router" process

called **mongos** which will serve as the interface to your sharded setup, as seen in *Figure 12.8*:

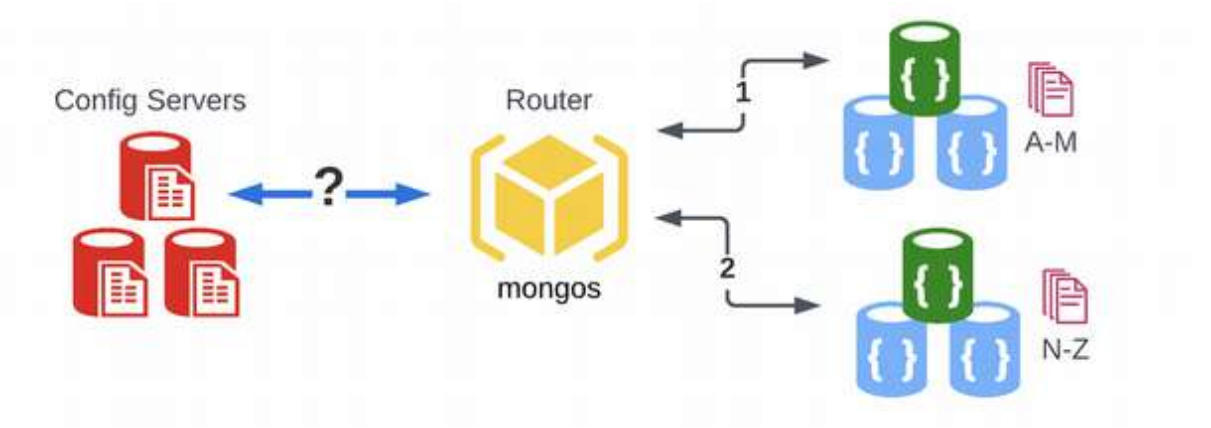


Figure 12.8: *The mongos Process and Config Servers*

As seen in *Figure 12.8,* there will be a replica set of so called *Config Servers*. This replica set maintains and stores a manifest of each shard, using the shard key, functioning much like an index. Using a cache of this manifest, any **mongos** process will know what replica set to connect to and get the data requested. The config servers are relatively light resource wise, as they only store this manifest. The **mongos** process, or processes, as there can be many, are also very minimal resource wise and are often run on the application servers themselves.

If you query uses the *shard key*, generally **mongos** can know exactly what replica set to send the query to. If the query uses a different field, **mongos** will send the query to all the replica sets and gather the results together before returning them to the application. Either way, this means your application only needs to know what **mongos** to connect to, instead of having to know anything about the sharded setup, or any of the replica sets.

Replica Set Considerations

You may be wondering if we can use sharding without replica sets. The technical answer is yes, although the correct answer is no. Why say it like that? Think about it this way: unlike a replica set that has the *full copy* of the database to ensure that if any one part goes down there is a way to pretty much instantly recover, the sharded data is split up between **mongod** instances.

So, in the case of a sharding setup without replica sets, if one of those instances goes down, *your entire sharding setup will be down* until you can restore that one instance. Not just down in the sense that you cannot write, or when a replica set does not have a primary; but down as in you cannot write or read.

If instead, you have a replica set for the **`A-H`** shard, and another replica set for **`I-Q,`** and so on, you will have the exact same resiliency as a normal un-sharded, replica set. You could opt to split this over three physical servers however, and run a replica set node for each shard on a different physical server.

If you are running a *development* setup, the replica sets are not necessary since downtime and data loss are not as big of a concern.

Sharding configuration

A sharded setup requires at least two shards to share or "distribute" data, although you could configure your application to use a sharded setup with a single replica set if you plan to shard your data later. However, due to the complexities of a sharded setup, it is not recommend to use sharding unless you know what you will need it. A replica set will work just fine for most use cases.

While the specifics of setting up a sharded server is beyond the scope of this book, much like replica sets, there are a number of helpful commands you can use to get information about

```
> sh.addShard()
> sh.shardCollection()
> sh.status()
> db.printShardingStatus()
```

If you believe you will need to know more about sharding or wish to try it out using your local replica set, go ahead! Everything is well documented on the MongoDB website. For now, you should have a good grasp of how and why you would use sharding and how a sharding setup works.

Conclusion

You should now have a solid grasp of how replication works in MongoDB, and more importantly, know why any production deployment of MongoDB should use a replica set. We discussed how a replica set will ensure that full copies of your database exist in multiple places, how to configure a replica set, how to administrate and monitor it, as well as how to use powerful scaling features such as sharding. In the next chapter, we will discuss the key concepts of security and backups.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 13

Being Proactive – Security and Backups

"If you put a key under the mat for the cops, a burglar can find it, too. Criminals are using every technology tool at their disposal to hack into people's accounts. If they know there's a key hidden somewhere, they won't stop until they find it."

— Tim Cook

"Always have a backup plan."

— Mila Kunis

Introduction

In this chapter, we will explore a couple of important aspects of databases which are often overlooked by newer users, but that are nonetheless, very important in landing and maintaining a job: security, backups and restore plans. Having a proper understanding of security and disaster preparedness will put you apart as a professional, and help you add value to any organization.

Structure

In this chapter, we will discuss the following topics:

- Authentication: Proving Who You Are

- Authorization: What You Can Do
- Backup Strategies
- Restoring Backups
- Database Encryption

Objectives

In this chapter, we will begin by discussing how to protect your database via authentication, authorization and roles. Then we will transition to how to effectively backup and restore your databases, wrapping up with a brief discussion about database encryption.

Authentication – Proving who you are

In MongoDB, *Authentication* is the process of verifying the identity of a user, or application, that is attempting to access a MongoDB database. It generally involves the use of a user name and password, or other security method, in order to authenticate the user and control access to the database.

In the past, with the exception of a service such as MongoDB Atlas, user accounts and security via authentication are not enabled for MongoDB. This is because the initial use case for MongoDB was within a "secured" server environment, much like how many key/store technologies such as **`memcache`** work.

As MongoDB became a full-fledged database product, the need for more security, as well as different levels of permissions within the database became necessary. These collectively are referred to as *authentication*, or "proving who you are", and *authorization*, or "what you can do".

A concept that may be familiar to someone that has worked in an office building, lived in a large apartment complex, or visited a hotel, is that access to a building is different from access to rooms *inside that building*.

As seen in *Figure 13.1*, an access card will grant you entrance into the building, but to get inside a room *in that building*, you will need a key. A room key can be different for each room, or you could have a "master" key to get inside any room. It is to be noted that a master key is only granted to someone with a high level of trust and responsibility. Please refer to the following figure:

Figure 13.1: *Building Access*

Authentication is like "getting into the building". You need to somehow prove who you are, and then that identity is checked against a list of allowed users. Once inside, access to individual rooms can be more granular, and to make managing this access easier, MongoDB has the concept of *Roles*. To continue our analogy, the roles might be guest, owner, managers or maintenance worker. Each role will have distinct things that they are authorized to do within the building.

Enabling access control

You can enable *Access Control* security either by using the **`--auth`** option when you start **`mongod`**, or by configuring **`security`** in your configuration file as shown here:

```
security:

    authorization: enabled
```

A reason you may want to use the **`--auth`** option is that it will force security every time, even if the configuration file changes. However, we will use the configuration option for our purposes.

Once **`authorization`** is enabled, a user account is required to access, or login to the database. You can either create this user *before* enabling authorization, or by using a one-time exception called the localhost exception.

Localhost exception

This special mode is available if you are logged into the same machine running MongoDB, the localhost, and have not yet created your first user or role. This will work for only for the first user added, so this user needs to be an admin user.

For our use case, we will use this localhost exception. Assuming that we have a default install of MongoDB, with authentication not yet enabled, we will first connect to MongoDB and shut down the server:

```
> db.adminCommand( { shutdown: 1 } )
```

Depending on your setup and how you connected to MongoDB, this may produce an error as shown:

```
MongoError: shutdown must run from localhost when running db without auth
```

This is in part due to the localhost exception, as well as a general fail safe. Make sure you connect on the same machine that MongoDB is running on, or container, if you are using Docker. After your server shuts down, you can edit your configuration file to enable authentication, which again looks like this:

```
security:

    authorization: enabled
```

Restart your **`mongod`** process, and connect on the **`localhost`**, generally by just running **`mongosh`**. If you need a refresher on configurations, starting, stopping MongoDB and so on, refer to the *MongoDB Server Operations* section of *Chapter 11*, *Make It Great – Configuration and Monitoring*.

Authorization on Docker

If you are using a Docker container, you will need to either add the **`--auth`** option right after the **`mongo`** tag like shown:

```
$ docker run -d mongo --auth
```

Alternatively, you can see the instructions on DockerHub for how to use a docker compose file, or a custom configuration file.

Creating Users

To create a user, and any additional users after that, you will use the **`db.createUser()`** command, and to create *your first user*, you will need to use the **`admin`** database as follows:

```
> use admin
> db.createUser(
  {
    "user": "myadmin",
    "pwd": passwordPrompt(),
    "roles": [
      { role: "userAdminAnyDatabase", db: "admin" },
      { role: "readWriteAnyDatabase", db: "admin" }
    ]
```

```
  }
)
```

This will create a user called **myadmin** (you can use whatever username you wish), and prompt you to enter a password via **passwordPrompt()** on the command line when you run the command. You can also type out in a plain text password, as a string in quotes, if you wish.

To create our admin user, we need to connect to the **admin** database because we want to create a master admin, over all the databases. So, first switch to the **admin** database, and then run the **db.createUser()** command. The process will look something like this:

```
> use admin
switched to db admin


admin> db.createUser(
...   {
...     "user": "myadmin",
...     "pwd": passwordPrompt(),
...     "roles": [
...       { role: "userAdminAnyDatabase", db: "admin" },
...       { role: "readWriteAnyDatabase", db: "admin" }
...     ]
...   }
... )
Enter password
****************{ ok: 1 }
admin>
```

This will create your new **myadmin** user, and that user will have both the role of **userAdminAnyDatabase** and **readWriteAnyDatabase**, which equates to full access on the **admin** database, as well as any other databases we have. We will discuss roles and privileges more in a later section.

After running this, the localhost exception will no longer be valid, so your connection is now unauthorized, and you need to **exit** and log back in with your new user.

Logging in With a User

There are two general approaches to connecting and authenticating. You can specify your credentials when you connect, or after you connect. Using **mongosh**, you can authenticate with a command as follows:

```
$ mongosh --authenticationDatabase "admin" -u "myadmin" -p
```

You will then be prompted for the password you used when you created the account. You can also supply your password in quotes after the **-p**, but this is generally not recommended as your password would be visible to anyone that has rights to view processes on your machine.

The other method is to authenticate after you connect. This can be useful if you need to change users while already connected. Simply change to the database the user is in, in this case **admin**, and issue the **db.auth()** command as follows:

```
> use admin

> db.auth("myadmin", passwordPrompt())
```

Again, you can provide a password in quotes, instead of prompting for one, but it is generally the safest to use a prompt, as your password will be obfuscated even as you type it. Once you are logged in, you can perform commands on the database, such as listing the users, for example:

```
admin> db.getUsers()
{
  users: [
    {
      _id: 'admin.myadmin',
      userId: new UUID("684c41e9-d79c-4875-adaa-260646bb6689"),
      user: 'myadmin',
      db: 'admin',
      roles: [
        { role: 'userAdminAnyDatabase', db: 'admin' },
        { role: 'readWriteAnyDatabase', db: 'admin' }
      ],
      mechanisms: [ 'SCRAM-SHA-1', 'SCRAM-SHA-256' ]
    }
  ],
```

```
  ok: 1
}
```

Since this user was created in the **`admin`** database, if you switch to a different database, you will not get that user back in the **`users`** array. You can see the exact **`roles`** the user has, as well as the **`mechanisms`** array, which we will discuss next.

Types of authentication

In our example so far, we are using MongoDB's default type of authentication, that is, **Salted Challenge Response Authentication Mechanism** (**SCRAM**), which is a modern password-based authentication protocol.

You can also use LDAP, x.509 Certificates or Kerberos Authentication, all of which are well explained in the official documentation.

If you are using MongoDB Compass, the process is fairly straightforward using the **Advanced Connection Options**, as seen in *Figure 13.2*:

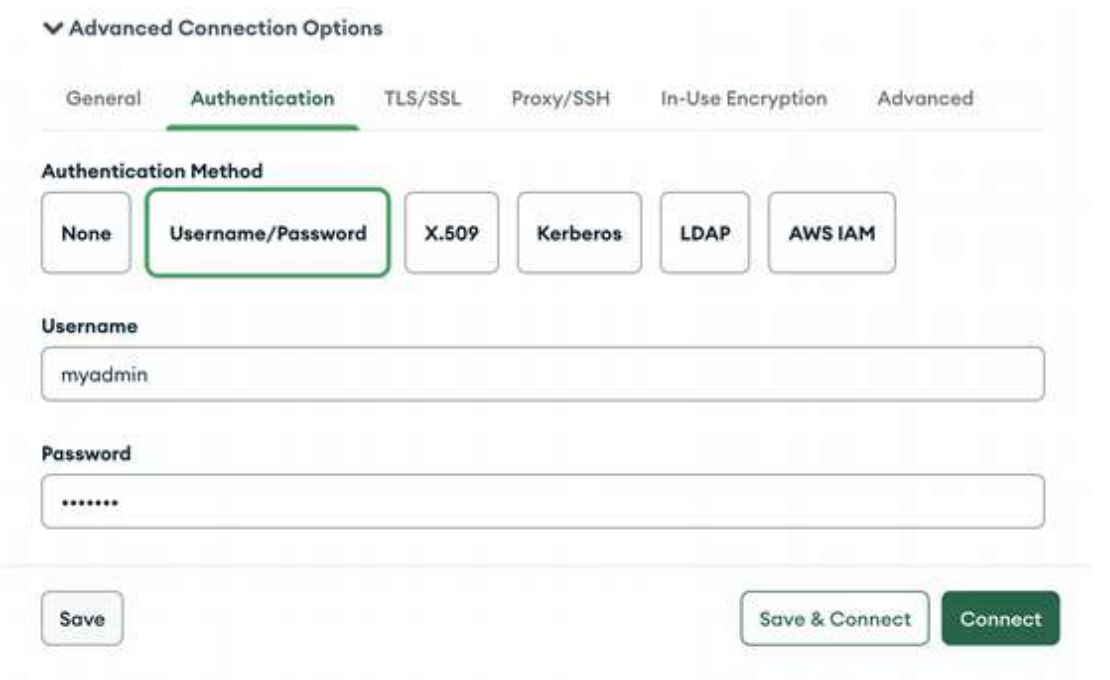


Figure 13.2: *User Login via MongoDB Compass*

Users on different databases

You can also create users within individual databases, which means that those users will need to "authenticate" against that database instead. This is useful for non-admin users, unless they are the admin of only that database, and is often used to

further scope the user's access. All the commands we have used thus far will work in any database. Just run the **`use`** command to switch to the database and run them.

To list all the users across databases, you can run the following query from the **`admin`** database:

```
> use admin
> db.system.users.find()
```

If you are using MongoDB Compass, then this collection will be hidden from you, but you can use the built-in **`>_MONGOSH`** to see this collection, as seen in *Figure 13.3*:

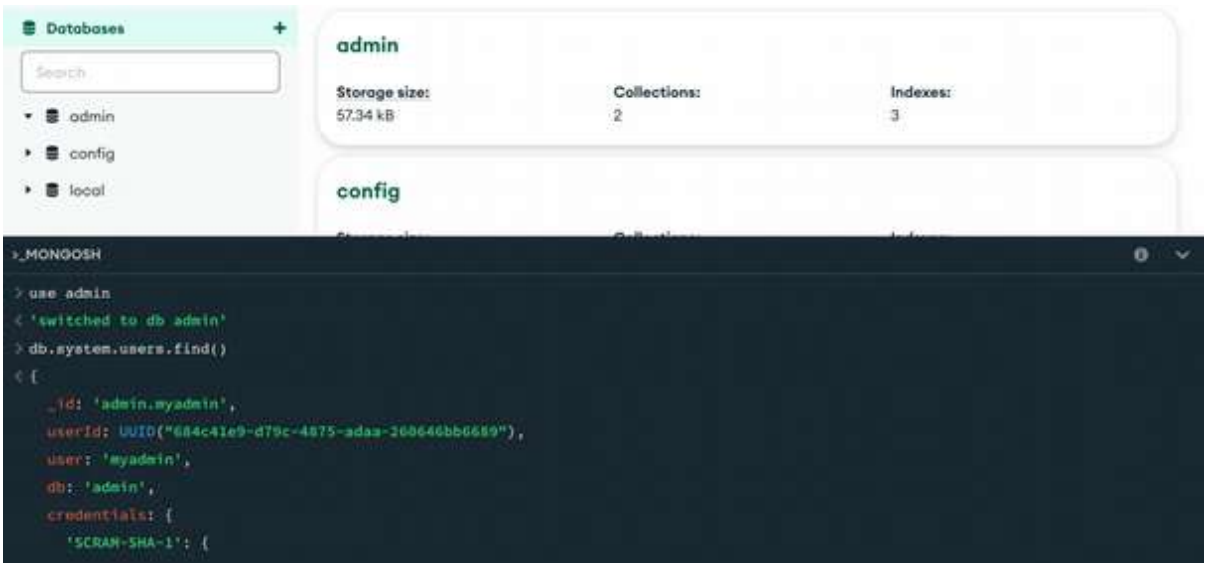


Figure 13.3: *List All Users via MongoDB Compass*

Do not attempt to manually update or add to the **`system.users`** collection. Always use the user related commands.

Authentication and replica sets

While out of scope for this book, there are additional options available for authentication between and amongst Replica Set members. Make sure to check out the documentation if you need to setup or modify how your Replica Set is using authentication.

Authorization: What you can do

Moving on in our analogy, *Authorization* is the concept of what an *Authenticated* user can do, within the database. Now that the user proved who they are, we switch to what actions they can perform.

Perhaps a user named **reporting** can only read data. Another user named **website**, that your application uses, can read some databases and write in a specific database. Or, you might have more powerful users such as the **myadmin** account that we created in the last section. Each of these different accounts will have particular rights assigned to them, also known as privileges, which we will now explore further.

Privileges

The core of *Authorization* is the granting of rights to perform actions on the database, and we call these actions privileges. Some privileges are fairly straightforward, for example, **insert** allows you to insert new documents, and the **update** and **remove** privileges allow modifying or deleting of documents.

Other privileges allow a user to carry out multiple actions. For example, the **find** privilege allows you to run the **find()** command, which is basically a "read" privilege, and also allows the user to run the **aggregate()**, **count()**, **listCollections()**, and so on, as these are all needed in common **find()** actions.

There are many other privileges such as **createIndex()** and **createCollection()**, as well as privileges for commands we have used for creating and modifying *Replica Sets*, *Sharding*, as well as shutting down the server. Additionally, there are privileges which allows an user to run various commands we have used for server administration as well, such as **collStats()** or **dbStats()**; the list goes on and on.

Roles

While the flexibility of having so many different privileges is great, knowing the combination you need for a particular user can be, at times, challenging. To make things easier, MongoDB comes with a number of pre-defined "built-in" roles, which group together permissions needed for common user types.

These built-in roles essentially build on each other, adding more and more privileges, or sometimes granting very task-based privileges. The most basic role is **read**, which can be granted per database, and give the user the **find** privilege and a few others, such as **listCollections**.

The next is **readWrite**, which provides all the privileges a user would need for general create, read, update and delete operations. You assign these roles as part of the **roles** array when you create, or update, a user:

```
> use admin
db.createUser({
  user: "website",
  pwd: passwordPrompt(),
```

```
  roles: [
    { role: "readWrite", db: "cookbook" },
    { role: "read", db: "archive" }
  ]
});
```

Note the **role**, and the target **db**. In this case, we created a user named **website** that has access to read and write in the **cookbook** database, and only to **read** in the **archive** database. This might be useful if we want the **website** user to be able to run code that reads and writes recipes in the **cookbook** database, but only to be able to read recipes out of our **archive** database, since we want to treat that as "read only" for that user's purposes.

There are also more powerful roles such as **dbAdmin**, which grants privileges related to database management, or **dbOwner** which has all those same privileges plus the ability to create and modify users on that database. There are also roles specifically around clusters, as well separate **backup** and **restore** roles, which have privileges required for those tasks.

You can also grant roles to an existing user, without otherwise modifying that user, for example, a user named **chefAdmin** by using the following command:

```
> db.grantRolesToUser(
  "chefAdmin",
  [ { role: "dbAdmin", db: "cookbook" } ]
)
```

Lastly, as we have already seen, there are so called "all-database" roles such as **readWriteAnyDatabase** and **readAnyDatabase** which are useful if you are not concerned about a user reading from any database you currently have, or will in the future, but do not want to grant any higher privileges than simply reading.

User-defined roles

If none of these built-in roles handle the specific combinations of privileges you need, then you can also create your own roles, as seen in this example:

```
> db.createRole({
  role: "findAndKill",
  privileges: [
    {
```

```
      resource: { db: "", collection: "" },
      actions: [ "find", "killop", "killCursors" ]
    }
  ],
  roles: []
})
```

This new **`findAndKill`** user-defined role will allow the user to perform both **`find`** and **`kill`** actions. This is simply an example however, and your custom role probably would make more practical sense.

We did not specify a **`db`** or **`collection`** here, and so, this will apply to all databases and collections. However, you can set either one of those to scope your role. Additionally, there is an optional **`roles`** array; any roles you specify in this array, and that role's associated privileges will be *combined* with your new role.

You can view your custom roles by using the **`db.getRoles()`** command:

```
> db.getRoles()
{
  roles: [
    {
      _id: 'admin.findAndKill',
      role: 'findAndKill',
      db: 'admin',
      roles: [],
      isBuiltin: false,
      inheritedRoles: []
    }
  ],
  ok: 1
}
```

Creating the right mix of privileges for different users in your database is a key way to protect your data and reduce risk. The other pillar of protection and reduced risk is having a proper backup and restore plan for your databases, which we will cover next.

Backup strategies

Even with excellent replication technologies, and the best server setup, there is no replacement for backups. The failure to have a good backup strategy can be disastrous. Data loss due to hardware failures, cyber-attacks or natural disasters are almost always out of your control. Additionally, many countries and governments have various compliance standards which require backup and restore plans.

Fortunately, MongoDB offers a number of ways to back up your database, some of which we discussed in *Chapter 10*, *Getting In and Getting Out – Data Migration*, such as **`mongodump`** and **`mongoexport`**. In this section, we will discuss some additional methods for backing up your MongoDB databases.

Filesystem Backups

One of the simplest and most straightforward ways to backup MongoDB is to make a copy of the MongoDB database data files themselves. While this method might not be best for more complex setups, it works in almost all cases. The key to this method is that you need to stop any writes to the database either manually, such as by using the **`db.fsyncLock()`** command, or by shutting down the **`mongod`** process for the duration of the backup. This is to ensure that no data changes while you copy the files, which would create an invalid backup.

Note: If you are running a replica set, which you should be, you can take a single node offline to accomplish this process without causing any downtime for your overall database or application.

Once all writes are stopped, you can copy the data folder to another location on disk. As discussed earlier, the data folder is set in the configuration file as the **`dbPath`**; you do not need to copy the log files. On Linux, this process might look like this:

```
$ cp /data/db /backups/mongodb/
```

This will copy all your MongoDB data files to a backup location somewhere else on your system. In a real-life situation, you will need to make sure the directory you are copying to, already exists and probably will want to name it in a such a way that it is clear when this backup happened.

You can also use a program like **`rsync`**, if you wish:

```
rsync -avz --progress --exclude="*.lock" /data/db/ backup-folder/
```

In this example, the **`-a`** option is used to preserve file attributes, the **`-v`** option is used to enable verbose output, the **`-z`** option is used to compress the data during transfer, the **`--progress`** option is used to display a progress bar during the transfer, and the **`--exclude`** option is used to exclude **`*.lock`** files from the backup.

On Windows, you can log into your server, and copy and paste the folder to a backup location. After the files are successfully copied, you can use a combination of **`tar`** and **`gzip`** or **`zip`** to turn your back up into a single file. See the example backup script later in this chapter for an example.

You can also use filesystem "snapshots" if your operating system and setup supports them, for example, if you are using Amazon's EBS storage system for EC2 or the Logical Volume Manager also known as LVM on Linux. This allows you to take a "snapshot" of the data at a point in time which you can use as a backup. To use this method, you must have the journaling option enabled, which as we discussed earlier, is on by default.

MongoDB Database Tools

We already reviewed using *MongoDB's Database Tools* to back up your databases in *Chapter 10, Make It Great – Configuration and Monitoring*, so we will not rehash that, but we will point out a few things to keep in mind.

If you use the **`mongodump`** tool, it will create an optimized "binary" backup of your databases and put these backups in a folder structure matching your database. This is both more efficient and needs less storage space than copying the data files manually, as **`mongodump`** will deduplicate and cleanup things such as indexes.

As a reminder, if you have enabled *Access Control*, you will need to make sure the user you use to connect to the database has the correct permissions.

Example backup script

You will most likely want to encapsulate the logic to perform a backup in script that is run on a schedule. You can run that script, nightly, for example, with a program like **`cron`**, or using a more complex solution like Jenkins (which is an automation server available at **https://www.jenkins.io/** for free.

The following is an example script written in BASH, to back up a database, in the example **`test`** database, using **`mongodump`**:

```
#!/bin/bash


# Set the MongoDB host and port
host=localhost
port=27017


# Set the name of the database to be backed up
```

```
db_name=test

# Set the location where the backup will be saved
backup_location=/store/backups/

# Get the current date and time to be used as the backup file name
now=$(date +"%Y-%m-%d-%H-%M-%S")

# Create the backup directory if it doesn't exist
if [ ! -d "$backup_location" ]; then
  mkdir -p "$backup_location"
fi

# Run the mongodump command
mongodump --host $host --port $port --db $db_name --out $backup_location$now

# Compress the backup into a tar.gz file
tar -zcvf $backup_location$now.tar.gz $backup_location$now

# Remove the unzipped backup
rm -rf $backup_location$now

# Print a success message
echo "Backup complete: $backup_location$now"
```

If you want to run this script nightly with **`cron`**, you setup something like the following process, or your system's equivalent. Start by opening the **`cron`** editor:

```
$ crontab -e
```

Then add a line, which will run the backup script every night at midnight:

```
0 0 * * * /path/to/backup.sh
```

Now you have a simple, but effective, backup plan in place! You can find a copy of this **`backup.sh`** script, as well as examples in Python and PowerShell (for Windows) in the **`chapters/13`** folder of the book's git repo.

MongoDB services

There are also services provided by MongoDB as part of their Atlas Cloud, including *Cloud Backups* which has continuous backups or on-demand snapshots. *MongoDB Cloud Manager* is also available as well as *Ops Manager*.

While not free, these services can greatly simplify your backup and restoring workflows so it is something you will want to investigate depending on your needs.

Restoring backups

Having a plan to restore your backups, and testing that plan is crucial to having a resilient database setup. We will discuss two basic ways to restore your databases: by using raw data files or the *MongoDB Database Tools*.

Restoring via MongoDB data files

If you have copied the whole data files folder, you can simply "restore" by either pointing the **`mongod`** process, via the config file, to the location of the backedup folder. Alternately, you can move the contents of the backup to your server's **`dbPath`**. Make sure not to mix the backup files with any existing files in the **`dbPath`**, only use the files in the backedup folder.

MongoDB Database tools

A more powerful way to restore is by using the previously discussed **`mongorestore`** program, that comes as part of the *MongoDB Database Tools*. Assuming a default server setup running on a local host, you can run **`mongorestore`** with a dump folder named **`backup_dump`** as follows:

```
$ mongorestore backup_dump/
```

You can also restore to a MongoDB instance not running on your localhost, with a backup folder on your local machine using the **`--host`** option:

```
$ mongorestore --host myserver.example.com backup_dump/
```

By default, this will restore all the databases that were backed up in your dump. If you want to specify or replace a database, like in this case **`cookbook`**, you can use the **`--db`** and/or **`--drop`** options:

```
$ mongorestore --db cookbook backup_dump/cookbook/
```

If you need to restore to a different database name, use a command as follows:

```
$ mongorestore --db newCookbook --drop dump/cookbook/
```

Again, if using *Access Control*, you will need to make sure to include any user name, password and database to authenticate against:

```
$ mongorestore --username admin --password --authenticationDatabase admin
backup_dump/
```

Example Restore script

The following is an example of a script, written in BASH, to restore a database using **`mongorestore`**:

```
# Set the MongoDB host and port
host=localhost
port=27017

# Set the location of the backup file
backup_file="/store/backups/test-2022-10-01-10-00-00.tar.gz"

# Set the location where the backup will be restored
restore_location="/tmp/restore"


# Extract the backup file
tar -xzvf $backup_file -C $restore_location

# Get the name of the backup directory
backup_dir=`ls $restore_location`

# Create the mongorestore command
mongorestore_command="mongorestore --host $host --port $port --drop
$restore_location/$backup_dir"

# Run the mongorestore command
$mongorestore_command

# Print a success message
echo "Restore complete"
```

You can find a copy of this **`restore.sh`** script, as well as other examples in Python and PowerShell (for Windows), in the book's git repository.

Make sure to test these backups and restore processes regularly, sort of like a "fire drill". For example, take a node from your replica set offline and try restoring a backup to that node. This will not only help you feel confident in how to run a restoration, but it will also reveal any issues in your restore plan before a real emergency happens.

Database encryption

There are various methods of encryption you can use in MongoDB, much of which is out of scope for this book. However, to make sure you have a good idea of the options out there, we will briefly touch a couple of main areas. It should be noted that many of the encryption options are only available with the *Enterprise* addition of MongoDB, which is not the same as the free *Community Edition* that we have been working with thus far.

Network encryption

MongoDB supports network-based encryption between client and servers, as well as between members of a replica set, using **Transport Layer Security** (**TLS**) or **Secure Sockets Layer** (**SSL**) security. When using TLS, all the data transmitted between MongoDB instances is encrypted and thus protected from eavesdropping or tampering.

To use TLS in MongoDB, you need to generate TLS/SSL certificates and configure the server to use them. Once the certificates are in place, you can enable TLS encryption for incoming client connections, outgoing connections from the server to other servers, or both via the configuration file:

```
net:
   tls:
      mode: requireTLS
      certificateKeyFile: /etc/ssl/mongodb.pem
```

Application-level encryption

Another type of encryption you can leverage in MongoDB is called application-level encryption, which is the process of encrypting data at the application layer before it is written to the database. This allows the database to store encrypted data and the application to handle the decryption. This method provides a high level of security, as the encryption key is never stored in the database.

Additionally, there are two new types of encryption we will briefly consider which, at the time of writing, are not "general release" but that you should be aware of as

they will become fully available in the future: **Client-Side Field Level Encryption** (**CSFLE**) and Queryable Encryption.

Client-side Field Level Encryption (**CSFLE**) takes the concept of encryption a bit further down to the field level, whereas Queryable Encryption allows you to query data in your database, without the server having knowledge of the data it is actually processing. Both of these are only currently available with the *Enterprise* version of MongoDB or MongoDB Atlas.

You can find out more here: **https://www.mongodb.com/docs/manual/security/**

Conclusion

While not as fascinating as some of the topics in this book, this chapter encapsulates some of the most important features any professional should know, and internalize, when working with databases. We have learned how to secure your databases with authentication, how to grant privileges properly with authorization, as well as how to leverage helpful built-in roles. Building on knowledge we gained earlier, we have learned how to create backup and restore plans, as well as learning how to script those plans. With this more secure, solid base, we can confidently progress into the next chapter where we will start to make stuff, by programming with MongoDB!

Chapter 14

Making Stuff – Programming with MongoDB

"Everybody should learn to program a computer because it teaches you how to think."

— Steve Jobs

"Most good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program."

— Linus Torvalds

Introduction

One of the first things you might be excited to use your MongoDB knowledge for, is some code! Understanding how to program with MongoDB is a useful skill even if you are not, or do not plan to be, a full time programmer. If you are a programmer or wish to land a job as one, learning at least the basics of programming with MongoDB is a must. In this chapter, we will focus on a couple of core programming languages and how they work with MongoDB, but make sure to check out the end of the chapter for more examples in various languages.

Structure

In this chapter, we will discuss the following topics:

- Programming with MongoDB

- Python and MongoDB
- Node.JS and MongoDB
- PHP and MongoDB
- Other Language Examples

Objectives

By the end of this chapter, you should have a solid idea of how to connect and perform basic queries against MongoDB, using code written in Python, Node.JS (JavaScript) and PHP. Make sure to see the section about the book's free GitHub Codespace, where you can try some of this code out for yourself, without installing anything locally.

Programming with MongoDB

The following examples are meant to be a stepping off point for programming with MongoDB. They are just touching on the powerful things you can do with a programming language and the various frameworks that work with MongoDB. Indeed, they represent only a small sampling of the available programming languages and drivers for MongoDB, although we will be touching upon the most popular languages used by developers every day.

Code examples

For each language featured in this chapter, mainly Python, Node.JS and PHP, you will be provided with similar examples of connecting to MongoDB, as well as finding and inserting documents. If you are only interested in a single language, feel free to skip directly to that section. You will also learn the variances in how each of these languages will handle all operations differently.

Book GitHub Codespace

Along with all the code examples being available in the course's git repo, if you happen to have a GitHub account, you can use this book's built-in Codespace to try them all out in your browser. There is no need to install anything on your local machine.

Codespace allows you to run a version of Visual Studio Code, and a virtual machine in the cloud, to run and even edit these code examples. You can launch the Codespace, if you have a GitHub account, via the **Code** | **Codespaces** button from within GitHub, as seen in *Figure 14.1*:

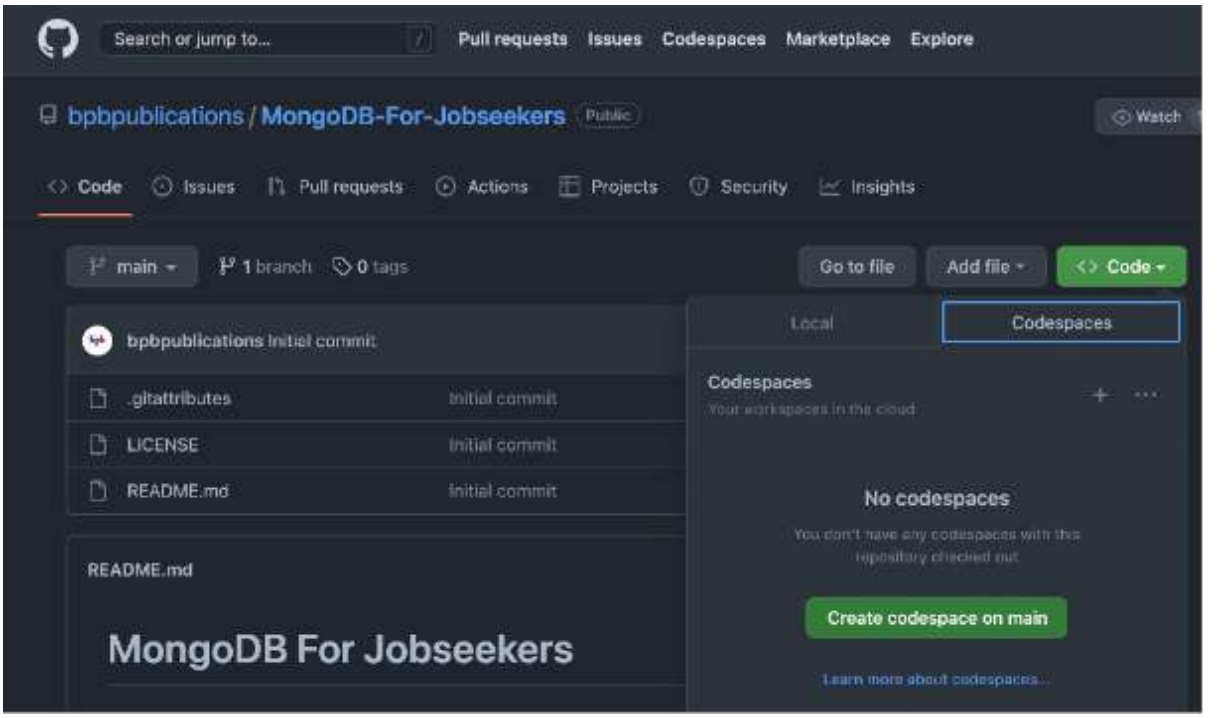


Figure 14.1: *Book Codespace*

When you launch the Codespace, you will be able to view and edit all files and folders in the book's git repo, as well as run the supported code examples against a MongoDB, with pre-loaded data. This is optional, but make sure to check it out if that interests you.

Python and MongoDB

You can find more about Python and MongoDB by going to MongoDB's official Python site: **https://www.mongodb.com/languages/python**. For the purposes of this chapter, we will demonstrate connecting to MongoDB and performing some basic queries using Python.

Installing the PyMongo library

The main prerequisite for programming with Python and MongoDB is installing *PyMongo*, which we will do via the **`pip`** command:

```
$ python -m pip install "pymongo[srv]"
```

You can also install it globally if you wish. See more in the **`pip`** documentation. After that is installed, we can import it into our Python programs and start working with MongoDB.

Connecting to a Database with Python

The following Python code will connect to our MongoDB database and perform a **findOne()**, to get back a single document from our collection. If you are following along on your own, make sure to change the connection string:

```
1   import pymongo
2
3   # Replace with your MongoDB connection string and database name
4   connection_string = "mongodb://<username>:<password>@<host>"
5
6  database_name = "website"
7
8  # Connect to the MongoDB
9  client = pymongo.MongoClient(connection_string)
10
11 # Set the database
12 database = client[database_name]
13
14 # Example query to test the connection
15 collection = database["cookbook"]
16
17 doc = collection.find_one()
18
19  print(doc)
```

On line 1, we have imported **pymongo**, and then set our connection string on line 4 along with our database name on line 6. We setup a client using that connection string on line 9, and set our database on line 12 along with the collection name on line 15.

Using this client, we then perform a **find_one()** which runs **findOne()** on the database and assign the resulting document to a variable called **doc** on line 17. This variable will be an object, which we print out on line 19. Now that we know how to connect and perform a query, let us try out some more complicated queries.

Performing queries with python

Next, we will refactor our code slightly, and run a **`find()`**:

```
1   import pymongo
2
3   client = pymongo.MongoClient("localhost", 27017)
4
5   # Set our database to "website"
6   db = client.website
7
8   for item in db.cookbook.find():
9       print(item["title"])
```

Since **`find()`** returns a cursor, we will need to iterate over it, which we do on line 8, and assigning each document, we get back to the **`item`** variable. We can then print out just the **`title`** field of the document by using array syntax on line 9, **`item["title"]`**, and so on for any other field we need.

Query options with python

Sorting and limiting in Python works much like **`find()`** on the MongoDB Shell, with some slight differences. For example, to sort results by **`title`** in descending order while limiting to three results, you can change your code slightly, as follows:

```
1   import pymongo
2
3   client = pymongo.MongoClient("localhost", 27017)
4
5   # Set our database to "website"
6   db = client.website
7
8   for item in db.cookbook.find().sort("title", pymongo.ASCENDING).
    limit(3):
9       print(item["title"])
```

Notice that instead of **`1`** or **`-1`**, we have used the **`pymongo.ASCENDING`** enum, and aside from that, the query looks very familiar.

Inserting a document with Python

Creating a new document with Python in MongoDB is quite straightforward. In this example, we will create a new recipe for waffles:

```
1   import pymongo
2
3   client = pymongo.MongoClient('mongodb://localhost:27017/')
4
5   db = client['website']
6   collection = db['cookbook']
7
8    # Create document
9    document = {
10    'title': 'Waffles',
11    'directions': [
12      'Mix batter',
13      'Cook on waffle iron until golden brown',
14      'Put lots of butter and maple syrup'
15    ],
16  }
17
18  # Insert the document
19  result = collection.insert_one(document)
20
21  document_id = result.inserted_id
22
23  # Output the _id of the inserted document
24   print('Inserted successfully! The _id is: ', document_id)
```

Here, we composed a document as an object, in this case, a variable named **`document`** on line 9-16. Notice the object has an embedded array. Next, on line 19, we insert it into the collection and assign the result to a variable named **`result`**.

We can then use **`result`** on line 21 to get the **`inserted_id`** and so we can show it on line **`24`**, thus returning the **`_id`**, back to the user, or your application, and so on.

If you save this as a file named **`insert.py`** and run this program, you should get output as follows:

```
$ python3 insert.py
Inserted successfully! The _id is: 63f01c0449fe811c5fc2baca
```

Putting this all together we can combine **`insert()`** and **`findOne()`** as follows:

```
 1   import pymongo
 2
 3   client = pymongo.MongoClient('mongodb://localhost:27017/')
 4
 5   db = client['website']
 6   collection = db['cookbook']
 7
 8    document = {
 9     'title': 'Waffles',
10     'directions': [
11       'Mix batter',
12       'Cook on waffle iron until golden brown',
13       'Put lots of butter and maple syrup'
14     ],
15   }
16
17   result = collection.insert_one(document)
18   document_id = result.inserted_id
19
20    # Get back the inserted document
21    new_document = collection.find_one({'_id': document_id})
22
23    document_title = new_document['title']
24
25    # Print the title of the inserted document
26    print('Successfully inserted: ', document_title)
```

Here we have again inserted a document and got back the new **_id** on lines 17-18, which we then use on line 21 to run a query to get the newly inserted document back via its **_id**.

Lastly, we use that document to display the **title** to the user on line 26.

Aggregation with Python

Using the *Aggregation Framework* from Python code is also pretty painless. You can, for the most part, copy and paste a pipeline query directly into your code as we have in this example:

```
 1   import pymongo
 2
 3   client = pymongo.MongoClient("localhost", 27017)
 4
 5   # set our database to "cooker"
 6   db = client.cooker
 7   collection = db["recipes"]
 8
 9   pipeline = [
10       {
11           "$group": {
12               "_id": "$type",
13               "count": {"$sum": 1}
14           }
15       },
16       {
17           "$sort": {"count": -1}
18       }
19   ]
20
21   result = list(collection.aggregate(pipeline))
22
23   for doc in result:
24       print(doc["_id"], "has", doc["count"], "recipe(s)")
```

Notice that here we start off with defining an *Aggregation Framework* pipeline on lines 9-19 which contains two steps, a **`$group`** by a **`type`** field and then a **`$sort`** in descending order on the resulting **`count`**.

We send this pipeline to **`aggregate()`** on line 21 and wrap that in a **`list()`**, assigning the resulting cursor to a **`result`** variable. Using **`result`**, we can iterate over the cursor and display output of our aggregation. If you run this script, you should get output resembling the following:

```
$ python3 aggregation.py


Dinner has 4 recipe(s)
Dessert has 2 recipe(s)
Breakfast has 1 recipe(s)
```

There is a lot more to learn about Python and MongoDB, so if that is your language of choice, make sure to look up more on the MongoDB website.

Node.JS and MongoDB

There is wide spread support for MongoDB within the Node.JS universe, and it is quite easy to setup and get coding.

Installing the MongoDB Driver

The MongoDB Node.JS driver is installed much the same as any other package for Node, with the easiest ways being using either **`npm`** or **`yarn`**. Here, we will specify the **`5.0`** version of the driver (this is not MongoDB the version, but rather the driver version):

```
$ npm install mongodb@5.0
```

Or, again with yarn:

```
$ yarn add mongodb@5.0
```

This will download the driver to your **`node_modules`** folder and add it to your **`package.json`** file. You should now be ready to code with MongoDB.

Connecting to a Database with Node.JS

In the following example, we will connect to a MongoDB database:

```
1    const MongoClient = require('mongodb').MongoClient;
```

```
2
3   // Replace with your MongoDB connection string and database name
4    const uri = 'mongodb://<username>:<password>@<host>/website';
5
6   MongoClient.connect(uri, (err, client) => {
7     if (err) {
8       console.log(err.stack);
9       return;
10    }
11
12    console.log('Connected Successfully!');
13
14    client.close();
15  });
```

First, we establish a **`MongoClient`** on line 1; you could also use object destructing:

```
const { MongoClient } = require("mongodb");
```

Or alternatively, an import:

```
import { MongoClient } from 'mongodb'
```

We then use this client to connect to the database on line 6, here called **`website`**, handle any errors, and disconnect.

You can also organize this code differently. Here, we are connecting inside an **`async`** function and running a **`findOne()`** query:

```
1    const MongoClient = require('mongodb').MongoClient;
2
3    const uri = 'mongodb://<username>:<password>@<host>';
4    const db = 'website';
5
6   (async function() {
7       const client = new MongoClient(uri, { useUnifiedTopolo-
    gy: true});
8
```

```
 9      try {
10        await client.connect();
11
12        const database = client.db(db);
13
14        const collection = database.collection('cookbook');
15
16        const doc = await collection.findOne();
17
18        console.log(doc);
19
20      } catch (err) {
21        console.log(err.stack);
22      }
23
24      await client.close();
25    })();
```

Notice that we created a **`MongoClient`** in the same way, but this time inside an anonymous **`async`** function. We have assigned our connection to a variable named **`client`** and then inside a **`try/catch`**, we are connecting to the database and collection on lines 10-14.

After connecting, we run a **`findOne()`** on line 16 to get back a single document, which we assign to a variable called **`doc`**.

Lastly, we print out **`doc`** via a **`console.log`** and handle any errors in the **`catch`** on line 20, following up by closing our connection, on line 24.

Query Options with Node.JS

You can easily add query options in Python by chaining them onto **`find()`**, just like you would in the MongoDB Shell:

```
1     const MongoClient = require('mongodb').MongoClient;
2
3     const uri = 'mongodb://localhost:27017';
```

```
4    const db = 'website';
5
6   (async function() {
7     const client = new MongoClient(uri, { useUnifiedTopolo-
    gy: true});
8
9     try {
10      await client.connect();
11
12      const database = client.db(db);
13
14      const collection = database.collection('cookbook');
15
16      const documents = collection.find().sort(['title', 1]).lim-
    it(3);
17
18      // Iterate over the cursor
19      while(await documents.hasNext()) {
20        const document = await documents.next();
21        console.dir(recipe.title);
22      }
23
24    } catch (err) {
25      console.log(err.stack);
26    }
27
28    await client.close();
29  })();
```

In this code sample, on line 16 we have added a sort, by **`title`**, and a limit, to a **`find()`** query. Since this is a **`find()`** and not a **`findOne()`**, we will get back a cursor, which we need to iterate over on lines 19-22 using a **`while()`** loop.

We await the **`hasNext()`** method on the **`documents`** cursor, as the parameter for the **`while()`** loop. This allows us to check if there are any documents left in that cursor, and then assign any next document on line 20 and output it on line 21. All this is done in a try/catch so that we can handle any errors on lines 24-26 and then close our connection on line 28.

Inserting a Document with Node.JS

Adding a new document works in the same way, with some changes, as seen in the next example.

```
1   const MongoClient = require('mongodb').MongoClient;
2
3   const uri = 'mongodb://localhost:27017';
4   const db = 'website';
5
6   const document = {
7    'title': 'Waffles',
8    'directions': [
9      'Mix batter',
10     'Cook on waffle iron until golden brown',
11     'Put lots of butter and maple syrup'
12   ]
13 };
14
15 (async function () {
16   const client = new MongoClient(uri);
17
18   try {
19     await client.connect();
20
21     const database = client.db(db);
22     const collection = database.collection('cookbook');
23
24     const result = await collection.insertOne(document);
```

```
25
26        const id = result.insertedId;
27
28        // Get back the inserted document
29        const insertedDocument = await collection.findOne({ _id: id });
30
31        const documentTitle = insertedDocument.title
32
33        console.log(`Successfully inserted: ${documentTitle}`);
34
35      } catch (err) {
36        console.log(err.stack);
37      }
38
39      await client.close();
40    })();
```

This time, we created an object variable named **`document`** on lines 6-13 that represents the document we want to insert, and on line 24, we use **`insertOne()`** to insert it into our collection, assigning the response to a variable named **`result`**.

Using the **`result.insertedId`**, we can get the **`_id`** of the newly inserted document and then on line 29, do a **`findOne()`** to get the whole new document back, assigned to a variable called **`insertedDocument`**. Using this variable, we can get the **`title`** to display on line 33.

If you save this in a file named **`insert.js`** and run this code, you should get the following result:

```
$ node insert.js

Successfully inserted: Waffles
```

Aggregation with Node.JS

Using the *Aggregation Framework* with Node.JS is very straightforward; you can paste in a pipeline query directly into Node.JS, as shown:

```
1   const MongoClient = require('mongodb').MongoClient;
```

```
2
3     const uri = 'mongodb://localhost:27017';
4     const db = 'website';
5
6   (async function () {
7     const client = new MongoClient(uri, { useUnifiedTopolo-
    gy: true });
8
9     try {
10      await client.connect();
11      const database = client.db(db);
12      const collection = database.collection('cookbook');
13
14      const pipeline = [
15        { '$group': { '_id': "$type", 'count': { '$sum': 1 } } },
16        { '$sort': { 'count': -1 } }
17      ];
18
19      const cursor = collection.aggregate(pipeline);
20
21      // iterate over the cursor
22      while (await cursor.hasNext()) {
23        const doc = await cursor.next();
24        console.log(`${doc._id} has ${doc.count} recipe(s)`);
25      }
26
27    } catch (err) {
28      console.log(err.stack);
29    }
30
31    await client.close();
32  })();
```

On lines 14-16, we have created a two-step pipeline that runs a **`$group`** on the **`type`** field on line 15, along with a count of how many documents have that **`type`**.

Next, we **`$sort`** descending, on line 16 on that **`count`**.

This pipeline array gets sent to **`aggregate()`** on line 19, and then we iterate over the results on lines 22-25 in much the same way as we did with **`find()`**.

If you run this script, depending on the document you have in your collection, you should get the following output:

```
$ node aggregation.js
Dinner has 4 recipe(s)
Dessert has 2 recipe(s)
Breakfast has 1 recipe(s)
```

Using MongoDB with Mongoose

A popular framework to use with MongoDB and Node is Mongoose. The following is a very basic example of using Mongoose with MongoDB. For more information, see **https://mongoosejs.com/**

First, setup your connection:

```
1   const mongoose = require('mongoose');
2
3    // Set up a MongoDB connection
4   mongoose.connect('mongodb://localhost:27017/your_database_
    name', {
5
6     useNewUrlParser: true,
7     useUnifiedTopology: true,
8     authSource: 'admin',
9     user: 'your_username',
10    pass: 'your_password'
11
12  }).then(() => {
13
14    console.log('Connected to MongoDB server');
```

```
15
16  }).catch((err) => {
17
18    console.error('Error connecting to MongoDB server', err);
19
20  });
```

Then you setup a Model in this fashion:

```
1   const mongoose = require('mongoose');
2
3   const CookbookSchema = new mongoose.Schema({
4     // Define your schema fields here
5   });
6
7   const Cookbook = mongoose.model('Cookbook', CookbookSche-
    ma, 'cookbook');
8
9   module.exports = Cookbook;
```

Lastly, in your Controller, use the Model as shown:

```
1   const Cookbook = require('./models/Cookbook');
2
3   Cookbook.findOne({_id: 'your_document_id'}).then((document) => {
4
5     // Print the result
6     console.log(document);
7
8   }).catch((err) => {
9
10    console.error('Error finding document', err);
11
12  });
```

There is lots more to learn about Node.JS and MongoDB, as well as frameworks that use it. So make sure to get out the official documentation for more.

PHP and MongoDB

PHP has continued to be one of the most used languages for programming the web, and it has extensive support for MongoDB. The latest instructions, and a lot more information about PHP and MongoDB can be found on the official website:

https://www.mongodb.com/docs/drivers/php/

In this section, we will examine examples of connecting to MongoDB via PHP, as well running some basic queries.

Installing the MongoDB Driver and Library

The MongoDB driver for PHP is technically broken into two parts: the extension and the MongoDB PHP library. The basic steps for installing MongoDB support for PHP are first, installing the extension via **`pecl`**:

```
$ sudo pecl install mongodb
```

Make sure the new extension is loaded in your **`php.ini`** file:

```
extension=mongodb.so
```

Next, you will want to install the library, which is recommended to do via **`composer`**:

```
$ composer require mongodb/mongodb
```

For other install methods, or more about how to use **`composer`**, see the documentation.

Autoloading the PHP MongoDB library

In the following examples, we will not include formatting such as HTML, for brevity, but it is important to make sure to include the autoloader file, here **`vendor/autoload.php`**, in some fashion. For clarity, we have included it as a **`require`** at the top of each file, but in your production code, you will likely have included it globally.

```
1    <?php
2
3    require 'vendor/autoload.php';
```

Connecting to a database with PHP

In the following example, we will connect to a MongoDB Database, select a collection and output the documents. If you are trying this code out on your own, make sure to change the connection string.

```
<?php

require 'vendor/autoload.php';

// Replace with your MongoDB connection string and database name
$host = "mongodb://<host>";

$dbname = "website";

$options = array(
    "username" => "myusername",
    "password" => "mypassword"
);

// Connect to MongoDB
$client = new MongoDB\Client($host, $options);

// Select our database
$database = $client->selectDatabase($dbname);

// Query our collection
$collection = $database->cookbook;
$docs = $collection->find();

// This will return a cursor, so we need to iterate on each document
foreach ($docs as $document) {
    var_dump($document);
}
```

Here, we can see in lines 6-13, we setup our connection information including the host, and database, here called **website**, and any user name and password needed. Next, we setup a MongoDB Client on line 16, which we can use to switch to our database on line 19.

Using our **$database**, we can switch to our collection, **cookbook** and perform a simple **find()** on line 23.

As mentioned before, this will return a cursor, so we will need to iterate over that cursor on line 26 with **findeach()** to get back each document. The **var_dump()** will output the MongoDB document as a PHP object.

Using this knowledge, we can start writing some more complex code.

Query Options with PHP

Unlike some other languages which use the same "chaining" concept to add query options like sort and limit to a query, PHP uses the second parameter of PHP's **find()** method to send both the **projection** and options such as **sort** and **limit**.

In the following example, we will use this **options** parameter:

```
1   <?php
2
3   require 'vendor/autoload.php';
4
5   echo "<pre>";
6
7  $client = new MongoDB\Client(
8    'mongodb://localhost:27017'
9  );
10
11 $collection = $client->website->cookbook;
12
13 $options = [];
14 $options['sort'] = ['title' => 1];
15 $options['limit'] = 3;
16
17 $recipes = $collection->find([], $options);
```

```
18
19     foreach ($documents as $document) {
20       echo $document['title'] . "\n";
21     }
22
23     echo "</pre>";
```

Notice that we have setup the connection slightly differently, just to show you can do so, if you wish.

The real differences start at lines 13-15, where we have created an **`$options`** array and appended a **`sort`** option on line 14, and a **`limit`** option on line 15.

Then, on line 17, we run a **`find()`** on our collection and pass an empty array **`[]`** as the first parameter. This is the same as passing an empty document to the **`find()`** method in the MongoDB Shell, so the same query could be written like this:

```
> db.cookbook.find({}).sort({'title' : 1}).limit(3)
```

In most cases, where you might use an object, in PHP you will use an array, for example when inserting a document as shown in the next code snippet.

Inserting a document with PHP

Creating a new document in your collection is fairly straightforward with PHP; we will only use arrays as previously mentioned. In this example, we will insert a new document and then return back that document's title.

```
1     <?php
2
3     require 'vendor/autoload.php';
4
5     $client = new MongoDB\Client(
6       'mongodb://localhost:27017'
7     );
8
9     $collection = $client->website->cookbook;
10
11    $document = [
```

```
12      'title' => 'Waffles',
13      'directions' => [
14        'Mix batter',
15        'Cook on waffle iron until golden brown',
16        'Put lots of butter and maple syrup'
17      ]
18    ];
19
20    $result = $collection->insertOne($document);
21
22     if ($result->getInsertedCount() === 1) {
23      $_id = $result->getInsertedId();
24
25      $insertedDoc = $collection->findOne(["_id" => $_id]);
26
27      echo "Successfully insert-
      ed: " . $insertedDoc['title'] . "\n\r";
28    }
```

In lines 11-18, we have an array variable called **`$document`**, which represents our document's object. This variable gets passed to **`insertOne()`** on line 20 and then we do an **`if`** check on line 22 to see if the document inserted correctly.

Next, we get the **`_id`** of the inserted document by using **`getInsertedId()`** on line 23 and use that id to get back the newly inserted document and output its title on lines 25-27.

Aggregation with PHP

Running an aggregation pipeline via PHP is fairly straightforward. In this example, we will show a two-step pipeline for a common grouping operation:

```
1     <?php
2
3     require 'vendor/autoload.php';
4
5     $client = new MongoDB\Client(
```

```
6       'mongodb://localhost:27017'
7   );
8
9   $collection = $client->website->cookbook;
10
11  $pipeline = [
12    [
13      '$group' => [
14        '_id' => '$type',
15        'count' => ['$sum' => 1]
16      ]
17    ],
18    [
19        '$sort' => ['count' => -1]
20    ],
21  ];
22
23  $cursor = $collection->aggregate($pipeline);
24
25   foreach ($cursor as $type) {
26      echo $type['_id'] . " has " . $type['count'] . " reci-
    pe(s) \n\r";
27  }
```

In lines 11-21, we have made a rather simple aggregation pipeline, with each step as a member of a **`$pipeline`** array. We then send that array to the **`aggregate()`** method in line 23.

Each step in the pipeline is specified with a dollar sign $ just like within the MongoDB Shell. Here we are grouping on a **`type`** field in lines 12-17, and getting a **`count`** for how many recipe documents are of each **`type`**. Then we **`$sort`** by the **`count`** in descending order in line 19.

Lastly, we then assign the results to **`$cursor`** which we can iterate over in lines 25-27, to display our results. If you run this PHP file, depending on the actual documents you have in your collection, you should get output that looks somewhat like this:

```
Dinner has 4 recipe(s)
Dessert has 2 recipe(s)
Breakfast has 1 recipe(s)
```

Using MongoDB with Laravel

MongoDB can also be used with many popular PHP frameworks, such as Laravel. To do so, you will need to install the MongoDB Laravel package:

```
$ composer require jenssegers/mongodb
```

Then setup a configuration:

```
use Illuminate\Support\Facades\DB;

// Set up a MongoDB connection
$config = [
    'driver' => 'mongodb',
    'host' => env('MONGO_DB_HOST', 'localhost'),
    'port' => env('MONGO_DB_PORT', 27017),
    'database' => env('MONGO_DB_DATABASE'),
    'username' => env('MONGO_DB_USERNAME'),
    'password' => env('MONGO_DB_PASSWORD'),
    'options' => [
        'database' => 'admin' // use admin database to authenticate
    ],
];
// Connect to MongoDB
DB::connection('mongodb')->config($config);
```

Inside a Model, you can use MongoDB in this fashion:

```
namespace App\Models;

use Jenssegers\Mongodb\Eloquent\Model;

class Cookbook extends Model
```

```
{
    protected $connection = 'mongodb';
    protected $collection = 'cookbook';
}
```

Then in your Controller code, you can use the Model methods generally in the same way as you would with other data sources:

```
use App\Models\Cookbook;

// find all
$recipe = Cookbook::all();

// by id
$recipe = Cookbook::find('636821387dd21c28fda4939f');

// by field, also supports dot notation
$recipes = Cookbook::where('title', 'Toast')->get();
```

The PHP driver is very powerful and has a lot more features not mentioned here. Thus, if PHP is your thing, make sure to check out the official documentation for more.

Other language examples

There are other languages that support MongoDB in one way or another. In this section, we have simple examples of connecting to a database and running a **`findOne()`** query in Go, C#, Java, Kotlin, Rust, C++, Ruby, Swift, Perl, Scala and even BASH.

These are provided to give you a very basic idea of how MongoDB works with these languages. Make sure to check out the official documentation if you plan to actually program with them. In some cases, there are directions to install the driver; for others, you will need to look up the current instructions online.

Go

The latest instructions for installing the MongoDB Go driver are available on the official website **https://www.mongodb.com/docs/drivers/go/**; however, generally speaking, you should be able to run the following to add it as dependency:

```
$ go get go.mongodb.org/mongo-driver/mongo
```

Here is an example of some Go code in action:

```
1   package main
2
3   import (
4   "context"
5   "fmt"
6   "log"
7
8   "go.mongodb.org/mongo-driver/mongo"
9   "go.mongodb.org/mongo-driver/mongo/options"
10  "go.mongodb.org/mongo-driver/bson"
11  )
12
13  type Recipe struct {
14  Title string
15  Ingredients []string
16  }
17
18  func main() {
19  // Set client options
20  clientOptions := options.Client().ApplyURI("mongodb://local-
    host:27017")
21
22  // Connect to MongoDB
23  client, err := mongo.Connect(context.
    Background(), clientOptions)
24  if err != nil {
25   log.Fatal(err)
26  }
27
28  // Check the connection
29  err = client.Ping(context.Background(), nil)
30  if err != nil {
```

```
    log.Fatal(err)
  }

  // Access database and collection
  collection := client.Database("website").Collection("cookbook")

  // Find one document with the title "Waffles"
  var result Recipe
  err = collection.FindOne(context.
Background(), bson.M{"title": "Waffles"}).Decode(&result)
  if err != nil {
    log.Fatal(err)
  }

  // Print the result
  fmt.Println(result)
 }
```

A big difference with Go is the typing. Here, we have created a simple type called **`Recipe`**, but in production code, this would need to be more robust.

C#

You will need to install the MongoDB driver via your preferred package manager, as shown:

```
using System;
using MongoDB.Driver;
using MongoDB.Bson;

namespace MongoDBExample
{
  class Program
  {
    static void Main(string[] args)
    {
```

```
        // Set connection string and client options
        string connectionString = "mongodb://localhost:27017";
        MongoClientSettings settings = MongoClientSettings.FromUrl
(new MongoUrl(connectionString));
        MongoClient client = new MongoClient(settings);

        // Access database and collection
        IMongoDatabase database = client.GetDatabase("website");
        IMongoCollection<BsonDocument> collection = database.GetCo
llection<BsonDocument>("cookbook");

        // Find one document with the title "Waffles"
        BsonDocument result = collection.
FindOne(new BsonDocument("title", "Waffles"));

        // Print the result
        Console.WriteLine(result);
      }
    }
  }
```

Java

Refer to the following code:

```
  import com.mongodb.MongoClient;
  import com.mongodb.MongoClientURI;
  import com.mongodb.client.MongoCollection;
  import com.mongodb.client.MongoDatabase;
  import org.bson.Document;

  public class MongoDBExample {
    public static void main(String[] args) {
      // Set connection URI and client options
      String connectionString = "mongodb://localhost:27017";
      MongoClientURI uri = new MongoClientURI(connectionString);
```

```
12        MongoClient client = new MongoClient(uri);
13
14        // Access database and collection
15        MongoDatabase database = client.getDatabase("website");
16        MongoCollection<Document> collection = database.getCollec-
    tion("cookbook");
17
18        // Find one document with the title "Waffles"
19        Document result = collection.find(new Document("title", "Waf-
    fles")).first();
20
21        // Print the result
22        System.out.println(result);
23     }
24  }
```

Kotlin

You will need to add the MongoDB driver as a dependency in your **`build.gradle`** file, as follows:

```
1   import com.mongodb.client.MongoClients
2   import org.bson.Document
3
4   fun main() {
5       // Set connection URI and client options
6       val connectionString = "mongodb://localhost:27017"
7       val client = MongoClients.create(connectionString)
8
9       // Access database and collection
10      val database = client.getDatabase("website")
11      val collection = database.getCollection("cookbook")
12
13      // Find one document with the title "Waffles"
14      val result = collection.find(Document("title", "Waffles")).first()
```

```
15
16        // Print the result
17        println(result)
18    }
```

Rust

Refer to the following code:

```
1     use mongodb::{bson::doc, Client};
2
3     #[tokio::main]
4     async fn main() -> Result<(), Box<dyn std::error::Error>> {
5        // Set connection string and client options
6        let uri = "mongodb://localhost:27017";
7        let client = Client::with_uri_str(uri).await?;
8
9        // Access database and collection
10       let database = client.database("website");
11       let collection = database.collection("cookbook");
12
13       // Find one document with the title "Waffles"
14       let filter = doc! {"title": "Waffles"};
15       let result = collection.find_one(filter, None).await?;
16
17       // Print the result
18       println!("{:#?}", result);
19
20       Ok(())
21    }
```

C++

Refer to the following code:

```
1     #include <mongocxx/client.hpp>
```

```
#include <mongocxx/instance.hpp>
#include <mongocxx/uri.hpp>
#include <bsoncxx/json.hpp>

using bsoncxx::builder::stream::document;
using bsoncxx::builder::stream::finalize;
using bsoncxx::document::view_or_value;
using mongocxx::client;
using mongocxx::collection;
using mongocxx::database;
using mongocxx::instance;
using mongocxx::uri;

int main() {
    // Set connection URI and client options
    uri connection_uri("mongodb://localhost:27017");
    instance inst{};
    client conn{connection_uri};

    // Access database and collection
    database db = conn["website"];
    collection coll = db["cookbook"];

    // Find one document with the title "Waffles"
    auto filter = document{} << "title" << "Waffles" << finalize;
    auto result = coll.find_one(filter.view());

    // Print the result
    std::cout << bsoncxx::to_json(*result) << std::endl;

    return 0;
}
```

Ruby

You will want to install the gem first:

```
$ gem install mongo
```

```ruby
require 'mongo'

# Set connection URI and client options
uri = 'mongodb://localhost:27017'
client = Mongo::Client.new(uri)

# Access database and collection
db = client.database
coll = db['cookbook']

# Find one document with the title "Waffles"
filter = { 'title' => 'Waffles' }
result = coll.find(filter).first

# Print the result
puts result
```

Swift

You will need to add the MongoDB driver as a dependency in your **`Package.swift`** file.

```swift
import MongoSwift

// Set connection URI and client options
let uri = "mongodb://localhost:27017"
let client = try MongoClient(uri)

// Access database and collection
let db = client.db("website")
let coll = db.collection("cookbook")
```

```
10
11      // Find one document with the title "Waffles"
12      let filter: Document = ["title": "Waffles"]
13      let result = try coll.find(filter).next()
14
15      // Print the result
16      print(result)
```

Perl

To install the Perl drivers, use the following command:

```
cpanm MongoDB
```

```
1   use strict;
2   use warnings;
3   use MongoDB;
4
5   # Set connection URI and client options
6   my $uri = 'mongodb://localhost:27017';
7   my $client = MongoDB->connect($uri);
8
9   # Access database and collection
10  my $db = $client->ns('website');
11  my $coll = $db->ns('cookbook');
12
13  # Find one document with the title "Waffles"
14  my $filter = { title => 'Waffles' };
15  my $result = $coll->find_one($filter);
16
17  # Print the result
18  print "$result\n";
```

Scala

Refer to the following code:

```
import org.mongodb.scala._

// Set connection URI and client options
val uri: String = "mongodb://localhost:27017"
val client: MongoClient = MongoClient(uri)

// Access database and collection
val db: MongoDatabase = client.getDatabase("website")
val coll: MongoCollection[Document] = db.getCollection("cookbook")

// Find one document with the title "Waffles"
val filter: Document = Document("title" -> "Waffles")
val result: Option[Document] = coll.find(filter).headResult()

// Print the result
println(result)
```

Bash

This last example is not exactly code, but it shows how to use BASH if you want, in order to perform the same sort of operation:

```
 #!/bin/bash

# Set connection URI and database/collection names
uri="mongodb://localhost:27017"
database="website"
collection="cookbook"

# Connect to the MongoDB instance and find the docu-
ment with the title "Waffles"
```

```
    result=$(mongo --quiet --eval "db = connect('$uri/$data-
base'); db.$collection.findOne({title: 'Waffles'})")

    # Print the result
    echo "$result"
```

Conclusion

Hopefully, this chapter has whetted your appetite for programming with MongoDB. Make sure to check out the code samples in the book's git repo, or even try some of them out in the GitHub Codespace.

In the next chapter, we will be covering the client interfaces for MongoDB in more detail, including MongoDB Shell and MongoDB Compass, before moving on to covering MongoDB provided services such as MongoDB Atlas in *Chapter 16, Cloud Services – MongoDB Atlas* and serverless programming with MongoDB Stitch in *Chapter 17 – MongoDB Atlas – Application Services*.

Chapter 15

Tools for Success – MongoDB Shell and Compass UI

> **"The best investment is in the tools of one's own trade."**
>
> *— Benjamin Franklin*

> **"Technology is nothing. What's important is that you have a faith in people, that they're basically good and smart, and if you give them tools, they'll do wonderful things with them."**
>
> *— Steve Jobs*

Introduction

Knowing how to effectively use some of the core tools for MongoDB will greatly enhance both your day-to-day experience working with MongoDB, and also set you apart from others in your efficiency.

Structure

In this chapter, we will discuss the following topics:

- MongoDB Shell
- Visual Studio Code
- MongoDB Compass

Objectives

In this chapter, we will dig a bit deeper into some of the most useful and common tools you will use with MongoDB. We will learn how to configure and personalize the *MongoDB Shell*, **mongosh**, as well as how to create useful custom functions you can use within the shell. Then, we will explore the *MongoDB Visual Studio Code* extension, and *MongoDB Playgrounds*. Lastly, we will do a bit of a review, as well as a more expansive investigation of *MongoDB Compass*, the official GUI for MongoDB. By the end of this chapter, you should be empowered to take your use of all these tools, to the next level.

MongoDB Shell

We have been using the MongoDB Shell all along and have already considered some of its features. In this section, we will dive deeper into how to configure and get more out of **mongosh**.

Configuration

One thing a lot of us love is a customized command prompt. You can personalize it by putting any emojis as part of the prompt.

You can customize your own prompt and a number of other things with the **.mongoshrc.js** file. Following is an example that will turn your prompt into a line that shows the name of database you are in, plus a disk emoji:

```
1    const prompt = () => {
2      return `💾 ${db.getName()} >`
3    }
```

By defining the **prompt** variable, we are resetting the prompt in **mongosh**. Here we also used the **db.getName()** method to get the current database name, so as we move around databases we know where we are, which can help avoid mistakes or confusion. You can get much more complicated if you wish, by adding the server name, and all sorts of other customization if you wish.

You can see an example **.mongoshrc.js** file, with all the customizations we will talk about in this chapter, in the book's git repository.

Editor mode

If you find yourself editing a lot of text within mongosh, you might want to take advantage of the config option to set an external editor. You can choose any editor you have installed, such as **vi**, **vim**, or as shown here, with **nano**:

```
> config.set( "editor", "nano" )
Setting "editor" has been changed
```

Via the **mongosh** shell, we used **config.set()** to set our editor, and now when you use the **edit** keyword, it will open the external editor instead:

```
> var test = []
> edit test
Opening an editor...
> test = [1,2,3]
```

This opened the **nano** editor with the line **var test = []** prepopulated and allowed us to update the array. When the editor exited, it populated our edited variable in the shell. You can also use the editor with a statement, or query as shown:

```
> edit db.recipes.insertMany( [] )
```

This will open the editor again and let you edit the query and populate the array with complex documents, inside your preferred editor, which can make things a lot easier, all while not leaving the command line. After you exit your editor, the whole query you just edited will show up in the shell, ready to run.

You can use the **config** API any time to set this via the shell, as shown here, or add this to a **mongosh.conf** file.

See **https://www.mongodb.com/docs/mongodb-shell/configure-mongosh/** for more information about configuring **mongosh** settings.

Node.JS Scripting

Since **mongosh** is a Node.JS, or more simply, JavaScript environment, you can create your own scripts to use within the shell. To do this, include your custom code directly in your **.mongoshrc.js** file. It will be available when you load **mongosh**, such as this **getDate()** function we made to more easily create **ISODates**, which you can then use within your shell. Refer to *Figure 15.1*:

```
cookbook > getDate(2023,09,07)
ISODate("2023-09-07T00:00:00.000Z")
cookbook > var date = getDate(2023,09,07);

cookbook > date
ISODate("2023-09-07T00:00:00.000Z")
cookbook >
```

Figure 15.1: *Using a Custom Function*

Using more robust code, we can leverage the Node.JS support in **mongosh** for data population. You could pull data from backups, or some other source, such as an API. Using Node.JS, you can easily write scripts to transform data from JavaScript to MongoDB documents.

As a more intricate example, which we have included below, we have created a collection of functions to generate random recipe documents. Along with custom code, it uses a freely available node package called **falso**.

With these functions, we can generate a nearly unlimited number of example recipe documents. While going line by line on these functions is out of scope for this book, you can see the code in the book's git repository, and work with it the Codespace, or your local machine.

```
1   const falso = require('@ngneat/falso');
2
3   const unitTypes = [undefined, "teaspoon", "table-
    spoon", "cup", "liter", "milliliter"];
4
5   const getIngredient = () => {
6    const unit = unitTypes[falso.randNumber({ min: 0, max: 4 })];
7    const origin = ['france', 'japan', 'mexico'][falso.
    randNumber({ min: 0, max: 2 })]
8
9    return {
10     "name": falso.randFood({ origin }),
11     "quantity": {
12       "amount": falso.randNumber({ min: 1, max: 10 }),
13       ...unit && { unit }
14     },
15   }
16 };
17
18  const getIngredients = (count = 5) => Array(count).fill().
    map(() => getIngredient());
19
20  const getDirections = (steps = 5) => falso.
    randTextRange({ min: 10, max: 50, length: steps });
```

```
21
22    const getRatings = () => falso.
      randNumber({ min: 1, max: 5, length: falso.
      randNumber({ min: 0, max: 5 }) });
23
24    const generateRecipes = (documentCount = 100, recipes = []) => {
25
26      while (documentCount--) {
27        const ratings = getRatings();
28        recipes.push({
29          "title": falso.randFood(),
30          "servings": falso.randNumber({ min: 1, max: 12 }),
31          "calories_per_serving": falso.
      randNumber({ min: 100, max: 800 }),
32          "cook_time": falso.
      randNumber({ min: 10, max: 120, precision: 5 }),
33          "prep_time": falso.
      randNumber({ min: 5, max: 30, precision: 5 }),
34          "description": falso.randSentence(),
35          "ingredients": getIngredients(falso.
      randNumber({ min: 4, max: 10 })),
36          "directions": getDirections(falso.
      randNumber({ min: 3, max: 8 })),
37          ...ratings.length && { ratings },
38          "vegetarian_options": falso.randBoolean(),
39          "suggested_drink": falso.randDrinks(),
40          "tags": falso.randWord({ length: falso.
      randNumber({ min: 1, max: 5 }) })
41        });
42      }
43
44      return recipes;
45    }
```

The important bits are in the **generateRecipes()** function starting on line 24.

This uses **falso** and some custom functions to create a realistic looking recipe document. It generates a random **title**, cook and prep times, an array of complex **ingredient** objects as well as **directions**, and at random, a possible **ratings** array, and so on.

Using this function, we can create as many example documents we wish, and they will look nearly like the real recipe documents we have been using. To leverage this, we create a function, starting on line 8, called **populateCookbook()** within our **.mongoshrc.js** file:

```
1   load('./generators/recipes/fake-recipes.js');
2
3    const getCookbookCount = () => {
4     use('cookbook');
5     return db.recipes.countDocuments();
6   }
7
8    const populateCookbook = () => {
9     use('cookbook');
10
11    print('\n Starting to populate cookbook›)
12    print(`There are currently ${getCookbook-
    Count()} recipe documents ...`)
13
14    db.recipes.insertMany(generateRecipes());
15
16    db.recipes.updateMany(
17      { ratings: { $exists: true }, rating_
    avg: { $exists: false } },
18      [{ $set: { rating_
    avg: { $round: [{ $avg: "$ratings" }, 2] } } }]
19    );
20
21    print(` The cookbook collection now has ${getCookbook-
    Count()} recipe documents.`)
22  }
```

On line 1, in this example, we are loading a file named **`fake-recipes.js`**, that has the functions we wish to use. Then, in lines 3-6, we have a function that will get the count of documents in our **`recipes`** collection, followed by our **`populateCookbook()`** function on lines 8-19.

This function leverages **`generateRecipes()`**, taking its output and running an **`insertMany()`** on our collection. After that, on line 16, we perform an aggregation with **`updateMany()`** to calculate the average rating for each recipe, which will add a new **`rating_avg`** field in line 18, for the documents where that is relevant. Finally, we run the count again and output a message for the user.

This will create 100 "fake" recipe documents that look something like this:

```
{
  "title": "Guacamole",
  "servings": 8,
  "calories_per_serving": 701,
  "cook_time": 35,
  "prep_time": 10,
 "description": "Qui sunt quasi nihil repellat nulla asperiores, sunt.",
  "ingredients": [
    {
      "name": "Mole",
      "quantity": { "amount": 2, "unit": "cup" }
    },
    {
      "name": "Escargots au beurre persillé",
      "quantity": { "amount": 9, "unit": " teaspoon " }
    },
  ...
  ],
  "directions": [
    "Quasi quae blanditiis et quaerat magnam numqug.",
    "Aut sequi quasi commodi, vel ve.",
    "Consequatur ducimus conseql."
  ],
  "ratings": [4,3],
```

```
  "vegetarian_options": true,
  "suggested_drink": "Brandy",
  "tags": ["esse", "voluptatibus", "doloribus", "rerum"],
  "rating_avg": 3.5
}
```

In *Figure 15.2*, you can see this function being called, within **`mongosh`**, and populating documents into the collection:

Figure 15.2: *Calling a Custom Function from the Shell*

While obviously quite silly to actually read, creating a lot of realistic looking data can be very helpful for testing applications. The code for this, along with some other tricks, is available in the book's git repository. An example **`mongoshrc`** file is included as well: **`/extras/.mongoshrc.js`**

Note: There is a "." at the start of the file name. Some operating systems might hide this file; use the command line or an IDE to open it.

You can use this method to import the custom code you have, as well other node packages; the possibilities are endless. See **https://www.mongodb.com/docs/mongodb-shell/write-scripts/** for more on how this works.

Snippets

Finally, an experimental feature rolling out for MongoDB is called snippets. A snippet is a script that is packaged and stored in a registry. Currently, this is the working spec on how snippets work: *"You can write your own scripts in mongosh to manipulate data or to perform administrative tasks. Snippets are an improvement on locally stored scripts because they can be easily shared and maintained."*

You can find out more here: **https://www.mongodb.com/docs/mongodb-shell/snippets/**

Visual Studio Code

Many code editors have support for MongoDB; however, Visual Studio Code has an official extension maintained by MongoDB which we will briefly discuss. It should be noted that this extension is currently in "Preview", meaning it may change slightly, in the near future.

MongoDB extension

You can install the extension just like any other. Search for **MongoDB** and choose "**MongoDB for VS Code**", as shown in *Figure 15.3*:

Figure 15.3: *MongoDB for VS Code*

Once installed, you will see a "leaf" icon on the left side bar, and when you click on it, you should see **CONNECTIONS** on the left. If you press the **+** button by **CONNECTIONS**, it will open a page that lets you add a connection. Use whatever connection and login information you have setup for your MongoDB instance. In *Figure 15.4*, we have connected to the **`localhost`**:

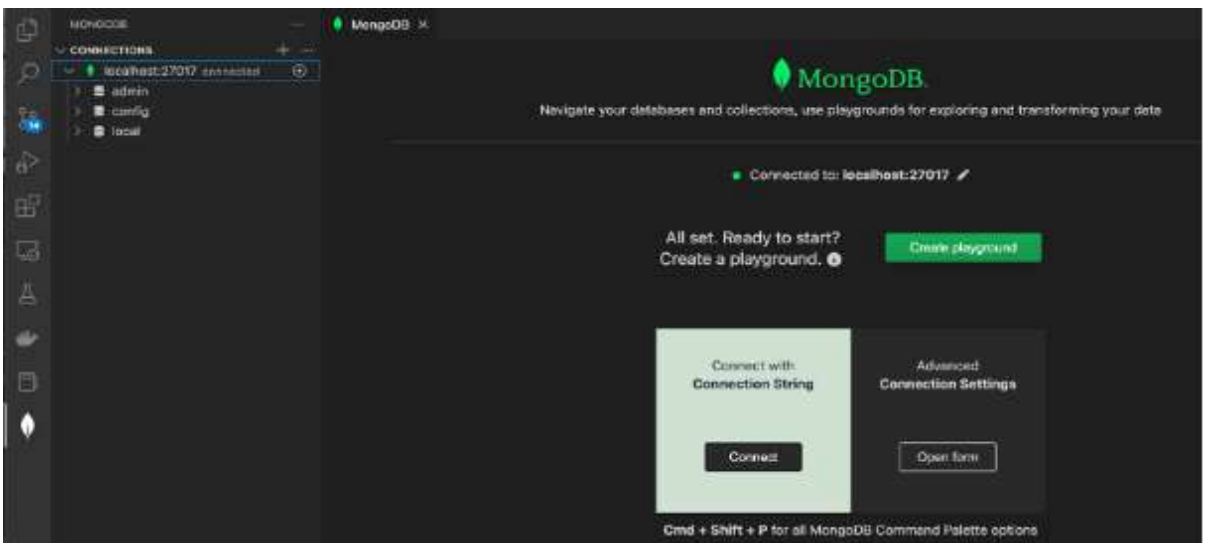


Figure 15.4: *Connecting in VS Code*

You can use this extension to view your databases and collections, as well as documents, which will open as JSON like files within VS Code.

MongoDB Playgrounds

A unique feature of the MongoDB extension is the concept of "playgrounds", which are JavaScript environments where you can run MongoDB commands and queries with syntax highlighting and autocomplete for operators, the shell API, as well as collection and database names. An example playground can be generated for you by pressing the **Create New Playground** button, as seen in *Figure 15.5*:

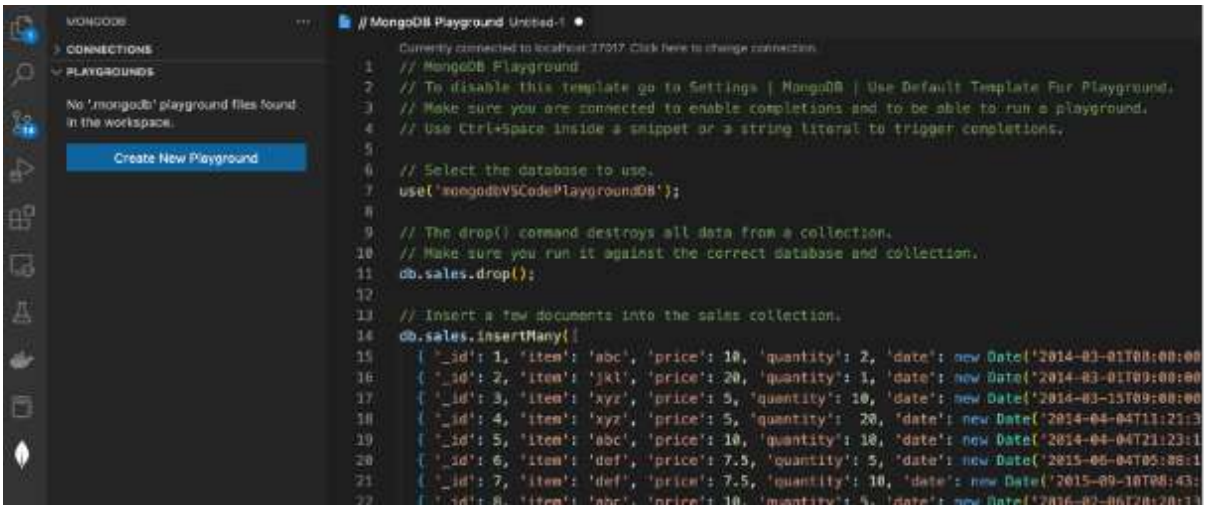


Figure 15.5: *MongoDB Playgrounds*

This playground will create an example database and collection as well as populate it with some example data. Additionally, it includes a find query and aggregation example. You can use these to see how the highlighting and autocomplete work, as seen in *Figure 15.6*:

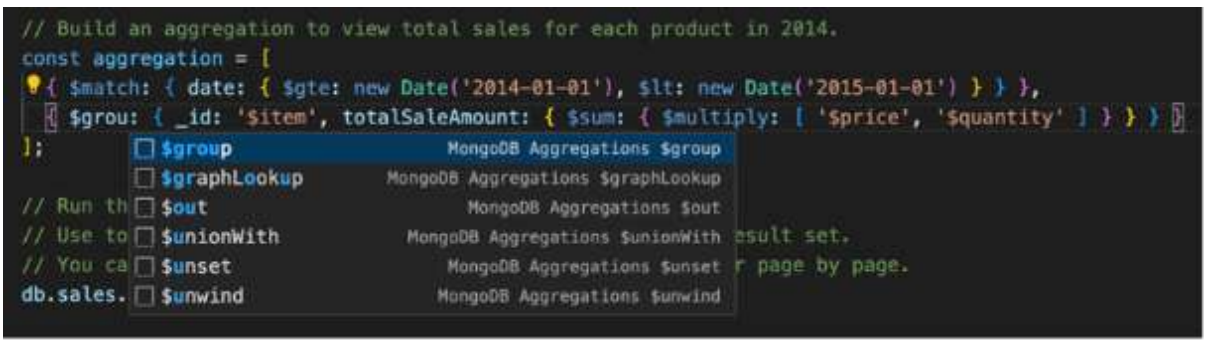


Figure 15.6: *Autocomplete in MongoDB Playgrounds*

This just touches on the features in a playground, so make sure to check that out if this feature interests you. There is also a playground included in this book's git repository and Codespace.

MongoDB compass

We have also used MongoDB Compass throughout this book. In this section, we will group together and look at some of the features it has, in a bit more detail.

Advanced connection options

A convenient feature of Compass is configuring and building your *Connection String*. This can get rather complex, or hard to remember how to format, as more and more options are required. So, make sure to check out the **Advanced Connection Options** right under the **New Connection** area when you open MongoDB Compass, if this applies to you. You can see in *Figure 15.7*, the various configuration options that are available:

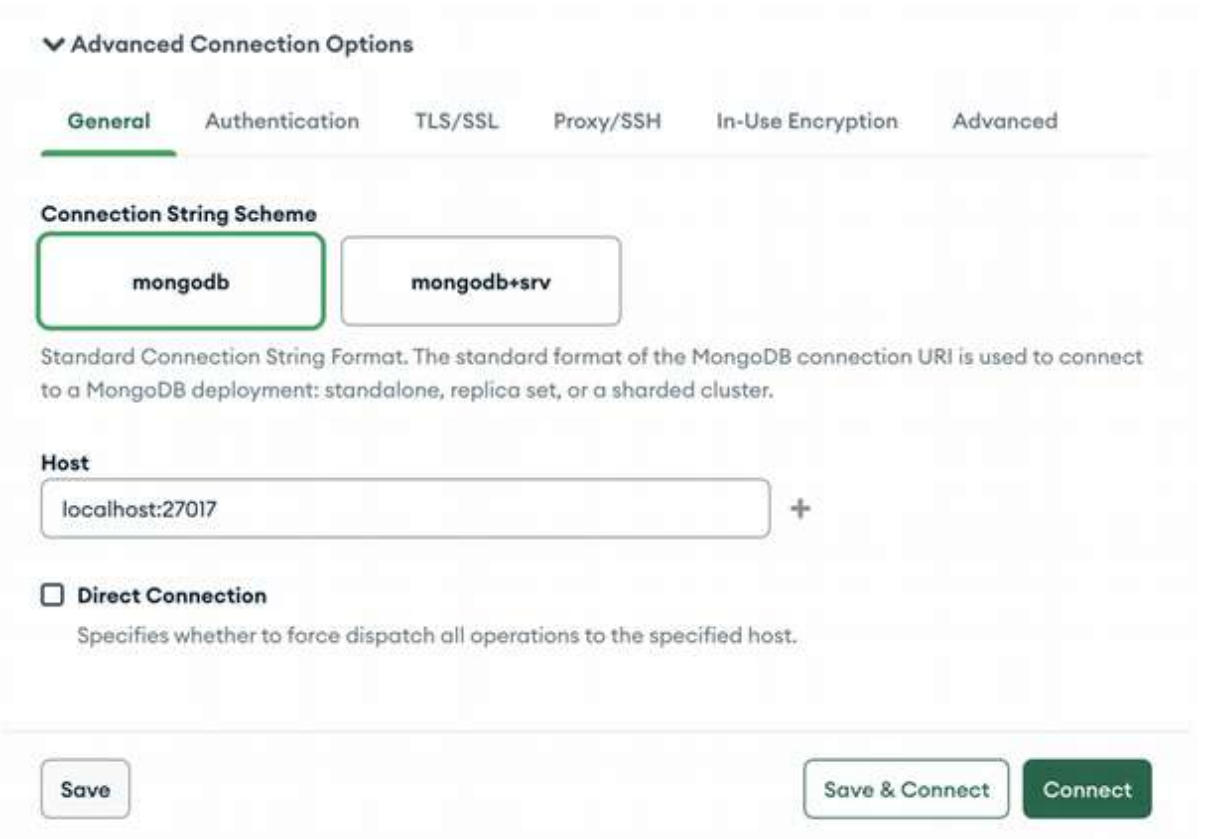


Figure 15.7: *Compass Connection Options*

If you are connecting to a server MongoDB that is running on an internal server that does not allow external connections, but you do have SSH access, or the like, the **`Proxy/SSH`** tab may be very useful to you, as shown here in *Figure 15.8*:

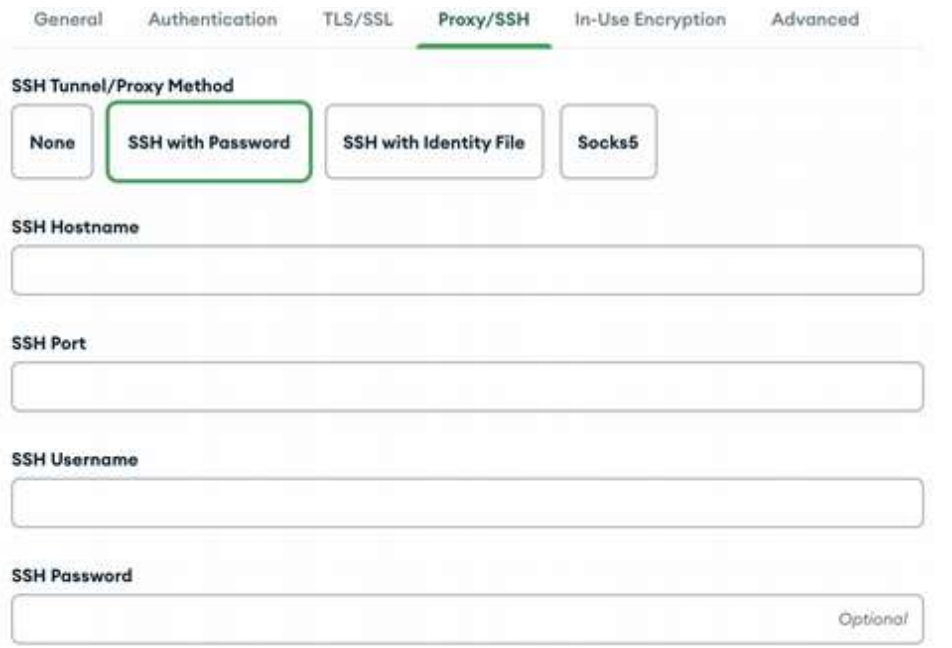


Figure 15.8: *Compass SSH Options*

Lastly, if you are connecting to a *Replica Set*, and need to adjust something like the *Read Preference* that we discussed in *Chapter 12, Seamless Scaling – Replication and Sharding*, or other options, you can set those under the **Advanced** tab, as seen in *Figure 15.9*:

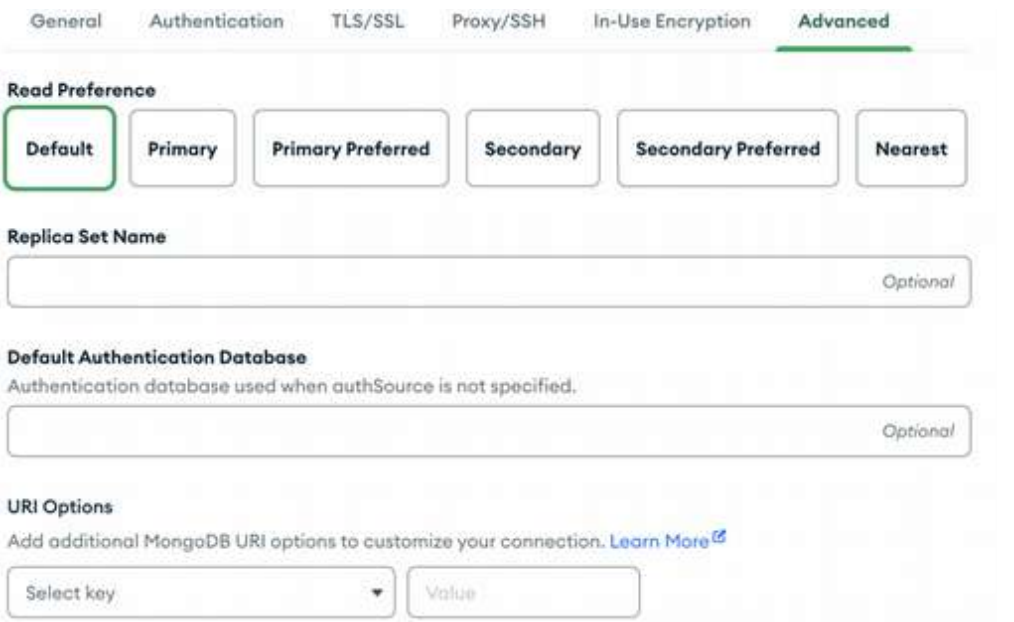


Figure 15.9: *Compass Advanced Connections*

Creating Collections

Many of the options we discussed in *Chapter 9, Planning for Performance – Collections and Indexes,* are available when you create a collection in Compass. There are also some helpful descriptions and links to documentation, as shown in *Figure 15.10*:

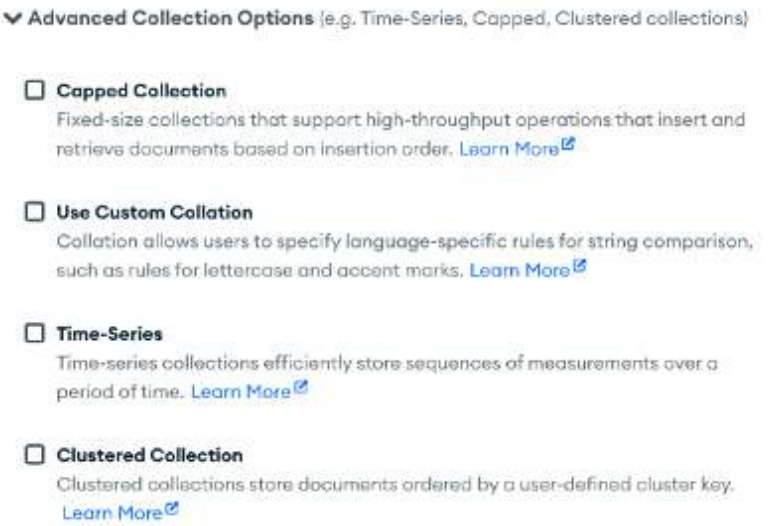


Figure 15.10: *Compass Create Collection*

My Queries

Compass also has a nice feature which keeps track of your recent queries and lets you "favorite" or save, them. In *Figure 15.11*, we are saving a recently run query as "**`Quick Meals`**". You can access your saved queries on the left navigation section under **{} My Queries.**

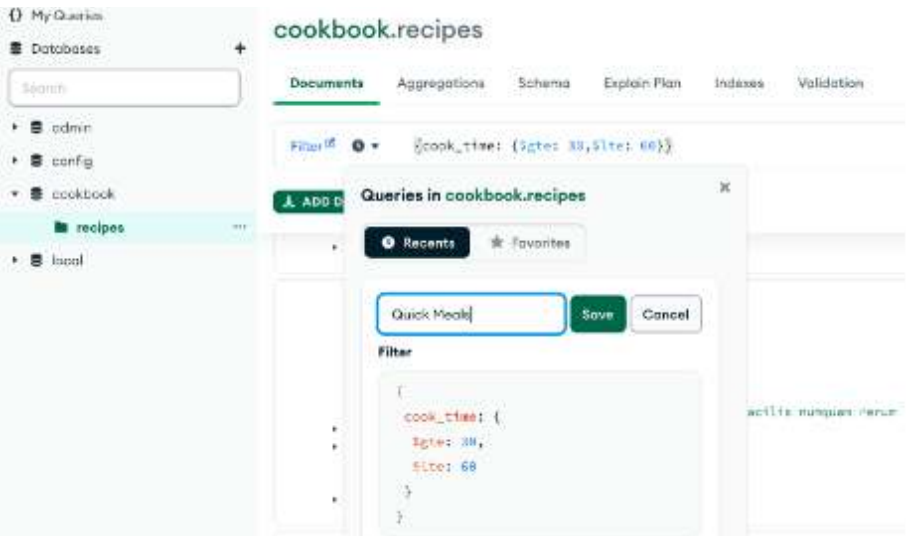


Figure 15.11: *Compass My Queries*

Exporting Queries

Using the **</>** button, right next to the **Find** button on the Documents tab, you can export the query you have typed to various different languages. This can be helpful if you do not want to rewrite your query into a programming languages slightly different format, or if you simply do not know how to write that query in a particular language.

In *Figure 15.12*, we are exporting the query into **C#**, but there are many more languages supported as well.

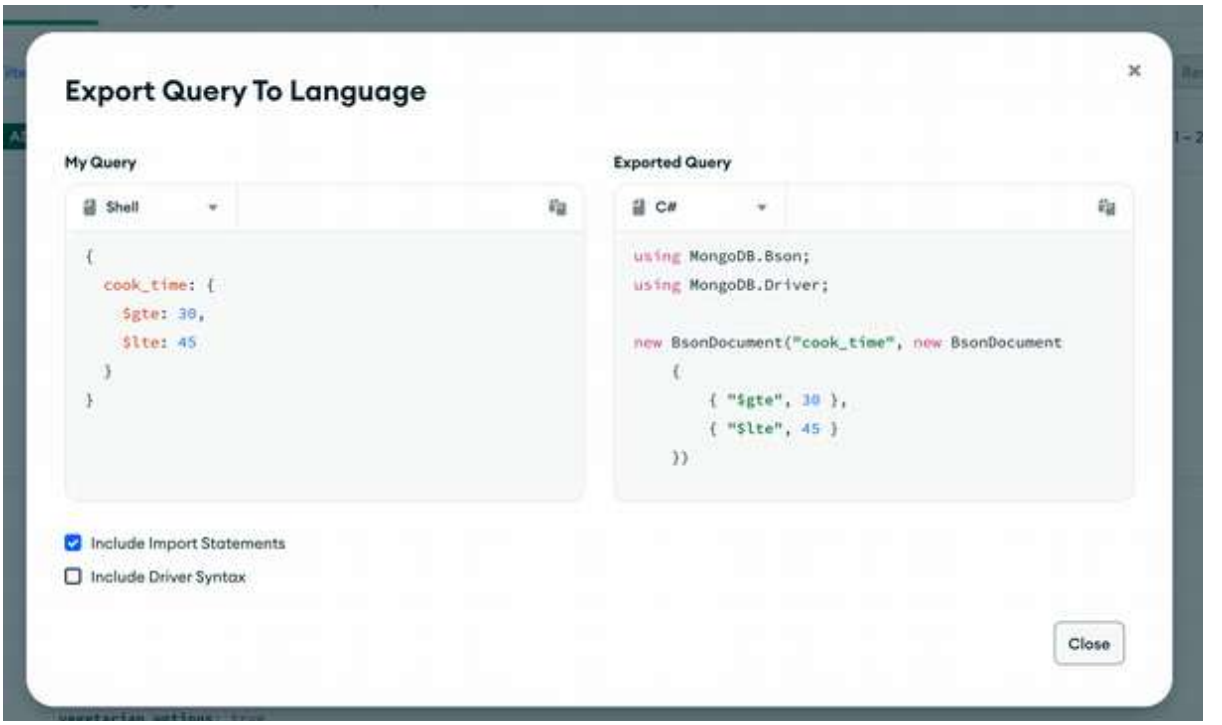


Figure 15.12: *Compass Export Query to Language*

Aggregation Framework and Pipelines

We touched on Compass' support for pipelines in *Chapter 8*, *The MongoDB Aggregation Framework*, but as a bit of a refresher. Compass has two main ways to view and modify pipelines. Under the **Aggregations** tab, first, there is the classic view, which breaks down each step into its own UI section, as seen in *Figure 15.13*:

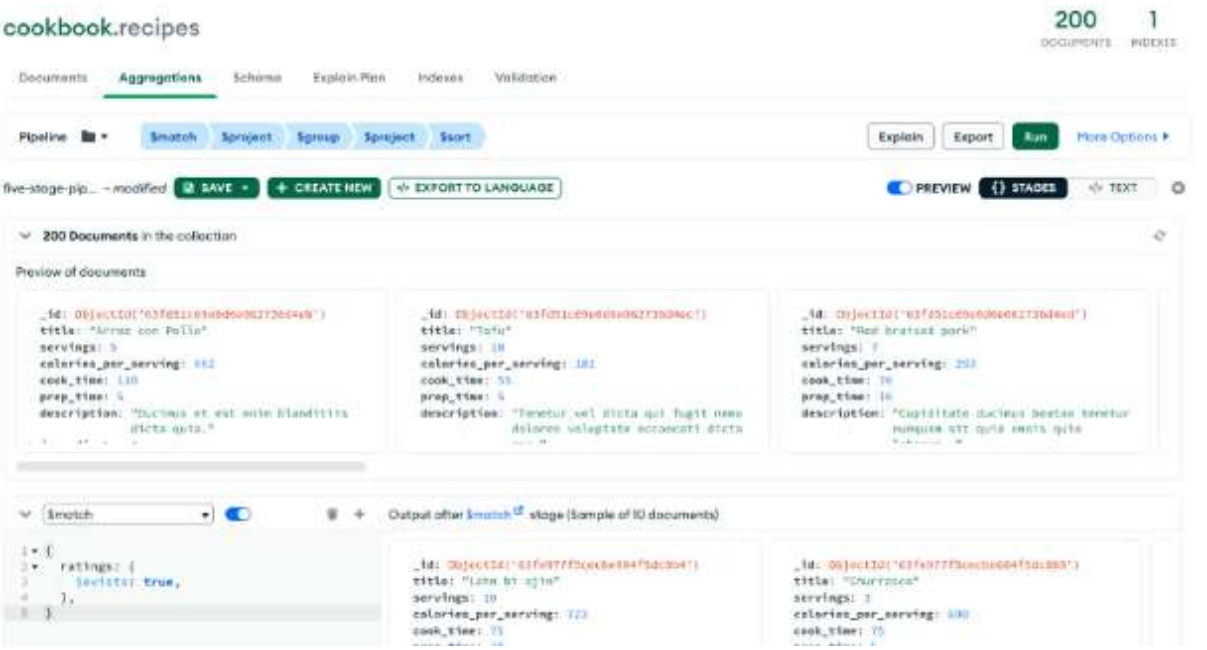


Figure 15.13: *Aggregation in Compass*

The other view, in newer versions of Compass, allows you to use a special inline text editor. You can toggle to this view by switching the **</> TEXT**, as seen in *Figure 15.14*:

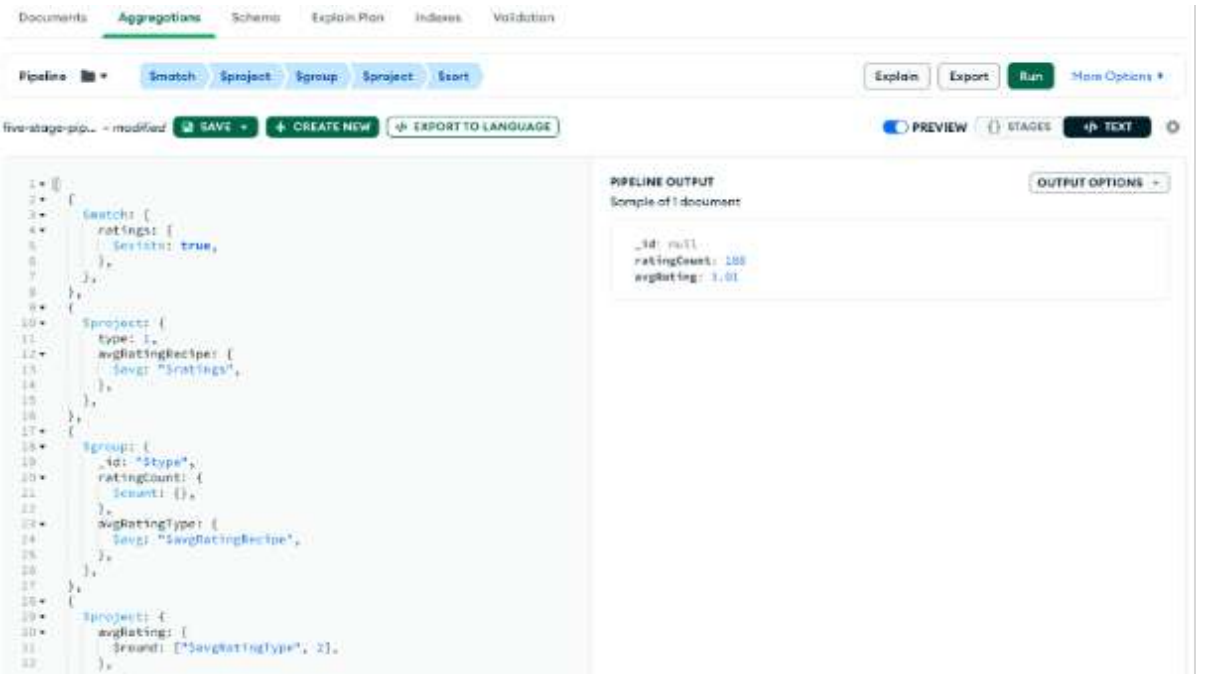


Figure 15.14: *Edit Aggregation in Compass*

This view can be very useful if you prefer to write out your query in the same way you might on the command line.

Schema

A really handy feature, which is especially helpful if you are interested in statistics on your documents and how they are composed, can be accessed under the **Schema** tab, as seen here in *Figure 15.15*:

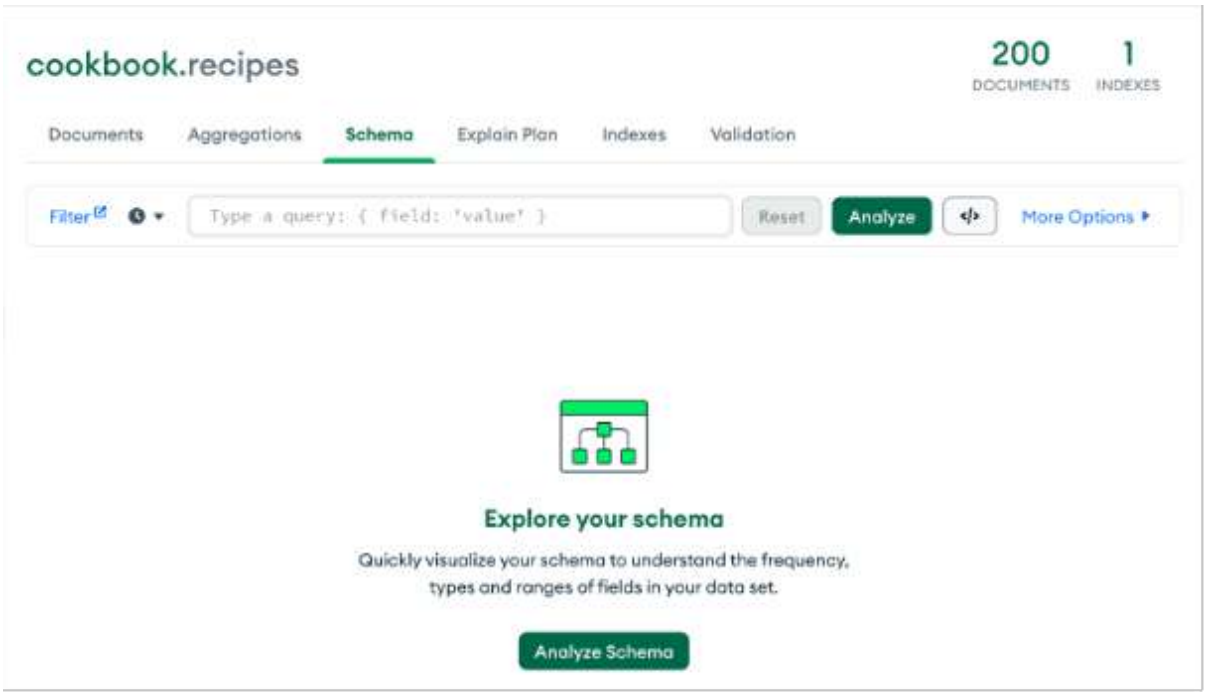


Figure 15.15: *Explore Your Schema*

After pressing the **Analyze Schema** button, Compass will analyze all your collection's documents, compile some statistics, and a series of graphs will show up. Here, we ran this on the "fake" recipes documents we created earlier. In *Figure 15.16*, we can see after the **`suggested_drink`** field was analyzed, and now know that **`3%`** of the recipes suggest a Tequila Sunrise:

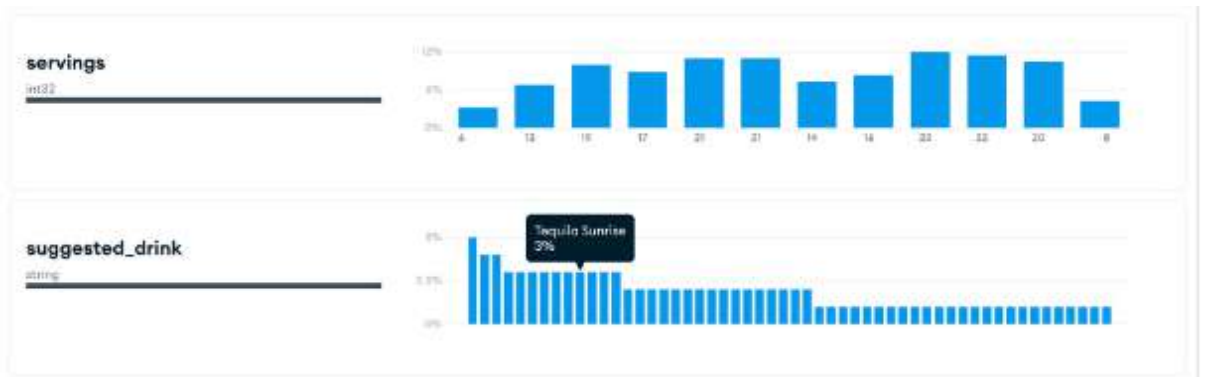


Figure 15.16: *Field Value by Percent, Per Field*

This also works for data *within* arrays. The recipe's **`ingredients`** array has a complex object which contains the ingredient's **`name`** as well as **`quantity`**, and so on. We can see this in *Figure 15.17*:

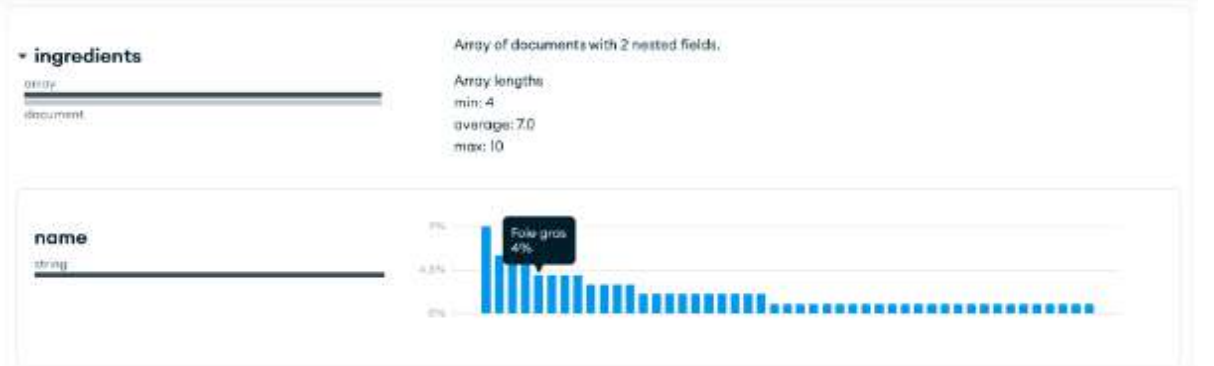


Figure 15.17: *Array Schema Statistics*

Here we can see a whopping **4%** of recipes have "foie gras" as an ingredient.

Explain plan

We reviewed Compass' **Explain Plan** tab already, but for thoroughness, we will show how this works in *Figure 15.18*:

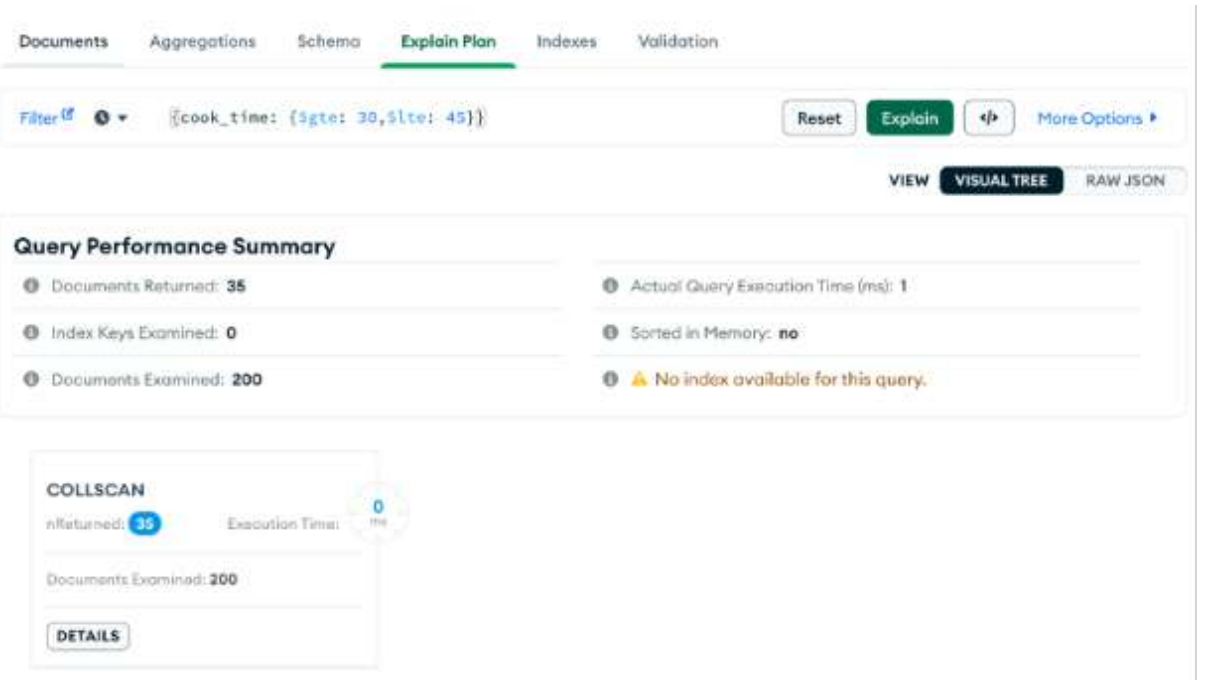


Figure 15.18: *Explain Plan GUI*

A notable option, much like how the newer pipeline view works, you can toggle to **RAW JSON** which will allow you to see the same output as if you ran the command with the MongoDB Shell. Refer to *Figure 15.19*:

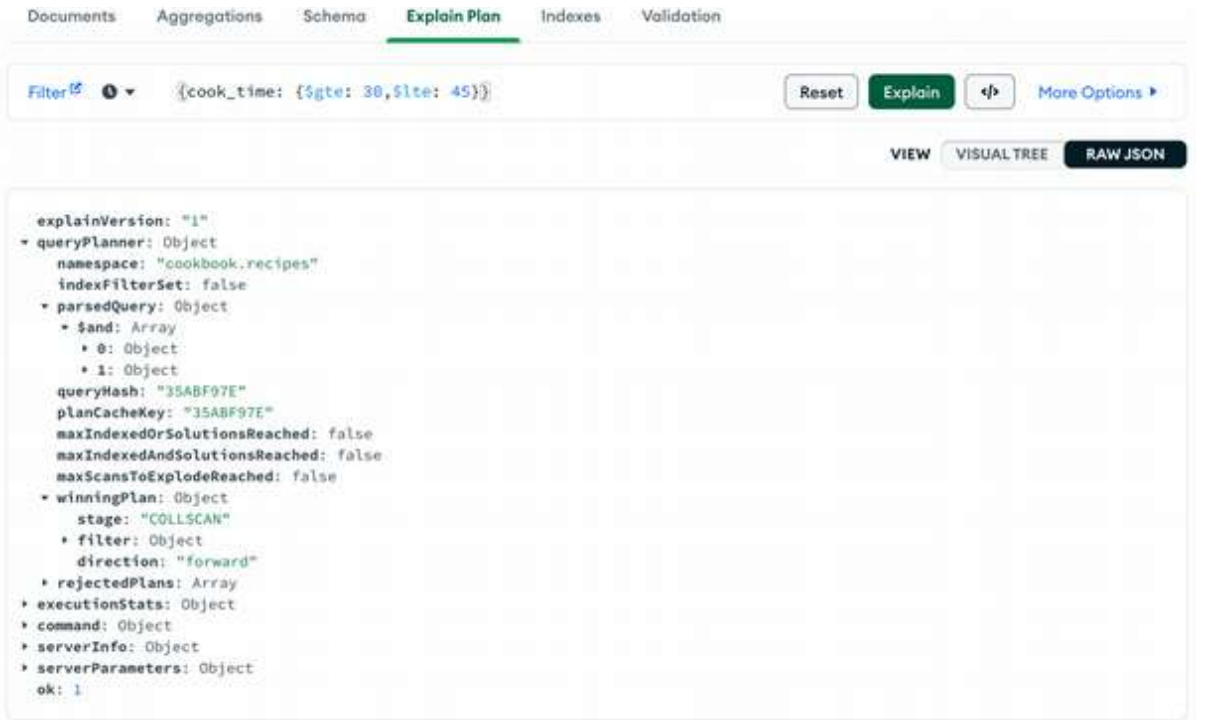


Figure 15.19: *Explain Plan Raw JSON*

Performance monitoring

While fairly limited, there is also some performance related GUI in Compass. You can view this, if you have rights to see it, by pressing on **Databases** on the left navigation pane, and then select the **Performance** tab at the top. In *Figure 15.20,* we can see some of the statistics:

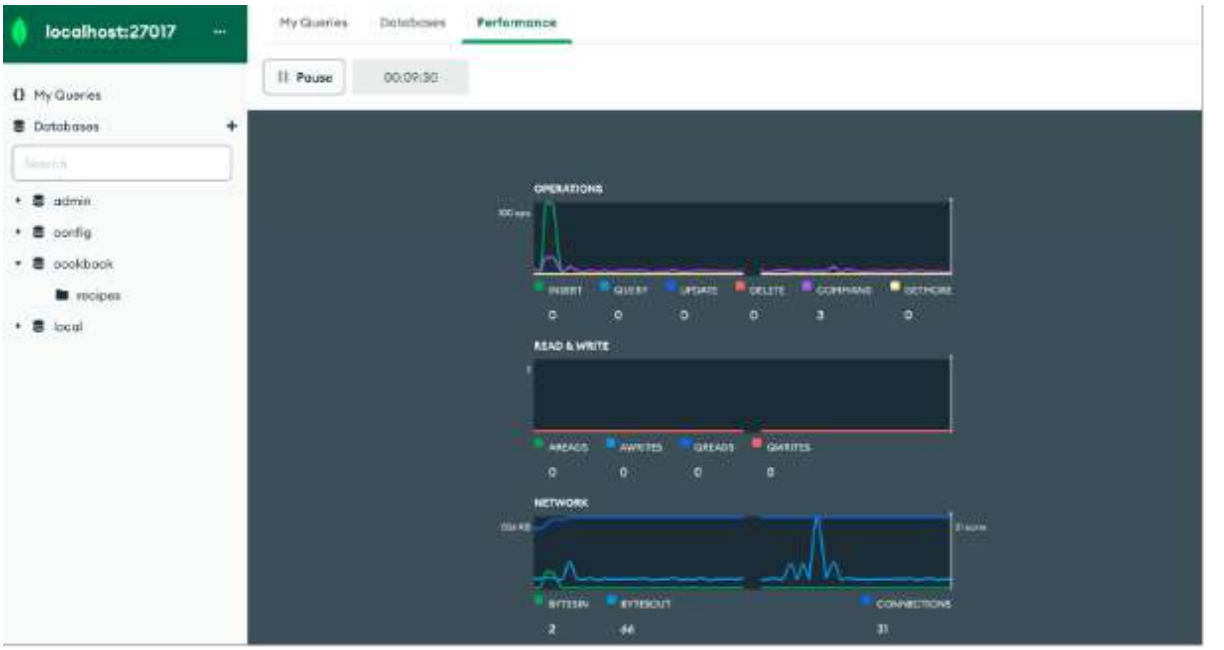


Figure 15.20: *Performance Metrics in Compass*

Conclusion

With what we learned, have you already customized your **`mongosh`** prompt? If not, go ahead and do that, and also try adding some custom functions to your **`.mongoshrc.js`** file as we learned about. Using some simple, and even not so simple Node.JS functions, can make your life a lot easier. We also learned about the Visual Studio Code MongoDB extension, and a whole bunch more about MongoDB Compass' features. In the next chapter, we will learn about the cloud services MongoDB offers, as part of its *MongoDB Atlas* solution.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 16
Cloud Services – MongoDB Atlas

"The best customer service is if the customer doesn't need to call you, doesn't need to talk to you. It just works."

— Jeff Bezos

"I'm sorry, I couldn't find a nicer place to crash-land. Should we try again?"

— Chief Miles O'Brien

Introduction

Beyond making the database itself, MongoDB also has a lot of related cloud services using, and sometimes extending, MongoDB in one way or another, all grouped under the moniker "Atlas". In the next two chapters, we will learn more about Atlas and the tools and services it offers.

Structure

In this chapter, we will discuss the following topics:

- Database Services
- Cloud Tools
- Charts

Objectives

This chapter will only briefly cover a couple of key aspects of the cloud services offered by MongoDB Atlas, as indeed, a whole book in itself could be written about Atlas. That said, after reading this chapter, you should have a solid understanding of what Atlas has to offer and how you can use it to leverage the power of MongoDB even further. Atlas has essentially three main categories of features, two of which we will talk about in this chapter: cloud hosting services and database tools, such as Search and Charts. In the next chapter, we will discuss the third category, Atlas Application Services.

Since each of these sections will be brief, links to find out more about the topic will be provided.

Database Services

At the core of MongoDB Atlas, is its **Database as a Service**, or **DBaaS**, which is a fully managed and hosted version of the MongoDB server. This means that you do not need to install, configure, and manage the database server yourself. Many developers prefer this if they are not comfortable, or do not have the time or resources to maintain their own MongoDB server setup.

Multi-Cloud Database Services

One of the most interesting features of Atlas overall is that it can run *in*, or perhaps more clearly, *on top of*, multiple cloud providers. You can choose to run Atlas with Amazon AWS, Google Cloud, Microsoft Azure, or even a mixture of providers. This allows you to deploy your instances, and application, in 100 different regions, so that you can geolocate closer to you, your users, or spread across multiple data centers for maximum redundancy.

There are three types of instances available, shared clusters, dedicated clusters, and serverless instances, all of which can be run on the different cloud providers. You do not need to login to these cloud services, or even have an account on them; Atlas handles everything for you via your Atlas account.

Clusters

MongoDB Atlas uses the term "cluster" to convey what we generally have referred to as a *Replica set* in this book thus far; they are basically the same thing. If you are using the free tier, which we discussed in *Chapter 3, Getting Started*, Atlas will create a three-member replica set as your cluster. In the following *Figure 16.1*, we have chosen to create a shared cluster, on AWS in Norther Virginia, using the free tier:

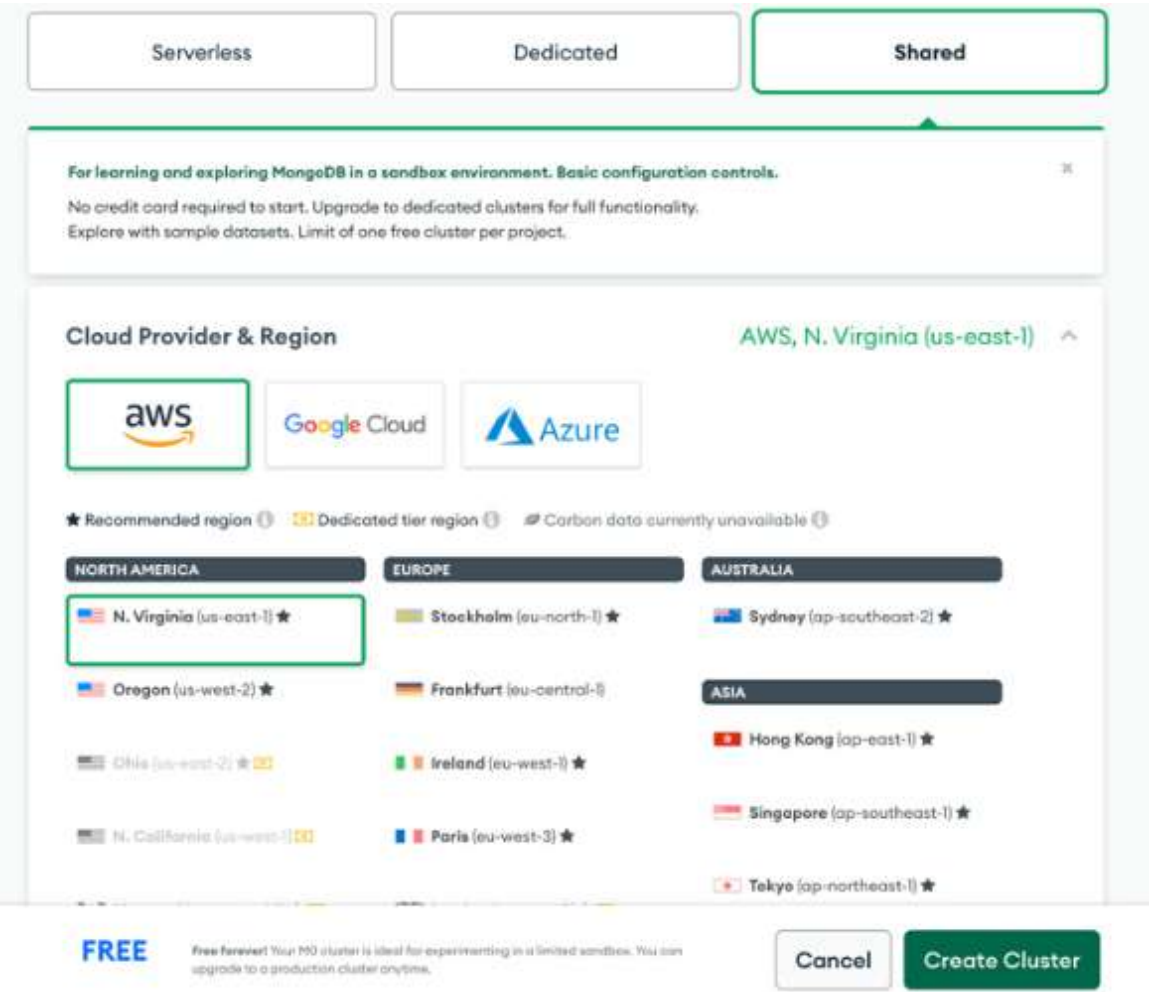


Figure 16.1: *Creating a Free Cluster on MongoDB Atlas*

Once you create your cluster, it should be available from the **Deployments | Database** section of the left navigation, and will look something like *Figure 16.2*:

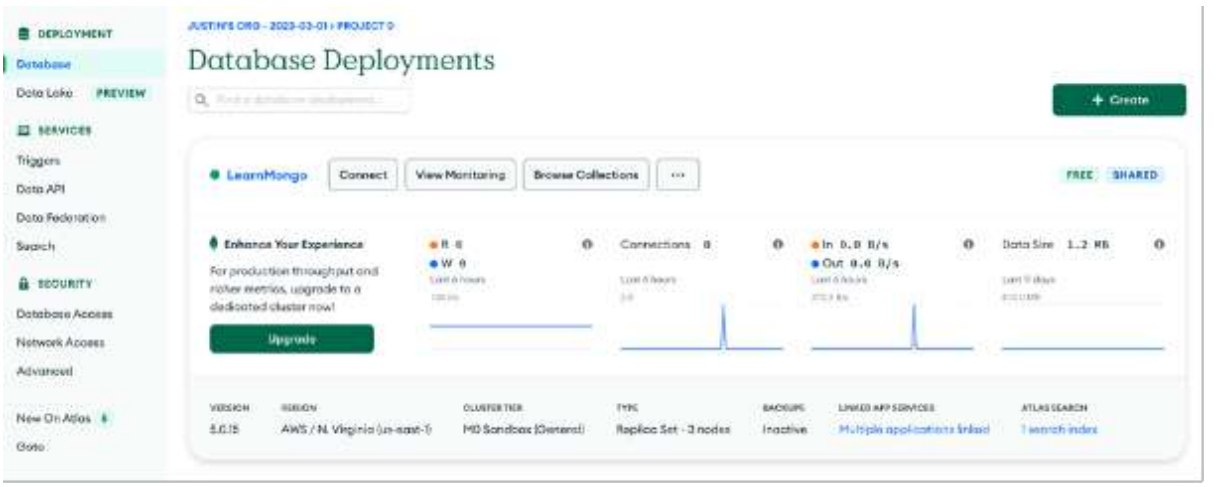


Figure 16.2: *Database Deployments in Atlas*

As seen in *Figure 16.2*, this will show an informational snapshot of your cluster, including some basic monitoring, what version of MongoDB it is running, data sizes, connections, and more. Depending on your cluster type and setup, your screen may look different.

Serverless

A newer option with MongoDB Atlas is *Serverless*, which allows you to deploy a database that scales on demand, and charges by the resources you actually use, instead of a set fee for a cluster, by size. This can allow you to scale up, or down your application on demand and save a lot of headache, and possibly cost, depending on how your application works.

Despite the name, serverless instances are real MongoDB servers, and managing anything related to the server, or scaling it, is generally hidden away and managed for you. *Serverless* instances have most of the same features as any other cluster, with some limitations you can read about here: **https://www.mongodb.com/docs/atlas/reference/serverless-instance-limitations/**

Viewing and Editing Data

From the **Database Deployments** page, that we referenced earlier, you can see a **Browse Collections** button. Press that button to open an interface that looks and works almost the same as MongoDB Compass. From this web based interface, as seen in *Figure 16.3*, you can view and edit your collection's data, indexes, and so on, along with running aggregation and more:

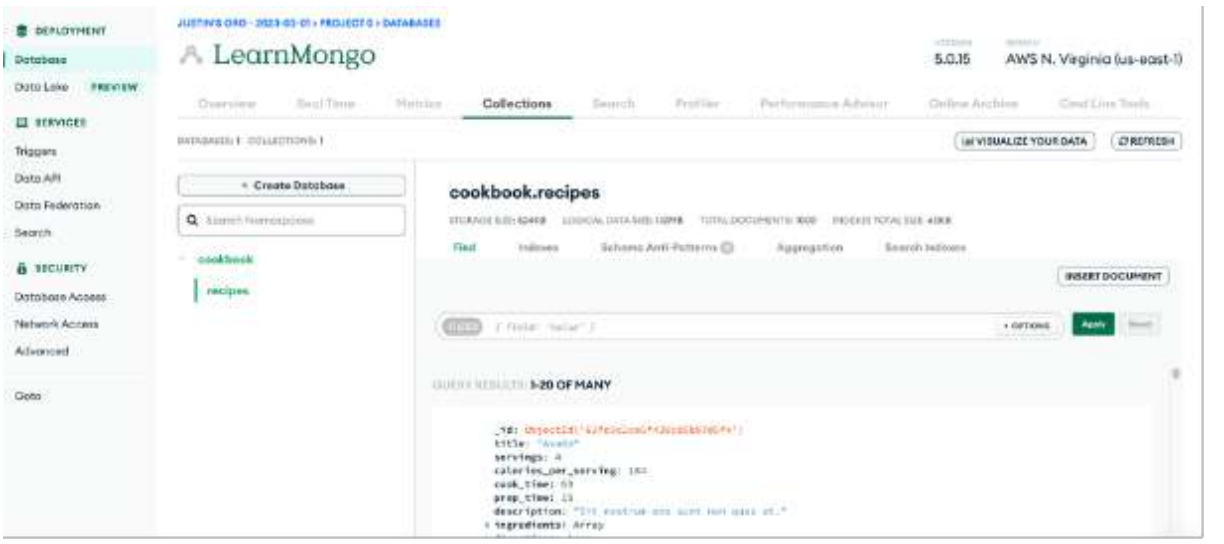


Figure 16.3: *Viewing a Collection in Atlas*

You can use this interface to do most operations you might need. However, to use your cluster from your location machine, or your application, you will need to configure users and network settings, which we will talk about in the next two sections.

Users

By default, there are no users created for your Atlas MongoDB cluster, which means there is no way to connect to your cluster since Atlas *requires* authentication. Atlas will launch a wizard when you first setup your cluster to create and assign users, but you can access the **Security | Database Access** section from the left navigation at any time to adjust your users as seen in *Figure 16.4*:

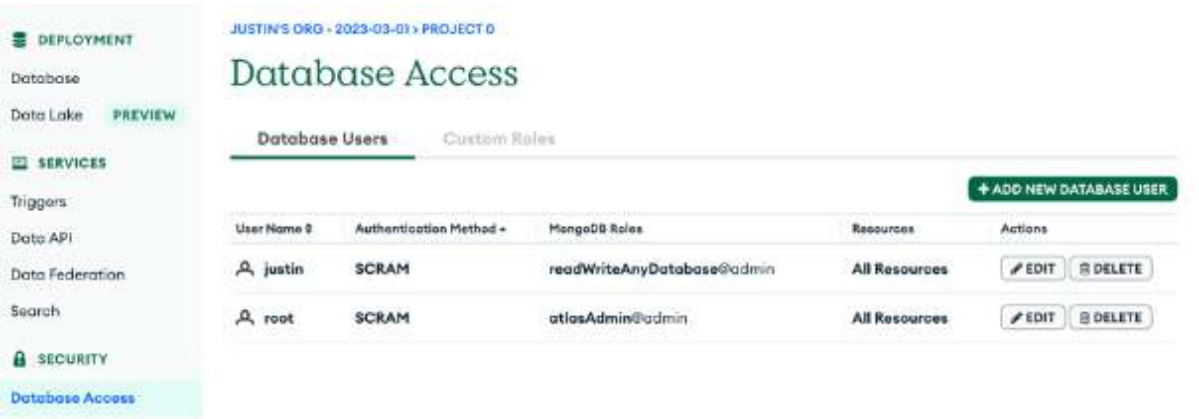


Figure 16.4: *Users in Atlas*

Also, note in *Figure 16.4*, next to the **Database Users** tab, is a **Custom Roles** tab. This provides a graphical interface for the creation of MongoDB Custom Roles, which we previously discussed in *Chapter 13, Being Proactive – Security and Backups*. You can use this interface to create any sort of custom roles you need for your application or database in general.

Network Access

Much like user accounts, network access to your Atlas cluster is not setup by default, and so, there is no way to connect remotely to the database. The most simple and

common way to enable this, is to go to the **Security** section of the left navigation and use the **ADD IP ADDRESS** button. This will launch a model like the one in *Figure 16.5*:

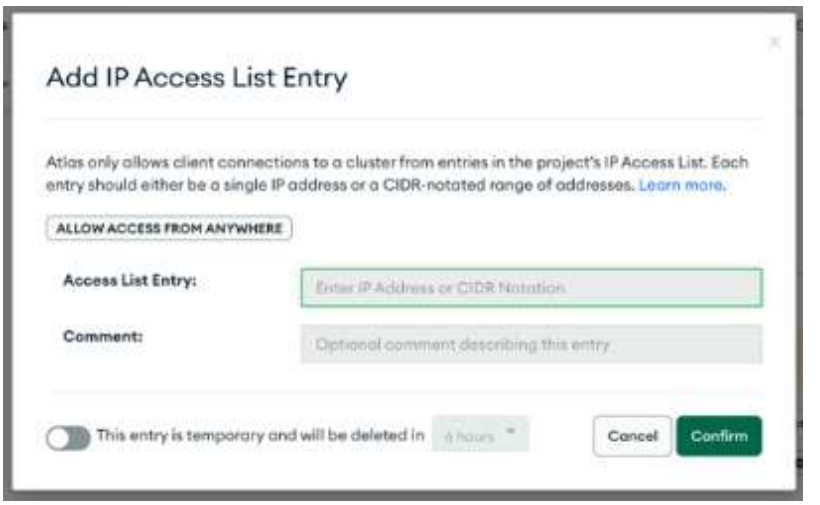


Figure 16.5: *Add IP Access In Atlas*

You can use this to add single IP addresses, or a range of addresses; for example, for a corporate network or datacenter, as well as allowing access from any IP. However, it is generally recommended not to allow access from anywhere, since this opens up connections to the entire internet.

Once you add IP addresses you can view, modify and delete them from the **IP Access List** tab as seen in *Figure 16.6*, as well as investigate some of the other connection options available, like a **Private Endpoint**:

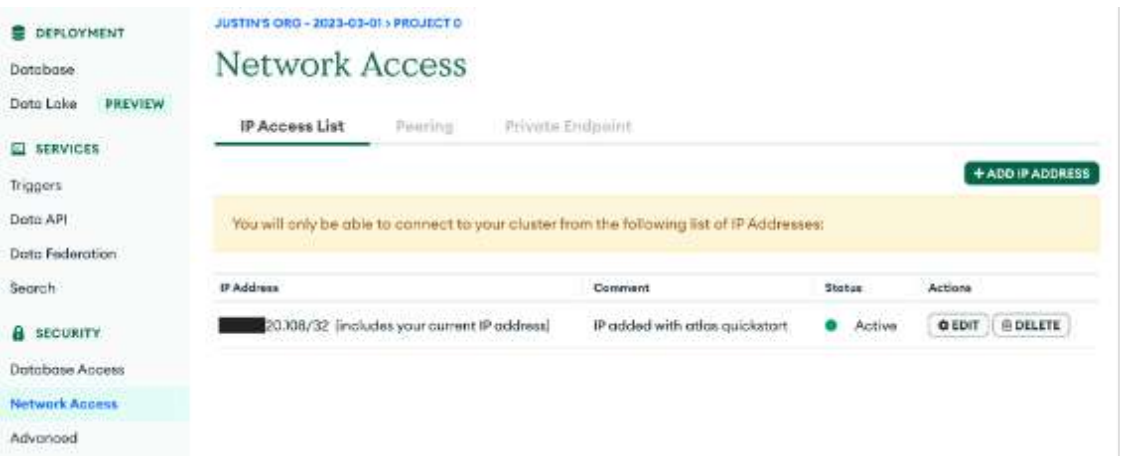


Figure 16.6: *IP Access List in Atlas*

There is also an **Advanced** section, which has a number of extra configuration options such as LDAP Authentication, security keys, and more. These settings correlate to the authentication options we discussed in *Chapter 13, Being Proactive – Security and Backups*.

Connecting to Atlas Cluster

Now that you have both your networking and user account(s) setup, you will be able to connect to your Atlas cluster via **`mongosh`**, MongoDB Compass, or your programming language. The connection string for connecting to an Atlas cluster is a little more simplified than connecting to the replica set/cluster we created in *Chapter 12, Seamless Scaling – Replication and Sharding*, as Atlas uses the *DNS Seed List Connection Format*.

This will use DNS to figure out each host in the replica set/cluster instead of you needing to assign them in the connection string. This format looks something like this:

```
mongodb+srv://<username>:<password>@<cluster>.mongodb.net/test
```

After replacing the placeholders for your username, password and cluster name, this will connect to the **`test`** database on your cluster. You can use this connection string for **`mongosh`**, or in MongoDB Compass, as we are, in *Figure 16.7*:

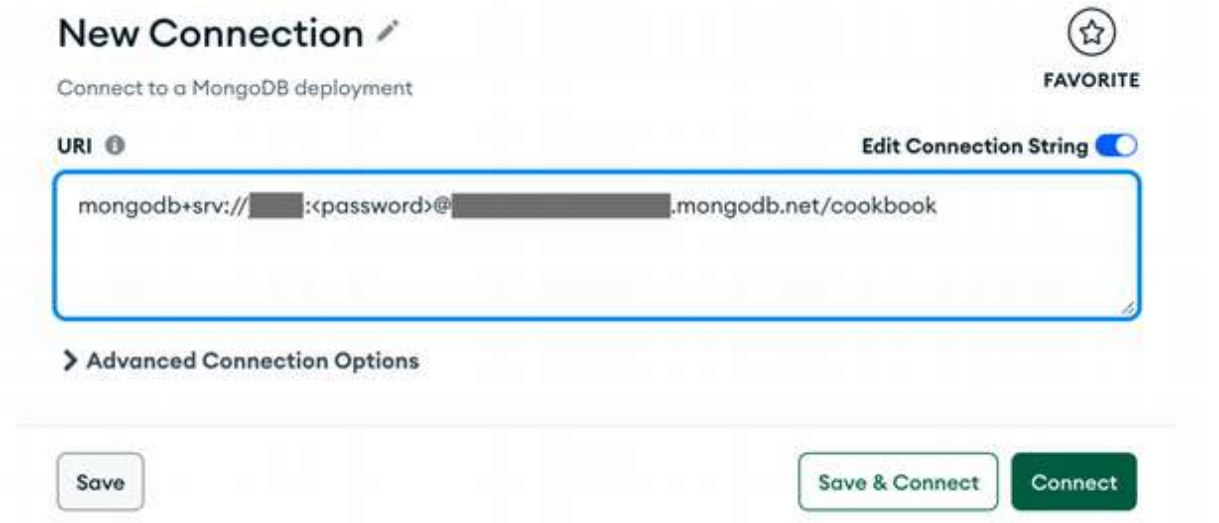


Figure 16.7: *Connecting to Atlas via MongoDB Compass*

You can read more about this format vs the standard format in the documentation: **https://www.mongodb.com/docs/manual/reference/connection-string/**

Data API

Another way to connect to your cluster is by using the Atlas *Data API* which allows you to interact with data in Atlas via simple HTTPS requests. You can insert, modify and delete data, along with performing other queries via standard HTTPS requests, all without the need to install any drivers, or clients.

This can be especially useful for use cases where your "thing", such as a mobile application, Internet-Of-Things device, a CI/CD pipeline, a command line script, and so on, needs to make a simple insert or find query to your MongoDB database.

As an example, you can use the **`curl`** command to make a call to the Data API as follows:

```
$ curl --request POST 'https://data.mongodb-api.com/app/<App_ID>/
endpoint/data/v1/action/insertOne' \
  --header 'Content-Type: application/json' \
  --header 'apiKey: <Data API Key>' \
  --data-raw '{
      "dataSource": "<cluster name>",
      "database": "website",
      "collection": "cookbook",
      "document": {
        "title": "Apple Pie",
        "description": "Grandma's Apple Pie"
      }
  }'
```

Breaking down this command a little, there are basically two parts: the Endpoint URL like the following, which has the action you wish to perform:

https://data.mongodb-api.com/app/<App_ID>/endpoint/data/v1/action/insertOne

Of course, you would replace **`<App_ID>`** with your app id, but more importantly, notice the **`action/insertOne`** at the end, which could be **`actions/findOne`** or another action, as well. This lets Atlas know which action to perform.

The URL is followed by headers which contain your **`apiKey`**, as well as the configuration of the request you are making:

```
--data-raw '{
      "dataSource": "myClusterName",
      "database": "website",
      "collection": "cookbook",
      "document": {
        "title": "Apple Pie",
        "description": "Grandma's Apple Pie"
```

```
    }
}'
```

This will send the JSON object to the Data API, which has the cluster name as well as the database, collection and, in this case, the contents of the document we want to insert.

The example shown is quite basic; you can also use the "extended" version of JSON in your request, which if you recall, can handle more complex datatypes such as Dates, and ObjectIDs. To support this, you will need to add an additional header.

Again, if using **`curl`**, it would look something like this:

```
-H 'Accept: application/ejson'
```

Now you could send *Extended JSON*, like an ObjectId, or Date, as part of your request in a format as follows:

```
"_id": { "$oid": "640c29fae9a647a214336232" }
```

On an application scale, the Data API probably is not the best option, since the overhead of the request is likely bigger than making a request via a driver, in whatever programming language you are using. However, for more niche uses, the Data API may be a perfect solution to allow your "thing" to interact with your Atlas cluster.

For more about the Data API, see: **https://www.mongodb.com/docs/atlas/api/data-api/**

Backups

If you are using the free **`M0`** cluster from Atlas, backup services are not included. However, you can use **`mongodump`**, **`mongorestore`**, **`mongoexport`**, and **`mongoimport`** just like with any other MongoDB instance. For a review of using those tools for backing up and restoring MongoDB, refer to *Chapter 13, Being Proactive – Security and Backups*.

If you have a paid cluster on MongoDB Atlas, there are a couple of options, depending on what exact tier you are using. Generally speaking, for the lower end shared clusters such as **`M2`** and **`M5`**, backups are automatically enabled along with daily snapshots.

You can find more about shared cluster backups here:

https://www.mongodb.com/docs/atlas/backup/cloud-backup/shared-cluster-backup/

For higher tiers, such as **`M10`** and up, MongoDB Atlas will use the snapshot functionality of the cloud provider you chose when setting up your cluster, such as AWS, Google GCP, or Microsoft Azure. The same goes for the *Serverless* option, and backups are automatically enabled and cannot be turned off. Atlas will take incremental snapshots of the data in your serverless instance in six-hour intervals, keeping the two most recent snapshots.

Cloud Tools

In the next few sections, we will discuss different tools which you can leverage in Atlas to make working with the data in your cluster even more seamless, such as, *Search*, *Triggers*, *Device Sync* and *Data Lake*.

Search

MongoDB Atlas *Search* allows you to create more advanced search capabilities for your application, including case insensitive and full-text search, along with more complex search scenarios via a simplified API. Atlas takes care of the more complicated aspects of analyzing your data to optimize it for search, and then provides a simple way to query your data via an aggregation pipeline query.

You can find this feature under the **`Search`** tab for your cluster, as shown in *Figure 16.8*:

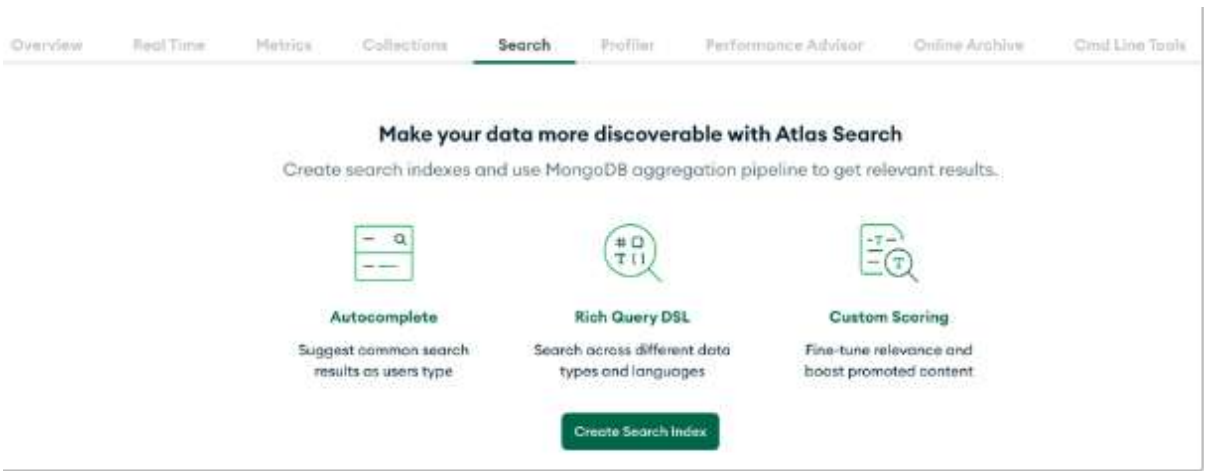


Figure 16.8: *Search in Atlas*

When you create your first Search index, which is available on the free tier, you will see the default index configurations as seen in *Figure 16.9*:

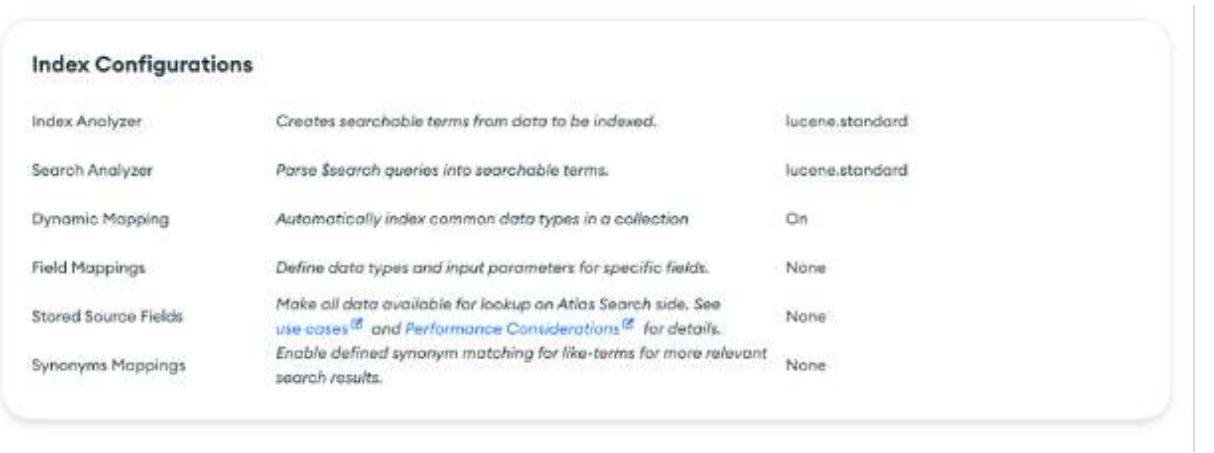


Figure 16.9: *Atlas Search Configuration*

The index and search "analyzer" referenced is *Apache Lucene*, an open-source search engine software. You can use either a visual editor to configure each setting, or a JSON file. Once created, you can use your search via an aggregation pipeline query.

Simply connect to your Atlas cluster, switch to the appropriate database, and then run an aggregation query, as we learned about in *Chapter 8, The MongoDB Aggregation Framework,* but this time using the **`$search`** operator:

```
> db.recipes.aggregate([
  {
    $search: {
      index: "myIndex",
      text: {
        query: "chicken",
        path: { wildcard: "*" }
      }
    }
  }
])
```

This will work just like any other aggregation query and return a cursor that you can use in **`mongosh`**, MongoDB Compass, or any supported programming language.

There are some helpful "quick start" tutorials available to help you create different search tools and interfaces for your data, as seen in *Figure 16.10*:

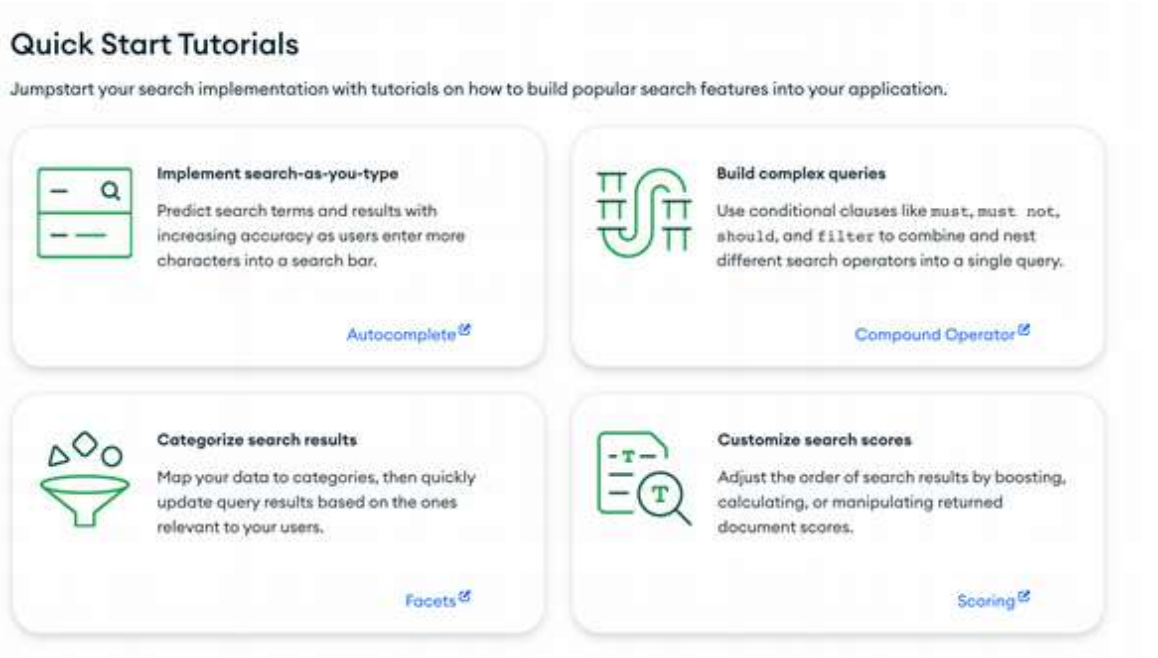


Figure 16.10: *MongoDB Atlas Search Quick Start*

Along with the examples in *Figure 16.10*, Atlas Search supports fuzzy search, synonyms, highlighting and more. Since all of these connect directly to your cluster's data you can update or add more search capability in a quick, easy and seamless way.

If Atlas *Search* interests you, find out more here: **https://www.mongodb.com/atlas/search**

Triggers

MongoDB Atlas *Triggers* are functions that can be executed in response to specific events, such as a document being inserted, updated, or deleted in your Atlas cluster. They can also be run on a schedule, much like a **`cron`** job.

A trigger's function can be written in JavaScript or can reference a function in your application code. Triggers can be really helpful if you need to make sure to perform a particular action either on a regular schedule, or for example, when you insert or delete certain documents, or even using a **`$match`** operation.

You can not only program your function to perform actions inside your own cluster's data, but you can also trigger code that performs actions in your application, or even an action on completely separate services, such as sending a text message via the Twilio API, for example.

To read more about the massive number of things you can do with Atlas Triggers, make sure to follow this link: **https://www.mongodb.com/docs/atlas/app-services/triggers/**

Device Sync

MongoDB Atlas *Device Sync* is another fully managed cloud service which allows you to synchronize data between your MongoDB Atlas cluster and mobile, or IoT devices, in near real-time. By using a "bidirectional sync" protocol, it ensures that data changes made on a device are automatically synced to your Atlas cluster, and vice versa.

Device Sync automatically resolves conflicts that arise when changes are made to the same document on different devices, and can also work in offline mode, allowing devices to continue syncing data, even when they are not connected to the network. Atlas Device Sync allows you to build applications that work seamlessly across different devices and platforms, without having to worry about managing data synchronization manually, all the while using end-to-end encryption to ensure that data is securely transmitted between devices and the Atlas cluster.

For more on this powerful feature, see the following: **https://www.mongodb.com/docs/atlas/app-services/sync/**

Data Lake

A more and more important need for organizations is the ability to easily analyze their data, without changing the structure of the data that their application, or applications need. Atlas Data Lake is a fully managed cloud storage solution optimized for analytical queries.

A *Data Lake* differs from a *Data Warehouse*, which stores highly structured information from various sources where a data lake stores data from difference sources in its original, raw format.

Atlas Data Lake will automatically take snapshots of your Atlas cluster and then optimize it by reformatting, partitioning, and indexing the data in an isolated environment. Thus, your production database is unaffected. This allows other roles such as business analysts and data scientists free, to run complicated queries without any impact on your application's database.

You can find more about Atlas Data Lake here: **https://www.mongodb.com/atlas/data-lake.**

Atlas CLI

Lastly, the *Atlas CLI* is a command line interface built specifically for MongoDB Atlas. You can interact with your Atlas database instances and other tools such as Atlas Search or Data Lake from your command line using the **`atlas`** command.

See more at the link below, if this option interests you:

https://www.mongodb.com/docs/atlas/cli/stable/install-atlas-cli/

Charts

MongoDB Charts let you easily hook up to the source of your Atlas cluster and build dynamic charts and dashboards. There are many chart types available, along with tables, word clouds, and even geo-charts which can be powered by geo data stored in MongoDB. You can create individual charts and then share or embed them in your website or combine multiple charts and types to create a dashboard. In the following *Figure 16.11*, you can see an example of one of these dashboards:

Figure 16.11: *Atlas Dashboard*

In the next few sections, we will delve into a brief overview of how to create, assign permissions to, and share these charts and dashboards.

Create Charts

The first step to creating charts is to setup your Data Source. Atlas makes this simple by giving you an interface to connect directly to a collection, in your Atlas cluster, as seen in *Figure 16.12*:

Figure 16.12: *Atlas Charts Sources*

After you select a data source, Atlas will automatically "sample" your collection's documents to find fields that look like they could be used in charts. Using our recipe documents, Atlas automatically came up with the fields in *Figure 16.13*:

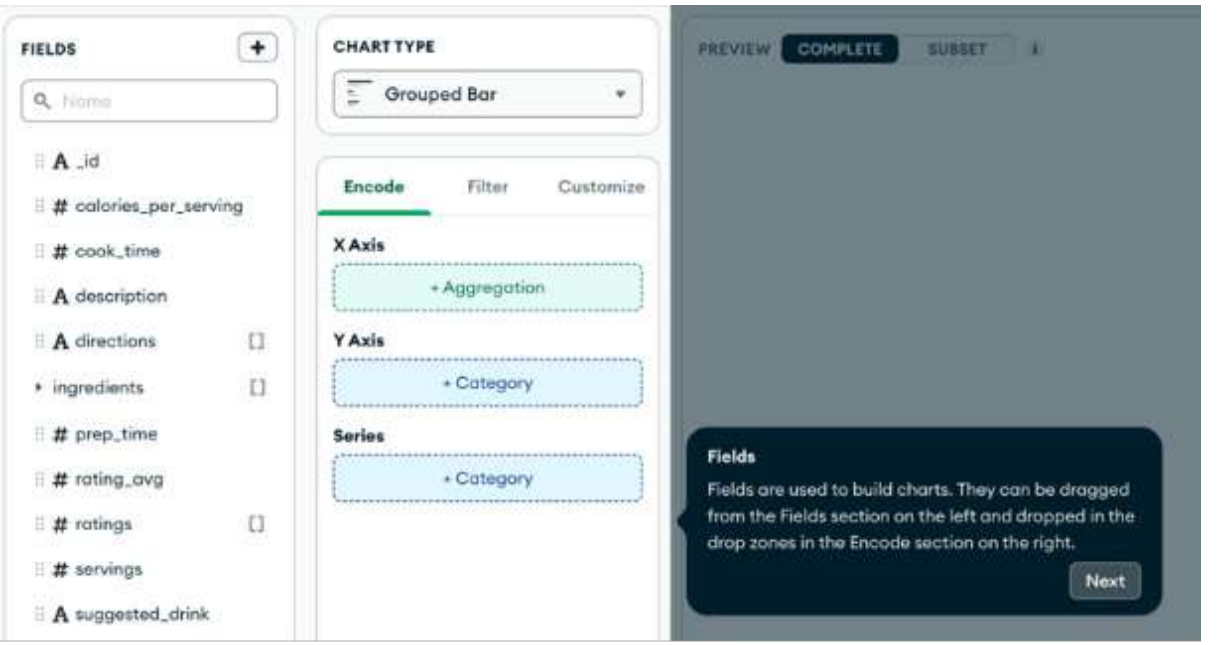


Figure 16.13: *Atlas Charts Fields*

Some fields are off the screen in *Figure 16.13*. However, you can see that Charts was able to determine some fields, such as **`cook_time`** and **`prep_time`**, are a number,

shown here with a **#** and others were generally strings like **description**, shown here with an **A**.

You can pick a **Chart Type** and then drag a field over to make a simple chart like *Figure 16.14* which is charting the **prep_time** of recipes:

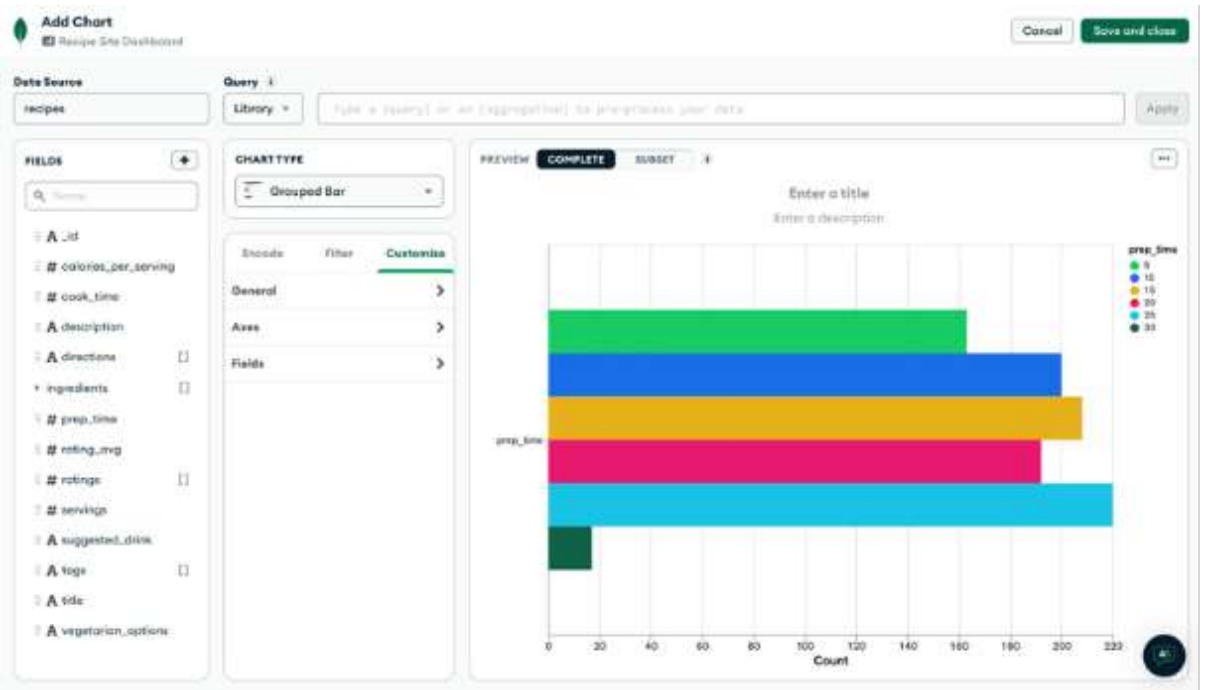


Figure 16.14: *Bar Chart in Atlas Charts*

You can adjust the chart's title, field labels, limit results, do different types of grouping, and so on, all within the Charts interface. You can also add your own fields to chart on by combining the data from multiple fields, or by running a calculation on a fields data using a **Calculated Field**. To find more about this feature, read more on this link: **https://www.mongodb.com/docs/charts/calculated-fields/**

If all those options do not get you the exact data you need, you can perform full MongoDB queries, including **Aggregation Pipelines** from within the Charts interface as well, as seen in *Figure 16.15*:

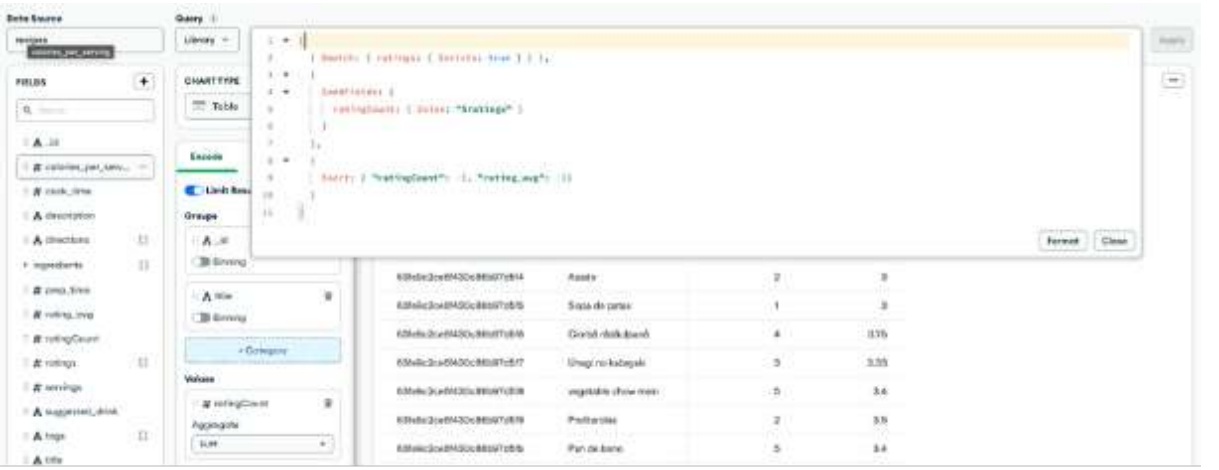


Figure 16.15: *Aggregation Query in Atlas Charts*

In this example, we are running an aggregation to add a field which has a **`ratingCount`** field for each recipe. This field will be automatically detected by Charts and added to the fields list on the left. We can then use this field, along with a calculated field, to create a table chart of the most popular recipes based off both number of ratings and the average rating.

Interestingly, the charts themselves use their own pipelines, which you can see by pressing the **...** at the top right of the chart, and choosing **View Aggregation Pipeline**, as see in *Figure 16.16*:

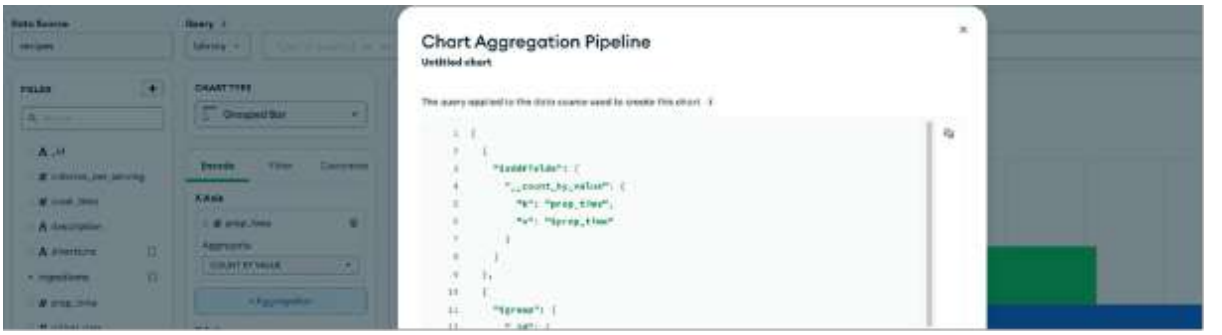


Figure 16.16: *View Chart Pipeline*

We will keep discussing a couple more aspects of Charts, but for more, see: **https://www.mongodb.com/docs/charts/**

Sharing and Embedding

By using the **Share** button at the top right of your Dashboard, there are many options and settings around permissions for sharing your Dashboards and Charts, on the project, organization and public levels, as seen in *Figure 16.17*:

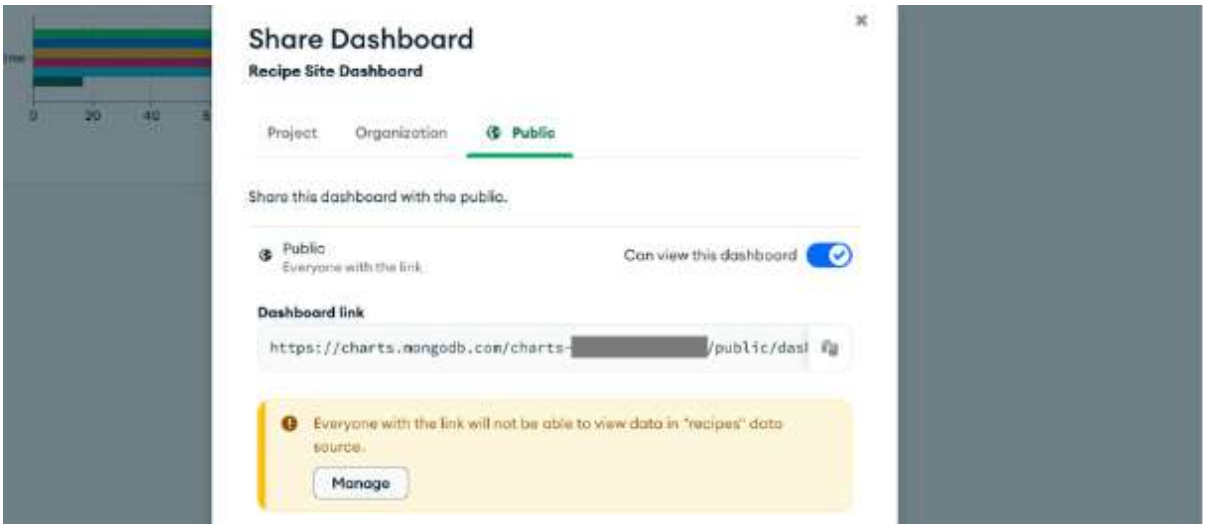


Figure 16.17: *Share Atlas Chart Dashboard*

There is also the ability to embed the Dashboard or Chart via either an iFrame or via the JavaScript SDK, as seen in *Figure 16.18*:

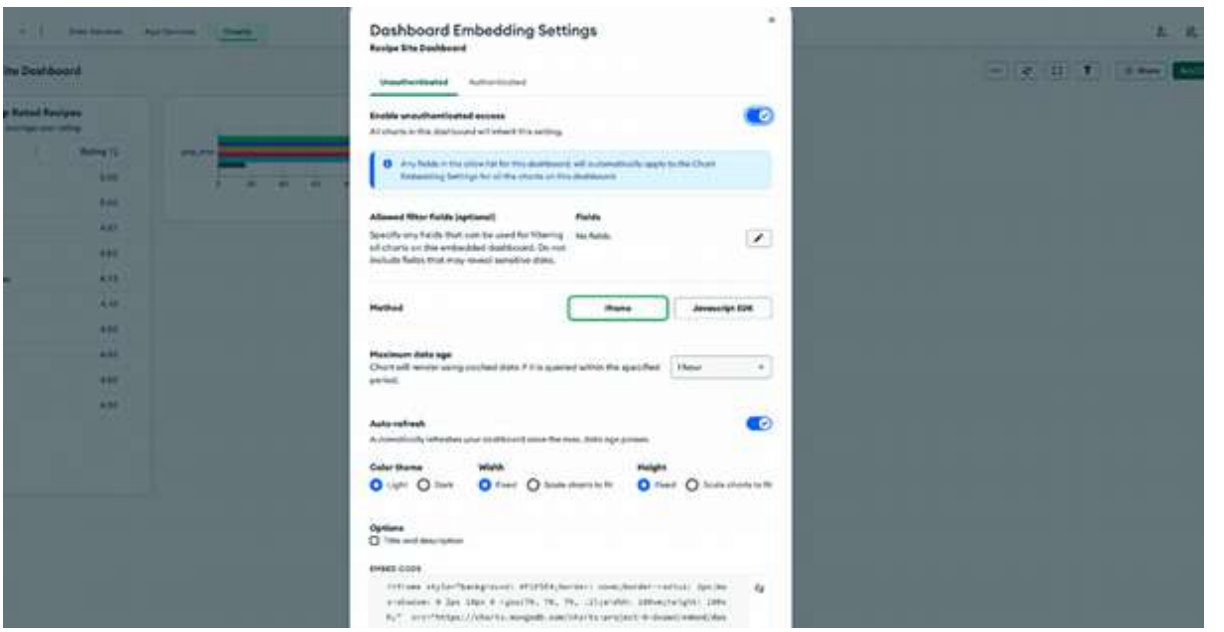


Figure 16.18: *Embed Atlas Chart or Dashboard*

There are lots more options around permissions, authentication, sizes, themes, and so on. You can find a lot more information about permissions at the following link:

https://dochub.mongodb.org/core/charts-dashboard-permissions

Conclusion

In many ways, this chapter was a bit of a "whirlwind tour" of the services offered by MongoDB Atlas, but you should now have a good idea of a lot of the key services it offers, and with the aid of the provided links, you can find out more. By taking on a lot of the complication and worry of hosting a database with automatic setup of replica sets, Serverless scaling, Search, Data API, Security, Triggers and Charts, MongoDB Atlas may be a great fit for your application or company.

In the next chapter, we will delve into the last leg of the MongoDB Atlas stool, so to speak, the *Application Services* which provides tools to develop applications within, and leveraging Atlas itself.

Chapter 17

MongoDB Atlas – Application Services

> **"In the word question, there is a beautiful word - quest. I love that word."**
>
> — *Elie Wiesel*

> **"True perfection is a bold quest to seek. Only the willing and true of heart will seek the betterment of many."**
>
> — *Socrates*

Introduction

In the previous chapter, we introduced MongoDB Atlas in general, and its Cloud Services. These services have been known by a couple of different names over time, such as MongoDB Stitch, Realm, and others, as the product was built out. In this chapter, we will delve deeper into the overall *Application Services* that Atlas offers, by making our own simple "Serverless" app, called MongoQuest using Atlas and the popular JavaScript Framework, React.

Structure

We will discuss the following topics:

- Atlas Database and App

- React App
- Coding the App

Objectives

We will be building a React app using Atlas App Services, which allows us to build on top of MongoDB, without actually having to run or maintain our own Replica Set, or server. Rather, we will rely on a shared free MongoDB Cluster, Atlas Functions, the Realm SDK for Web and the Atlas' Data API.

You can follow along using the code samples provided. There is also a git repository with the entire source of the app which we reference later in this chapter. Even if you do not know React at all, the concepts apply to many other programming languages and Frameworks, and therefore, by the end of this chapter, you should have a good idea of how to use them for your preferred language and setup.

Atlas Database and App

To build our app, we will use React and the Realm SDK for Web, but there are many other SDKs available for other languages such as Java, Kotlin, .NET, Node.js, Swift, Flutter, React Native and so on. If React is not relevant for you, feel free to skip past some of the code, but the fundamentals will be the same.

You can find out more about the various SDKs here:

https://www.mongodb.com/docs/realm/sdk/

Question Example

Our app will be a "quest" to answer, or give a response to, real interview-like questions about MongoDB, and hence the name *MongoQuest*. We will discuss importing a collection of these questions next, for how we can take a look at an example document for a question. As you can see in the following example document, each question will have a **`level`**, representing its general difficulty, as well as the text of the question, in the **`question`** field, and an object, or subdocument, containing a **`response`**.

```
{
  "_id": {
    "$oid": "64192b927b7fbb7150674782"
  },
  "level": 2,
  "question": "Can you explain the concept of indexes?",
```

```
    "response": {
      "short": "An index is a data structure that improves the speed of data retrieval operations on a collection. When a query is executed, MongoDB can use an index to quickly locate the documents that match the query criteria, rather than scanning through the entire collection.",
      "reference": {
        "chapter": 9,
        "section": "Indexing collections"
      }
    }
  }
```

The **`response`** is made up a field called **`short`**, which is a short, possible response, as well as another object called the **`reference`**, which contains a **`chapter`** and optionally a **`section`**. These correspond to where in this book you can find out more about the question and response.

We will use these fields to construct our app's screens.

Importing Data

If you are following along, you can import the question documents included in the book's git repository. You can find them in the **`questions.json`** file in the corresponding folder for this chapter. Insert the question data by whatever means you prefer; we are connecting to a free Atlas cluster via *MongoDB Compass* and using its import interface, via the **ADD DATA** button, as seen in *Figure 17.1*:

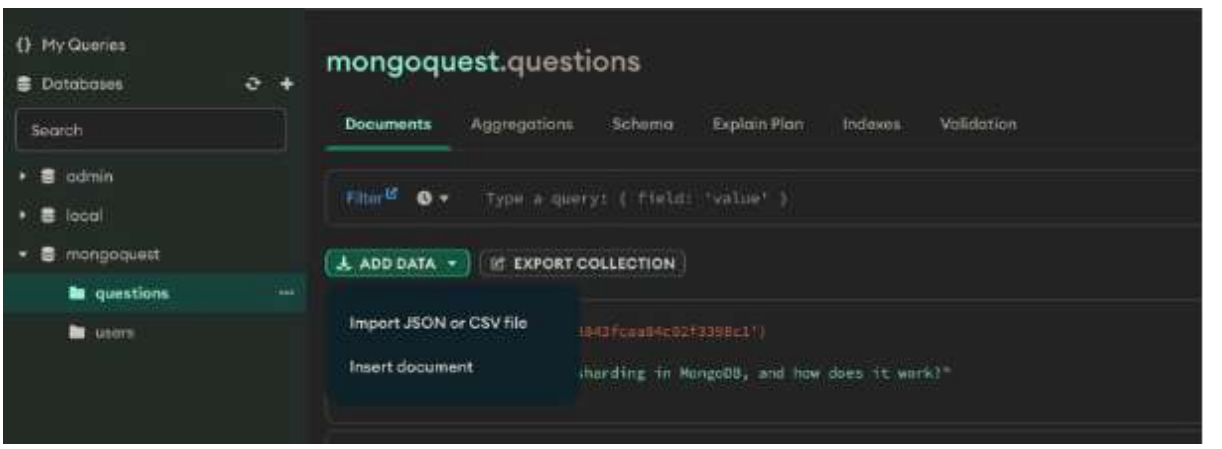


Figure 17.1: *Importing question documents via MongoDB Compass*

After the question data has been imported, if using Atlas, you should be able to see it by navigating to your database in Atlas and clicking on **Browse Collections**, as shown in *Figure 17.2*:

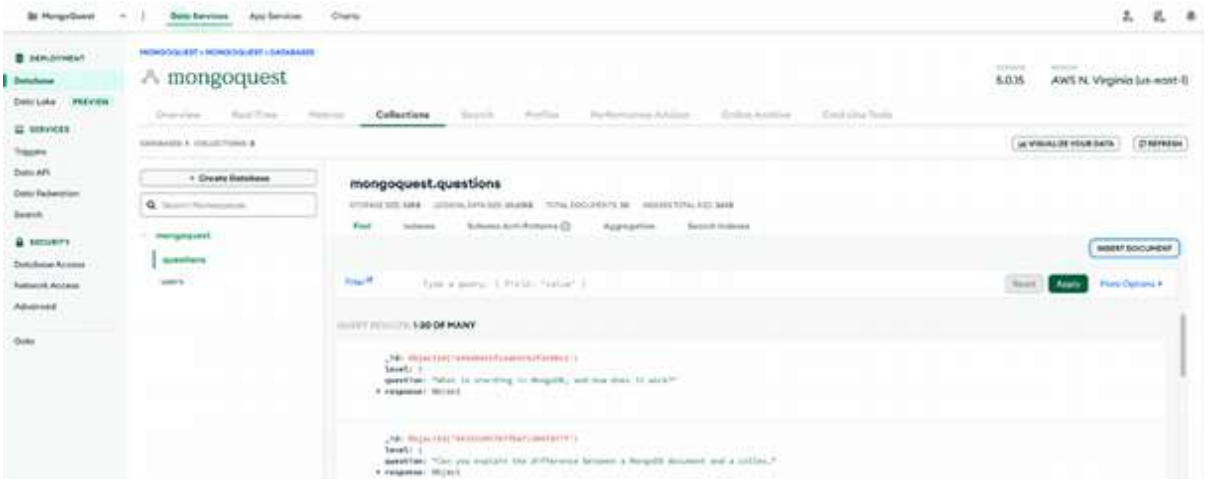


Figure 17.2: *Viewing imported question documents inside MongoDB Atlas*

Once you confirm your data is imported, you can move on to creating your Atlas app.

Create Atlas App

Navigate from your Database in the **Data Services** tab, to the **App Services** tab and press the **Create a New App** button to start the creation of your Atlas app, as seen in *Figure 17.3*:

Figure 17.3: *Creating a new App*

Your experience may vary at this point, so follow either the wizard or modals to setup a new App Service, and make sure to link your data source. It will look something like *Figure 17.4*:

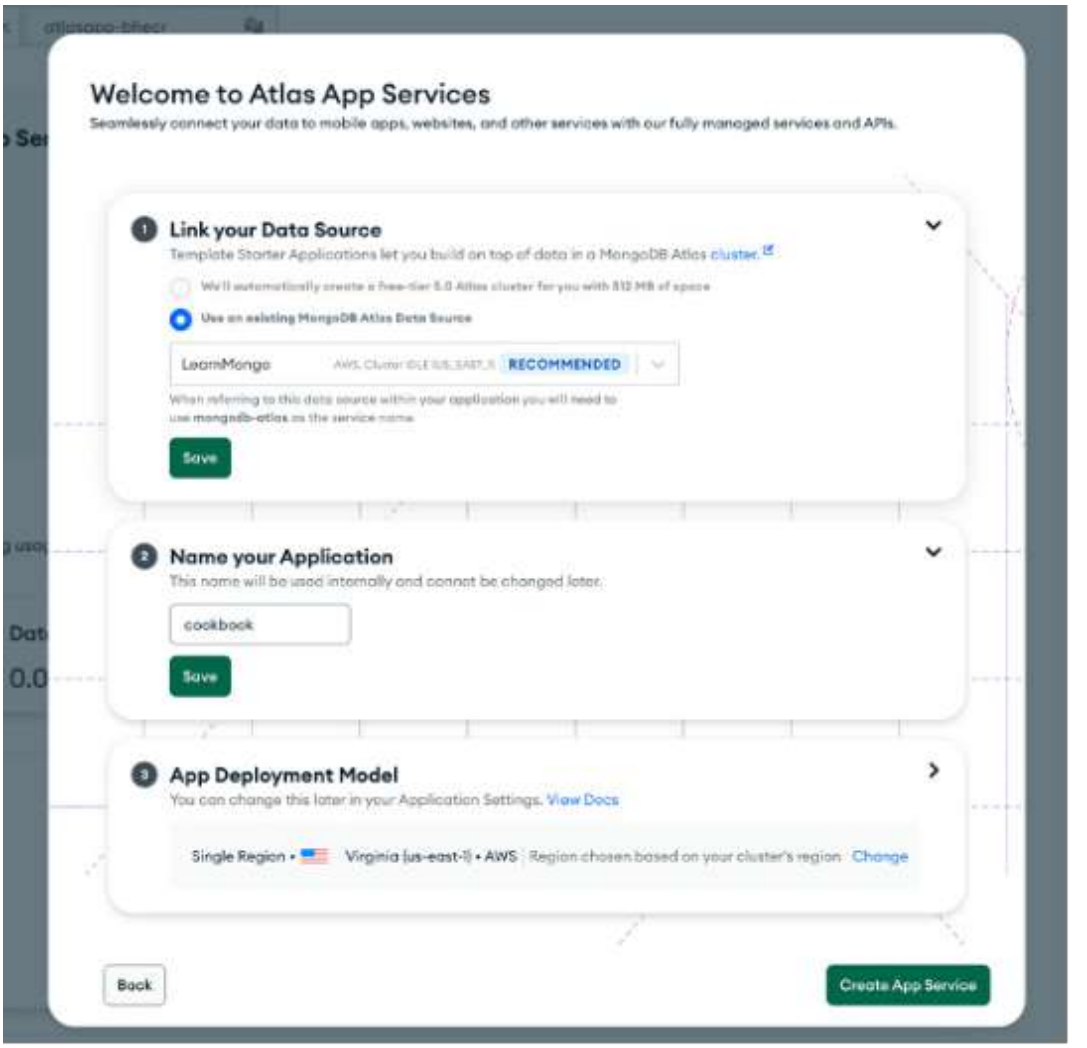


Figure 17.4: *Atlas new App wizard*

You should be able to generally follow the defaults. The important part is to make sure your app is linked to the right data source, which in this case, is the Atlas cluster we imported our data into. Along the way in this process here, and elsewhere too, you will be prompted to apply and publish your changes. Feel free to make any changes right away, we do not have a live application yet.

Lastly, when the app creation process is complete, make sure to take note of your **App ID**, at the top near your app's name; you will need this later. *Figure 17.5* shows the location of the **App ID**.

Figure 17.5: *Location of your App ID*

Database User Access

An important next step is to configure user access, meaning **Roles and Permissions**, to your Database and Collections. You can follow the wizard in Atlas or manually via the **DATA ACCESS | Rules** page. We will want to make sure that anyone can **read** the data out of the questions collection, but not **write**. Once set, the permissions on your collection should look something like *Figure 17.6*:

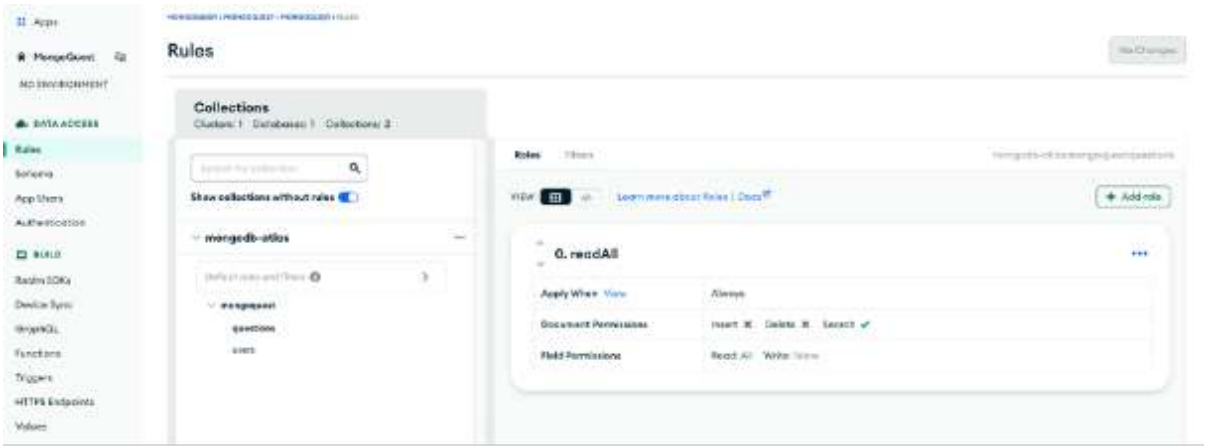


Figure 17.6: *Permissions on your collection*

Make sure to apply your changes and complete any deployments.

Anonymous Realm Users

In order to connect to our collection from our React app, we will need to have some sort of Authentication. Atlas offers many forms of authentication such as API Keys, Email/Password, JWT, Facebook, Google and so on. For our purposes, we will use Atlas' *Anonymous* authentication method which will generate a unique user id and store it the browsers' local storage.

Note: The Anonymous accounts will expire after 90 days of inactivity, if the user logs out, or if the user clears their browser data.

You can enable Authentication methods from the **DATA ACCESS | Authentication** page. Click **EDIT** next to the method you want to setup. Here, it is **Allow users to log in anonymously**. Once it is enabled, it should show a green **On** text and look like the following *Figure 17.7*:

Figure 17.7: *App Authentication Settings*

You can read more about Atlas user authentication works here:

https://www.mongodb.com/docs/atlas/app-services/users/

For the purposes of our app, anonymous users are all we need, since this is not an app which a user will likely use for more than a few weeks at most, and does not need to have user data maintained long term.

User Create Function

By default, *Anonymous* authentication only creates and stores a unique id and no other metadata. However, for our application, we want to allow the user add notes on questions and mark them as answered. To do this, we will need to add the option for *Custom User Data*. You can read more about *Custom User Data* here:

https://www.mongodb.com/docs/atlas/app-services/users/custom-metadata/

Additionally, we will want to setup a **User Creation Function**. We can setup both of these within the **App Users** page. Browse to the **User Settings** tab and expand open the **CUSTOM USER DATA** area. Here, we can set which cluster, database and collection we

would like to use to store this data. We have used a collection called **`users`** and a **User ID Field** on the document called **`user_id`**, as seen in *Figure 17.8*:

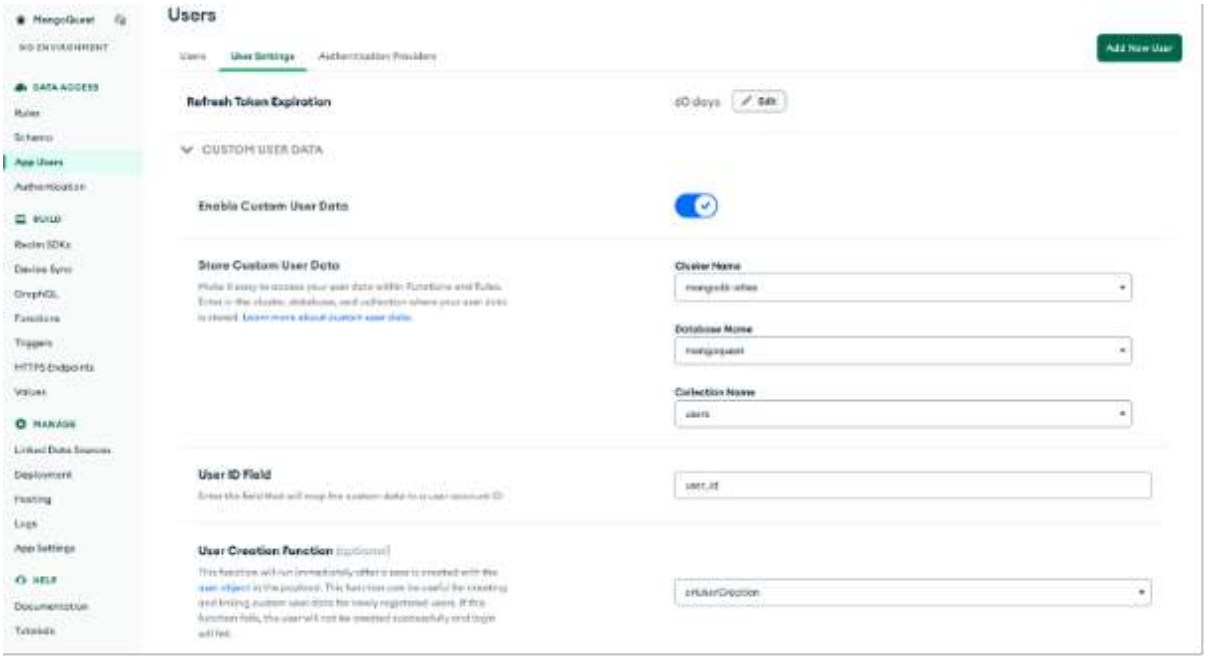


Figure 17.8: *App Users settings page*

Additionally, in *Figure 17.9*, you can see a **User Creation Function** has been assigned at the bottom. The Atlas interface will allow you to pick an existing function, or create one as seen in *Figure 17.9*:

Figure 17.9: *Setting a User Creation Function*

We have linked this to a function, which we created, called **`onUserCreation`** that looks like this:

```
1   exports = (user) => {
2     // use collection that Custom User Data is configured on
3     const collection = context.services.get("mongodb-atlas").
    db("mongoquest").collection("users");
```

```
4
5       const defaults = {
6         name: '',
7         questions: {
8           answered: [],
9           saved: [],
10        }
11      };
12
13      // insert custom data into collection, us-
    ing the user id field that Custom User Data is configured on
14      const doc = collection.insertOne({ user_id: user.id, ...de-
    faults });
15    };
```

This will run automatically after Atlas creates a user, and a corresponding document in the **`users`** collection, which you can see on line **14**. We have also added some fields that we will use in our app, such as the **`questions`** object with **`answered`** and **`saved`** arrays.

When you create your function, under the function's **Settings**, make sure the function is *not* set as **Private**, or you will be unable to use it within the client application. See *Figure 17.10*:

Figure 17.10: *Making sure function is not set as Private*

We also need to set permissions for the **`users`** collection, which we can on the **DATA ACCESS | Rules** page, as seen in *Figure 17.11*:

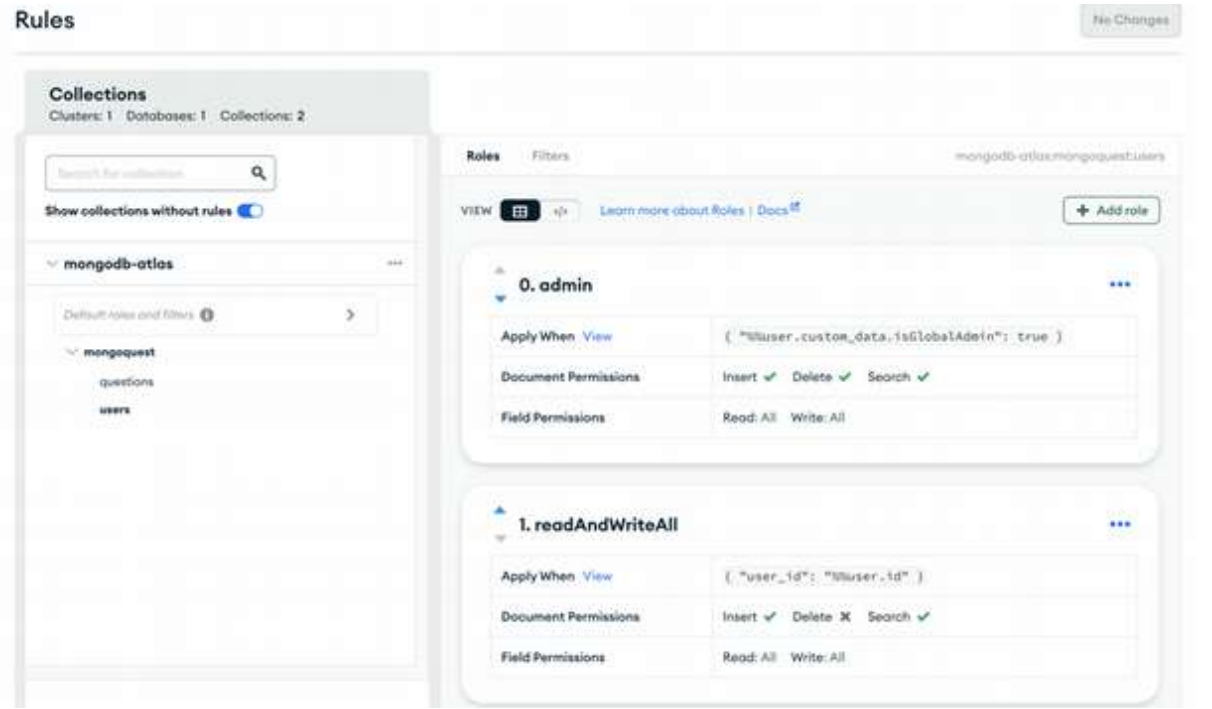


Figure 17.11: *Users collection permissions*

On this page, we will set the *Apply When* rules to only allow **`search`** and **`insert`** for the user id of the logged in user, as seen in *Figure 17.12*:

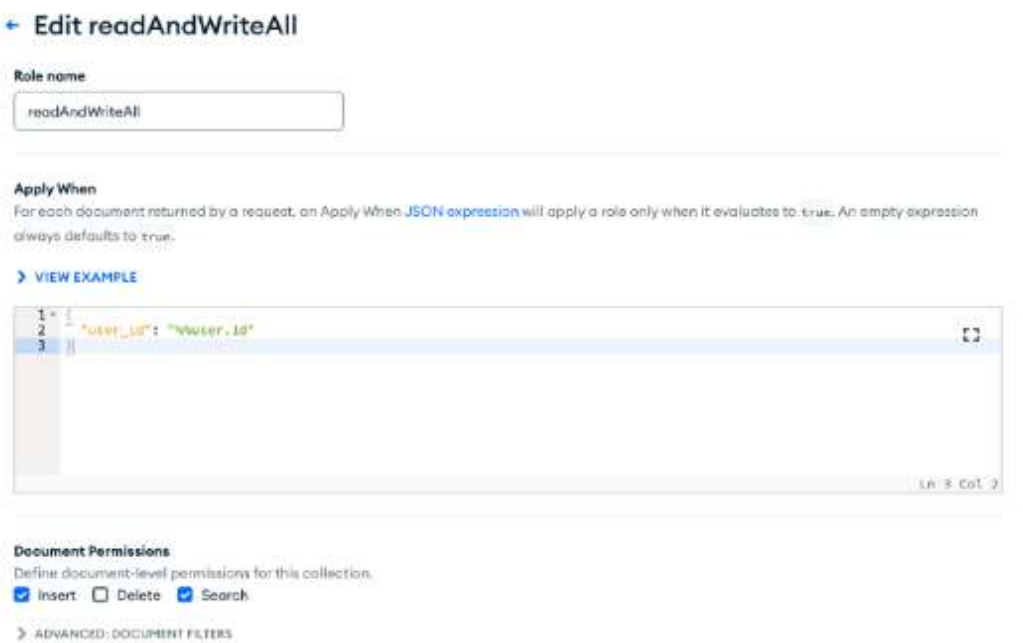


Figure 17.12: *Setting Apply When and Document Filters*

For the *Apply When*, as well in the filters in the **`ADVANCED: DOCUMENT FILTERS`** sections, this is expressed as a query like so:

```
{
  "user_id": "%%user.id"
}
```

Finally, save and publish any necessary deploys. We will have users trigger this user create process from within our app.

React App

Now we can get started coding *MongoQuest*! The general interface of the app will look like *Figure 17.13*:

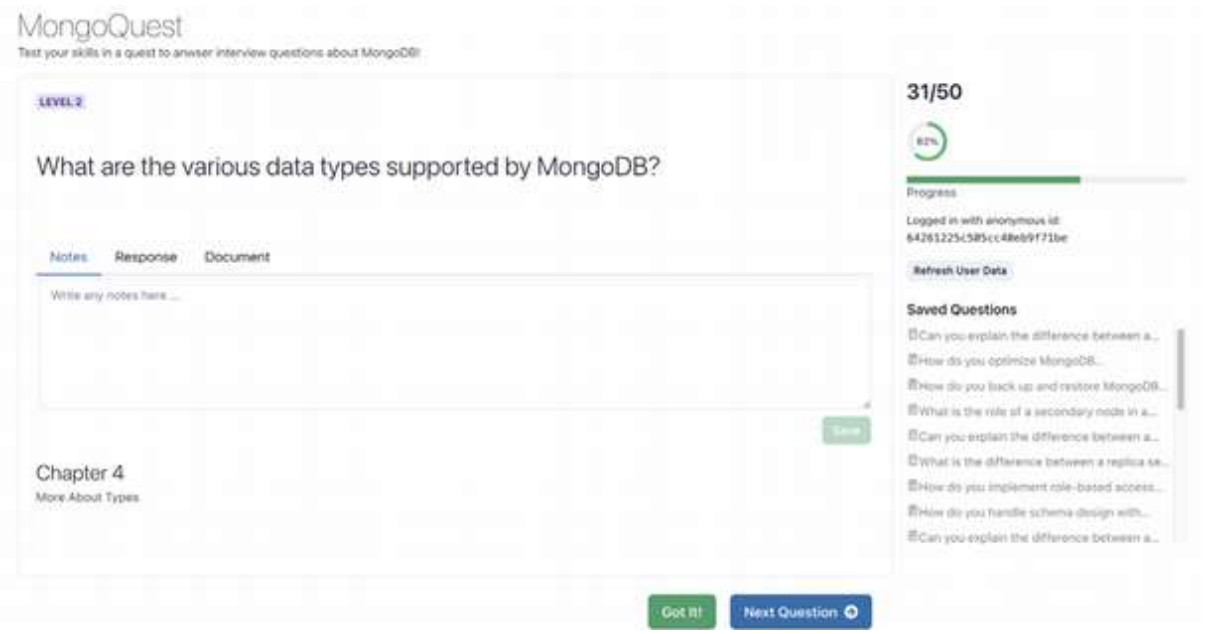


Figure 17.13: *MongoQuest App*

Users will be able to anonymously "login" to save and view their progress, add notes, as well as easily see the question at hand, a possible response, and view the raw question document. To do so, we will use a mix of React, Atlas App Services, the Realm SDK and MongoDB queries.

Create React App

To start things off, the *MongoQuest* app was created using *Create React App* and **`yarn`**, with a command as follows:

```
$ yarn create-react-app mongoquest
```

This will create an empty React app. So, if you want to run or follow along with the fully built out MongoQuest app, you can get the code from the following GitHub repo:

https://github.com/learnmongo/mongoquest

If you are comfortable using JavaScript, feel free to download the source code and follow along on your computer or in the GitHub Codespace, in the repository. If not, no need to worry because we will discuss the code in detail in the next few sections.

There is an included **`README`** file, but in short you can start the app with the following command:

```
$ yarn start
```

Dependencies

There are two major dependencies for our app, the first being the **`realm-web`** package, which we will discuss next. The other is the Chakra UI React package **`@chakra-ui/react`** which we will use as a basis for our UI. You can find out more about *Chakra UI* at **https://chakra-ui.com/**

The rest of the dependencies can be found in the **`package.json`** file in the app's git repository.

Using the Realm Web SDK

We will not be going into an in-depth explanation of how the Realm Web SDK works, as that could easily take up a few chapters on its own. Instead, we will use real examples from the *MongoQuest* app code to illustrate its usage. If you are interested in learning more on your own, you can there are a few *Quick Start* guides we recommend:

https://www.mongodb.com/docs/realm/web/quickstart/

https://www.mongodb.com/developer/languages/javascript/realm-web-sdk/

At a high level, you can connect to Realm via JavaScript as follows:

```
1    // add your App ID
2    const APP_ID = "YOUR_APP_ID";
3    const atlas = new Realm.App({ id: APP_ID });
4
5    // create an anonymous user
6    const credentials = Realm.Credentials.anonymous();
```

```
7
8     // login the user
9     const user = await atlas.logIn(credentials);
```

You supply your **App ID** we mentioned earlier on line **2**, and then setup an anonymous user on line **6**. Then on line **9**, we pass that user to the **logIn()**, which will then allow us to connect to Atlas as that user and run commands and queries. We will cover all this in more detail, in the next few sections.

Coding the App

This app is purposely coded in such a way as to give *examples* of how to do things with Atlas and Realm. In many cases, any production code would look different; for example, it might have more safety checks, and so on. Take any of these examples as just that, examples, and if you plan to make your own app, use your best judgement and best practices.

The following code also, for conciseness, does not include tests. In any real app, you should be testing your code, and this application already has some of the shell to create and run tests if you so wish. With all that said, let us jump in and see how we can make an app with MongoDB *App Services*!

Core Components

In the next few sections, we will review the code for the core components of our app. If you are unfamiliar with React, we will try and highlight the key concepts, which apply, for the most part, to all the available SDKs. You can find all the components and files we will discuss in the git repository.

The start of our app will be the **App.js** component, as with most React apps. The contents of the **<App />** component are fairly simple:

```
1     import React from "react";
2     import { ChakraProvider } from "@chakra-ui/react";
3     import { RealmContextProvider } from "./context/RealmContext";
4
5     import MongoQuest from "./components/MongoQuest";
6
7     const App = () => {
8      return (
9        <ChakraProvider>
```

```
        <RealmContextProvider>
          <MongoQuest />
        </RealmContextProvider>
      </ChakraProvider>
    );
  };

  export default App;
```

App.js

This component contains two providers: one is for *Chakra UI*, the React component library that we will use to build the UI for *MongoQuest*. The other provider is a custom component, the **<RealmContextProvider>** which we will use for interacting and sharing data from *Realm* with all the components under it. We will discuss this *Realm Context* component in greater detail, in a later section. Suffice to say that each custom Context Provider we have, will have a corresponding hook to get data out of the Provider, such as **useRealmContext**. Again, we will discuss this in much more detail shortly.

All these components wrap the **<MongoQuest />** component, where our app really starts. You can find it in the **components/MongoQuest.js** file. This is a real, working fully functional app, so there is a lot going on here. Do not worry about all the details, we will break it down more simply, but let us start with the full code:

```
import { useMediaQuery, Grid, GridItem, Flex, Text } from "@
chakra-ui/react";
import { useRealmContext } from "../context/RealmContext";

import Header from "./Header";
import UserNav from "./UserNav";
import LoggedOut from "./LoggedOut";
import LoggedIn from "./LoggedIn";
import { UserContextProvider } from "../context/UserContext";

const MongoQuest = () => {
 const [isLargerThan800] = useMediaQuery("(min-width: 800px)");
 const { user } = useRealmContext();

```

```
14      const columns = isLargerThan800 && user ? 4 : 1;
15
16      const userNav = (
17        <GridItem h={isLargerThan800 && "60dvh"} colSpan={1}>
18          <UserNav />
19        </GridItem>
20      );
21
22      return (
23        <Grid
24          h="100px"
25          templateColumns={`repeat(${columns}, 1fr)`}
26          gap={2}
27          m={isLargerThan800 ? 6 : 2}
28        >
29          <GridItem colSpan={columns}>
30            <Header />
31          </GridItem>
32          {!user && <LoggedOut />}
33          {user && (
34            <UserContextProvider>
35              {!isLargerThan800 && <GridItem colSpan={1}>{userNav}</
    GridItem>}
36              <GridItem colSpan={columns - 1}>
37                <LoggedIn />
38              </GridItem>
39              {isLargerThan800 && userNav}
40            </UserContextProvider>
41          )}
42          <GridItem mt={10} h="50px" colSpan={columns}>
43            <Flex justifyContent="center">
44              <Text fontSize="sm" fontWeight={300}>
45                MongoQuest 2023
```

```
46              </Text>
47            </Flex>
48          </GridItem>
49        </Grid>
50      );
51    };
52
53     export default MongoQuest;
```

components/MongoQuest.js

The first part to bring your attention to is line **12**, where we are using a hook to get the Realm Context and pulling out the **user**.

```
const { user } = useRealmContext();
```

When the app first loads, we will check if the user is logged in. We will do this by using Realm's "anonymous user" logins, as we discussed earlier in the chapter. If the user is not logged in, the **user** variable will be **undefined**, and thus we will show an interface for them to login and hide parts of the app that should only show for logged in users.

We can see an example of this on lines **29-41**:

```
29    <GridItem colSpan={columns}>
30      <Header />
31    </GridItem>
32    {!user && <LoggedOut />}
33    {user && (
34      <UserContextProvider>
35      {!isLargerThan800 && <GridItem colSpan={1}>{userNav}</GridItem>}
36        <GridItem colSpan={columns - 1}>
37          <LoggedIn />
38        </GridItem>
39        {isLargerThan800 && userNav}
40      </UserContextProvider>
41    )}
```

Within this component, if the **user** variable is not set on line **32**, we will show a **<LoggedOut>** component, since that means they are not logged in. On the next

line **33**, we are checking again for the **user**, and this time, if it is set, we will show a **<LoggedIn>** component and the contents of **userNav**, which will display our questions, and the user's progress.

Note: If you are unaware what && does in JavaScript, it is a "short-circuit evaluation" which will render the contents on the right side if the left side is true.

If the user is logged out, they will see a page similar to *Figure 17.14*:

Figure 17.14: *MongoQuest start screen*

Thus, everything else in the UI is hidden until the user presses the **Begin** button. Let us look at the **<LoggedOut>** component, briefly:

```
1    import { Stack, Text, Button } from "@chakra-ui/react";
2   import { RiSwordLine } from "react-icons/ri";
3
4    import { useRealmContext } from "../context/RealmContext";
5
6   const LoggedOut = () => {
7     const { loginUser } = useRealmContext();
8     return (
9       <Stack height="600px" width="95dvw" spacing={5}>
10        <Text fontSize="xl" fontWeight={200}>
11          Do you wish to begin your quest?
12        </Text>
13        <Button
14          size="lg"
15          colorScheme="green"
16          leftIcon={<RiSwordLine />}
17          rightIcon={<RiSwordLine />}
```

```
          onClick={loginUser}
        >
          Begin
        </Button>
      </Stack>
    );
  };

  export default LoggedOut;
```

components/LoggedOut.js

In this component, we are again using our *Realm Context* hook **`useRealmContext`**, this time using the **`loginUser`** function that enables the user to initiate and "login" via the Anonymous User functionality of Realm. When the user presses the **`Begin`** button, the login code runs, and the app will automatically re-render and show the previously hidden components which only show for logged in users, as you can see in *Figure 17.15*:

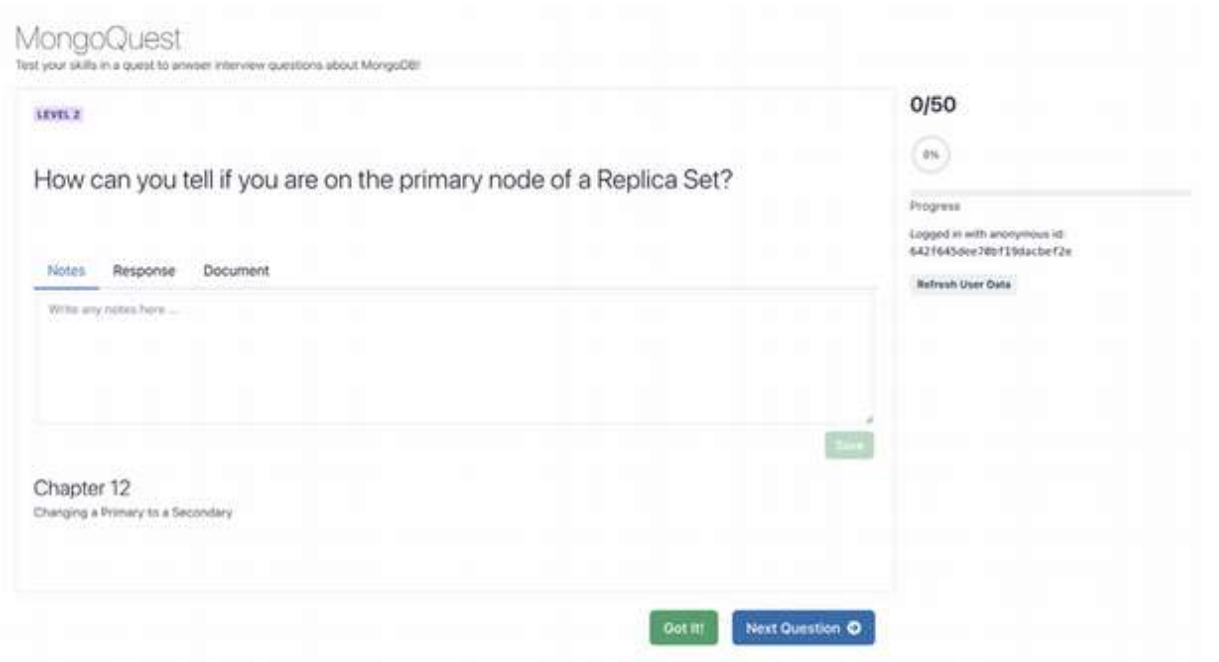


Figure 17.15: *MongoQuest as seen as a logged in user*

Realm Context

As we have discussed, to share the information about our Realm connection with all the components in our app, we will "wrap" them in a "context". If you are unfamiliar

with this concept, it allows us to easily "pass down" data, deep into our application without passing down props.

You can read more about how React Context works here:

https://react.dev/learn/passing-data-deeply-with-context

In our app, we will put this context in a component called **`<RealmContextProvider>`** and provide a **`useRealmContext`** hook. The code for our component is as follows and can be found in the **`context/RealmContext.js`** file:

```
1   import React from "react";
2   import * as Realm from "realm-web";
3
4   // Set your App ID
5   const APP_ID = "YOUR_APP_ID";
6   const atlas = new Realm.App({ id: APP_ID });
7
8   const RealmContext = React.createContext({});
9
10  export function RealmContextProvider({ children }) {
11   const [user, setUser] = React.useState(atlas.currentUser);
12
13   const loginUser = React.useCallback(async () => {
14     const user = await atlas.logIn(Realm.Credentials.
    anonymous());
15     setUser(user);
16   }, []);
17
18   const refreshUser = React.useCallback(async () => {
19     await atlas.currentUser.refreshCustomData();
20     setUser(atlas.currentUser);
21   }, []);
22
23   const logoutUser = React.useCallback(
24     async () => await atlas.currentUser.logOut(),
25     []
```

```
26      );
27
28      return (
29        <RealmContext.Provider value={{ user, loginUser, refreshUs-
      er, logoutUser }}>
30          {children}
31        </RealmContext.Provider>
32      );
33    }
34
35     export const useRealmContext = () => React.
      useContext(RealmContext);
```

context/RealmContext.js

On lines **5-6**, you will need to provide the **App ID** from your Realm application, which we covered earlier.

We expose the **user** via this context, but within the context, we store the user in state, after logging them in. This **user** is the same anonymous user that we created when the user pressed the **Begin** button in the **<LoggedOut />** component.

We will be able to use the data and methods from this context to run a number of asynchronous operations against Realm, such as logging in a user (lines 1**3**-**16**), refreshing their data (**18**-**21**), and logging them out (**23**-**26**). All these methods wrap calls to Atlas, which we start off on line **6**, by initiating our Atlas app and assigning it to the **atlas** variable.

We can then use **atlas.login()** and pass in **Realm.Credentials.anonymous()**, which will create a new anonymous user or reestablish the user from the browser's cache. Our **refreshUser()** method will update the *Custom User Data* for the user, which is cached and may become out of date. We may occasionally want to trigger this refresh as we add more user data custom data, such as which questions they have marked as answered. More on that later.

Context Hooks

As we briefly touched upon earlier, along with the components, each of our Contexts have their own hooks, to get back the values in the Context. They are named in the format of **useXxxxContext** and can be used like so:

```
const { var1, var2 } = useXxxxxContext();
```

This "pulls out" the **`var1`** and **`var2`** variables from the Context, and we can now use them in our component. Any components that are "wrapped", or a child of the Context Provider component, have access to these hooks.

User Context

Once we have a logged in user, we can provide data and context about that user to any components that needs it by using another Context Provider, the **`<UserContextProvider>`**. You can see the code for this as follows, or in the **`context/UserContext.js`** file.

This component and its hook can be used to get information about logged in user, such as the count of questions they have marked answered (lines **`11-13`**) or an array of those questions' **`_id`** (lines **`15-18`**) as well as an array of the saved data, like a note, for questions (lines **`20-23`**).

```
1   import React from "react";
2   import { useUser } from "../hooks/useUser";
3   import { useRealmContext } from "./RealmContext";
4
5   const UserContext = React.createContext({ user: null });
6
7   export function UserContextProvider({ children }) {
8    const { user } = useRealmContext();
9    const { setAnswered } = useUser();
10
11   const [answeredCount, setAnwseredCount] = React.useState(
12     user.customData?.questions?.answered?.length ?? 0
13   );
14
15   const answeredQuestions = React.useMemo(
16     () => user?.customData?.questions?.answered ?? [],
17     [user]
18   );
19
20   const savedQuestions = React.useMemo(
21     () => user?.customData?.questions?.saved ?? [],
22     [user]
```

```
23      );
24
25      const savedQuestionIds = React.useMemo(
26        () => savedQuestions.map((question) => question.id),
27        [savedQuestions]
28      );
29
30      const setQuestionStatus = React.useCallback(
31        async (id, status) => {
32          await setAnswered(id, status);
33          setAnwseredCount(user.customData?.questions?.answered?.
    length);
34        },
35        [setAnswered, user.customData?.questions?.answered?.length]
36      );
37
38      return (
39        <UserContext.Provider
40          value={{
41            answeredCount,
42            answeredQuestions,
43            savedQuestions,
44            savedQuestionIds,
45            setQuestionStatus,
46          }}
47        >
48          {children}
49        </UserContext.Provider>
50      );
51    }
52
53     export const useUserContext = () => React.
    useContext(UserContext);
```

context/UserContext.js

Within this context, we provide a function **`setQuestionStatus()`**, which we can use to set a question as "answered" on lines **`25-28`**, which in turn calls a function from the **`useUser`** hook, **`setAnswered()`** which we will discuss next. We make sure to **`await`** this call, so that we can show a loading state, and so that rest of the UI will update as soon the operation completes, like the count on the right side menu and the progress bar, as seen in *Figure 17.16*:

Figure 17.16: *User progress component*

We wrap all the components that might need information about the user in this component, and they can use the **`useUserContext`** hook to get a value for **`answeredCount`**, and so on.

useUser Hook

Separate from the Context, we have also created a hook for performing actions with, or on the logged in user. You can find this in **`hooks/useUser.js`** file, or as follows. Notice that we pull the **`user`** from the *Realm Context* on line **`5`**, and create a **`mongoClient`** connection on lines **`6-7`**, which takes a data source name, in our case **`mongodb-atlas`**, as that is the name of our *Linked Data Source*. However, yours may differ.

We will use this client to perform queries against our Atlas collection:

```
1    import React from "react";
2    import { useRealmContext } from "../context/RealmContext";
3
4    export const useUser = () => {
5     const { user, refreshUser } = useRealmContext();
6     const mongo = user.app.currentUser.mongoClient("mongodb-atlas");
7     const collection = mongo.db("mongoquest").collection("users");
8
```

```
  const [isLoading, setIsLoading] = React.useState(false);

  const setAnswered = React.useCallback(
    async (id, status = "ADD") => {
      if (status === "ADD") {
        setIsLoading(true);
        const response = await collection.updateOne(
          { user_id: user.id },
          { $addToSet: { "questions.answered": id } }
        );
        await refreshUser();
        setIsLoading(false);
        return response;
      }

      if (status === "REMOVE") {
        setIsLoading(true);
        const response = await collection.updateOne(
          { user_id: user.id },
          { $pull: { "questions.answered": id } }
        );
        await refreshUser();
        setIsLoading(false);
        return response;
      }
    },
    [collection, refreshUser, user.id]
  );

  const saveNote = React.useCallback(
    async (id, note = "") => {
      setIsLoading(true);
      // example of running two queries back to back
```

```
42          let responses = [];
43
44          responses.push(
45            await collection.updateOne(
46              {
47                user_id: user.id.toString(),
48                "questions.saved.id": id,
49              },
50              {
51                $set: { "questions.saved.$.note": note },
52              }
53            )
54          );
55
56          responses.push(
57            await collection.updateOne(
58              {
59                user_id: user.id.toString(),
60              },
61              {
62                 $addToSet: { "questions.saved": { id: id, note: note
    } },
63              },
64              { upsert: true }
65            )
66          );
67
68          await refreshUser();
69          setIsLoading(false);
70
71          return responses;
72        },
73        [collection, refreshUser, user.id]
```

```
74      );
75
76      return {
77        setAnswered,
78        saveNote,
79        isLoading,
80      };
81    };
```

hooks/useUser.js

This hook has two methods: **`setAnwsered()`** and **`saveNote()`**. We will not go into detail on these, other than pointing out we are either making an **`updateOne()`** query, with **`$addToSet`** to add a question's **`_id`** to the array, on lines **`15`**-**`18`** or a **`$pull`** to remove a question's **`_id`** from the array on lines **`26`**-**`29`**.

To save a note, we are showing an example of running two queries back-to-back, on lines **`38`**-**`66`**. In short, this will either add a note object to our array or update it by running these two queries one after the other.

useAggregate Hook

Another custom hook we have created is **`useAggregate`**, which we will use to run queries on our collection. It is not necessary do this in a hook. However, for ease of use and explanation, we have composed this in a hook, to perform aggregation queries. The hook has three parameters, the **`databaseName`**, **`collectionName`** and an aggregation **`pipeline`** array, as seen on lines **`4`**-**`8`**.

We use these parameters plus the **`user`** from our **`useRealmContext`** hook, on line **`9`**, to create a database client, on lines **`11`**-**`12`**. Refer to the following snippet:.

```
1   import React from "react";
2   import { useRealmContext } from "../context/RealmContext";
3
4   export const useAggregate = ({
5     databaseName = "mongoquest",
6     collectionName,
7     pipeline = [],
8   }) => {
9     const { user } = useRealmContext();
```

```
10
11      const mongo = user.app.currentUser.mongoClient("mongodb-
    atlas");
12      const collection = mongo.db(databaseName).
    collection(collectionName);
13
14      const [results, setResults] = React.useState([]);
15      const [isLoading, setIsLoading] = React.useState(false);
16
17      const getResults = React.useCallback(async () => {
18        setIsLoading(true);
19        const aggregation = await collection.aggregate(pipeline);
20        setResults(aggregation);
21        setIsLoading(false);
22        return aggregation;
23      }, [collection, pipeline]);
24
25      React.useEffect(() => {
26        (async () => await getResults())();
27        // eslint-disable-next-line react-hooks/exhaustive-deps
28      }, []);
29
30      return {
31        results,
32        firstResult: results[0],
33        getResults,
34        isLoading,
35      };
36    };
```

hooks/useAggregate.js

We then set which collection we will use on line **12** and use that to run an aggregate query on line **19**, which accepts the **pipeline** array, which might look like the following:

```
const pipeline = [
  { $match: { title: 'Tacos' } },
  { $limit: 3 },
];
```

In this example, we have a two stage pipeline that will first **`$match`** for any documents with the **`title`** "Tacos" and then **`$limit`** to three results. Of course, this particular query did not really need be an aggregate query, but you could add a lot more stages by simply adding more members to the **`pipeline`** array.

In the **`useAggregate`** hook, we set and update a loading state variable called **`isLoading`** with **`setIsLoading()`** in lines **`18`** and **`21`**. We can reference this value from the hook to set a loading state in the UI. Notice that we **`await`** the call to aggregate, which will need to make a network request to the Realm service in order to query our database on Atlas.

Lastly, we both set the results to a variable in state and return the results to the caller on line **`22`**. This is mostly an example of a way you can code these sort of interactions. For example, you can also run the **`useEffect`** hook as we have, on line **`25-28`** to **`await`** the **`getResults()`** method and thus automatically set the query results to the **`results`** variable, as well as assigning the first item in the array to the **`firstResult`** variable on line **`32`** and so on.

The usage of this hook might look as follows:

```
1     const {
2       isLoading,
3       firstResult,
4       getResults,
5     } = useAggregate({
6       collectionName: "questions",
7       pipeline,
8     });
```

Question Context

The last, and in many ways most important bit of data, is questions, which we make available via the **`<QuestionContextProvider>`** and its **`useQuestionContext`** hook. Within this context, we are going to fetch the question to display using an aggregation query and the **`useAggregate`** hook. You can see the following code, or in the **`context/QuestionContext.js`** file:

```
import React from "react";
import { useAggregate } from "../hooks/useAggregate";
import { useUserContext } from "./UserContext";

const QuestionContext = React.createContext({ question: null });

export function QuestionContextProvider({ children }) {
 const { answeredQuestions } = useUserContext();

 const pipeline = [
   // stage to hide answered questions
   {
     $match: { _id: { $nin: answeredQuestions } },
   },
   {
     $sample: { size: 3 },
   },
   { $limit: 1 },
 ];

 const {
   isLoading: isLoadingQuestion,
   firstResult: question,
   getResults: getNextQuestion,
 } = useAggregate({
   collectionName: "questions",
   pipeline,
 });

 const isAnswered = React.useMemo(
   () => answeredQuestions?.some((id) => id.$oid === question
?._id.toString()),
   [answeredQuestions, question]
```

```
41      );
42
43      return (
44        <QuestionContext.Provider
45          value={{ isLoadingQuestion, question, isAnswered, getNex-
    tQuestion }}
46        >
47          {children}
48        </QuestionContext.Provider>
49      );
50    }
51
52     export const useQuestionContext = () => React.
    useContext(QuestionContext);
```

context/QuestionContext.js

To get one question, at random, excluding questions we have already marked as answered, we will setup a pipeline with a couple stages:

```
8     const { answeredQuestions } = useUserContext();
9
10     const pipeline = [
11       // stage to hide answered questions
12       {
13         $match: { _id: { $nin: answeredQuestions } },
14       },
15       {
16         $sample: { size: 3 },
17       },
18       { $limit: 1 },
19     ];
```

Before building out our pipeline, we need to get some extra data. For example on line **8**, we get **answeredQuestions** from the *User Context*, which is an array made up of the **_id** for each previously answered question. We then build our **pipeline**, in an array of its own with three stages.

The first step will exclude answered questions by using a **`$match`** stage which uses a **`$nin`** (*not in*) query and the array of question ids. The second stage uses **`$sample`** to randomize the results. The third stage uses **`$limit`** to only get back one of the randomized questions.

The next bit, using the **`useAggregate`** hook, is a little more complicated, but we will break it down.

```
21     const {
22        isLoading: isLoadingQuestion,
23        firstResult: question,
24        getResults: getNextQuestion,
25      } = useAggregate({
26        collectionName: "questions",
27        pipeline,
28      });
```

When we call the hook, on lines **`25-28`**, we send it two parameters, the **`collectionName`**, and the **`pipeline`** array we just created. We then extract, and rename, the **`isLoading`** and **`firstResult`** variables along with the **`getResults`** function. We will use these elsewhere in the app to see if the question is loading, get the question and provide a function to call to get the next question.

Displaying Questions

The *Question Context* Provider is used in the **`<LoggedIn>`** component and wraps the two components that render the text and metadata for the question, as well as the navigation for the questions, which you can find in **`components/LoggedIn.js`** or as follows:

```
1     import React from "react";
2
3     import { QuestionContextProvider } from "../context/Question-
    Context";
4
5     import Question from "./Question";
6     import QuestionNav from "./QuestionNav";
7
8     const LoggedIn = () => {
9      return (
```

```
      <QuestionContextProvider>
        <Question />
        <QuestionNav />
      </QuestionContextProvider>
    );
  };

  export default LoggedIn;
```

components/LoggedIn.js

Question Component

For displaying our question, we have created a number of components, mostly to break up the formatting code and keep the amount of code in any one component more manageable. You can see the code for main question component in **`components/Question/Question.js`** or as follows:

```
import React from "react";
import { useQuestionContext } from "../../context/QuestionContext";
import { Heading, Badge, Card, CardHeader, CardBody, Stat, StatNumber, StatHelpText, Flex, Center } from "@chakra-ui/react";
import QuestionLoader from "./QuestionLoader";
import QuestionTabs from "./QuestionTabs";

const getLevelColor = (level) =>
  level === 1 ? "green" : level === 2 ? "purple" : undefined;

const QuestionDisplay = () => {
  const { question } = useQuestionContext();

  // destructure question document
  const {
    question: questionText,
    response: {
```

```
17          short: shortResponse,
18          reference: { chapter, section },
19        },
20      } = question;
21
22      return (
23        <Card variant="outline" height="65dvh" minHeight="500px" max-
    Height="700px" mb={5}
24        >
25          <CardHeader>
26            <Badge mb="5" colorScheme={getLevelColor(question.lev-
    el)}>
27              LEVEL {question.level}
28            </Badge>
29            <Flex height="10dvh">
30              <Center>
31                <Heading as="h3" size="lg" fontWeight={300}>
32                  {questionText}
33                </Heading>
34              </Center>
35            </Flex>
36          </CardHeader>
37          <CardBody>
38            <QuestionTabs shortResponse={shortResponse} />
39            <Stat height="60px">
40              <StatNumber fontWeight={300}>Chapter {chapter}</Stat-
    Number>
41              <StatHelpText>{section}</StatHelpText>
42            </Stat>
43          </CardBody>
44        </Card>
45      );
46    };
```

```
47
48    const Question = () => {
49      const { isLoadingQuestion, question } = useQuestionContext();
50
51      if (isLoadingQuestion || !question) return <QuestionLoader />;
52
53      return <QuestionDisplay />;
54    };
55
56    export default Question;
```

components/Question/Question.js

A few items to take note of are, first off, there are technically two components defined here **<QuestionDisplay>** and **<Question>**. We do this so that near the bottom of the file, we can make a check for if a question has been loaded yet, meaning the query has completed and returned a result, and if not, we show the **<QuestionLoader>** component as a placeholder on line **51**.

When we do have a question returned, we instead display the **<QuestionDisplay>** component, as follows:

```
48    const Question = () => {
49      const { isLoadingQuestion, question } = useQuestionContext();
50
51      if (isLoadingQuestion || !question) return <QuestionLoader />;
52
53      return <QuestionDisplay />;
54    };
```

Another point of interest is line **11**, where we get a **question** object from the *Question Context* that we just discussed. Additionally, on lines **13-20** we take the **question** object, which will have the same form as the document in our collection, and destructure it, or assign fields inside that object to their own variables.

```
13      // destructure question document
14      const {
15        question: questionText,
16        response: {
```

```
17          short: shortResponse,
18          reference: { chapter, section },
19        },
20      } = question;
```

This allows us, for example, to refer to a **`chapter`** variable, assigned on line **`18`** later on line **`40`**, instead of having to use the much longer **`question.response.reference.chapter`** which is the actual location of that data within the object/document.

We display the questions level, with a colored badge on lines **`26-28`** as well as the text of the question in a sized large header on lines **`29-35`**. There are three tabs under each question, within which, the user can save a note, see the possible response and also see the raw document from the collection. We compose this together in side a **`<QuestionTabs>`** component which we use on line **`38`**.

Question Tabs Component

Directly under our question text, are three tabs which allow the user to save a note, look a possible response, or view the raw document for the question, as seen in *Figure 17.17*:

Figure 17.17: *Question tabs component*

The component that composes these tabs is mostly presentational, in that it wraps the other tab components. You can see it in **`components/Question/QuestionTabs.js`** or as follows:

```
1   import { Tab, TabList, TabPanel, TabPanels,Tabs, Text }
    from "@chakra-ui/react";
2   import QuestionNote from "./QuestionNote";
3   import RawQuestion from "./RawQuestion";
4
5   const QuestionTabs = ({ shortResponse }) => (
6    <Tabs size="md" height="250px">
7      <TabList>
```

```
8           <Tab>Notes</Tab>
9           <Tab>Response</Tab>
10          <Tab>Document</Tab>
11        </TabList>
12        <TabPanels>
13          <TabPanel p={1}>
14            <QuestionNote />
15          </TabPanel>
16          <TabPanel height="180px" overflow="auto">
17            <Text>{shortResponse}</Text>
18          </TabPanel>
19          <TabPanel>
20            <RawQuestion />
21          </TabPanel>
22        </TabPanels>
23      </Tabs>
24    );
25
26    export default QuestionTabs;
```

components/Question/QuestionTabs.js

We will focus on the **<QuestionNote>** component on line **14** and the **<QuestionRaw>** component on line **20**. These will show up directly under the question text, as seen in *Figure 17.18*:

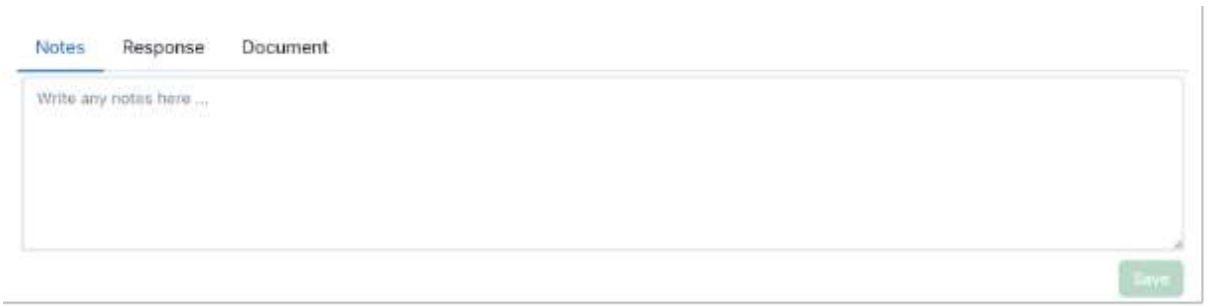


Figure 17.18: *Question note component*

First, let us look at the **<QuestionNote>** component's code, which you can find in **components/Question/QuesionNote.js** or as follows. This component also has a

decent number of things going on; however, of interest here, is that we are pulling from both the *User* and *Question* contexts on line **8** and **9**.

```
1   import React from "react";
2   import { Textarea, Flex, Button } from "@chakra-ui/react";
3   import { useUserContext } from "../../context/UserContext";
4   import { useQuestionContext } from "../../context/QuestionCon-
    text";
5   import { useUser } from "../../hooks/useUser";
6
7   const QuestionNote = () => {
8    const { savedQuestions } = useUserContext();
9    const { question } = useQuestionContext();
10
11   const { isLoading: isUserDataLoading, saveNote } = useUser();
12   const [currentNote, setCurrentNote] = React.useState();
13
14   const note = React.useMemo(() => {
15     if (!question) return null;
16
17     const savedNote = savedQuestions?.filter(
18       (note) => note.id.$oid === question._id.toString()
19     );
20     return savedNote[0]?.note;
21   }, [question, savedQuestions]);
22
23   React.useEffect(() => {
24     setCurrentNote(note ?? undefined);
25   }, [question._id, note]);
26
27   const handleNoteChange = React.useCallback((e) => {
28     let inputValue = e.target.value;
29     setCurrentNote(inputValue);
30   }, []);
```

```
31
32      return (
33        <>
34          <Textarea
35            size="sm" height={150} maxLength={500}
36            value={currentNote ?? note}
37            onChange={handleNoteChange}
38            placeholder="Write any notes here ..."
39          />
40          <Flex mt={2} justifyContent="right">
41            <Button
42              colorScheme="green" size="sm"
43              isLoading={isUserDataLoading}
44              isDisabled={currentNote === note ? true : undefined}
45              onClick={async () => await saveNote(question._
    id, currentNote)}
46            >
47              Save
48            </Button>
49          </Flex>
50        </>
51      );
52    };
53
54     export default QuestionNote;
```

components/Question/QuestionNote.js

We will review some of the hooks used on lines **11**-**30** in a moment. For now, we can focus on the UI returned starting at line **32**. We see here the code for the text area we could see in *Figure 17.16* as well as the green **save** button. On line **36**, we set the **value** of this text area with either the **currentNote**, which will be anything the user types in the box, or on load, any matching saved **note** we get from the user's **customData**. If the user types anything in the text area, we will fire the **handleNoteChange** function, as seen on line **37**.

Lastly, we will trigger a database save, using the **saveNote** function, from the **useUser** hook, on line **45**. Overall, to get the **note** we will use the logic on lines **12-25**:

```
12      const [currentNote, setCurrentNote] = React.useState();
13
14      const note = React.useMemo(() => {
15        if (!question) return null;
16
17        const savedNote = savedQuestions?.filter(
18          (note) => note.id.$oid === question._id.toString()
19        );
20        return savedNote[0]?.note;
21      }, [question, savedQuestions]);
22
23      React.useEffect(() => {
24        setCurrentNote(note ?? undefined);
25      }, [question._id, note]);
```

Using the user's **customData** on line **16**, will assign the array of saved questions to **savedQuestion**. We then run the **filter** array method on this array and look to see if the saved item's **id** is equal to the question's **_id**. There are a number of ways to handle this, but we bring up this comparison to point out a unique aspect of dealing with MongoDB data.

Since the default for a document's **_id** is an **ObjectId**, we need to know how to compare **ObjectId** data. On line **18**, we can see a comparison that looks as follows:

```
18      (note) => note.id.$oid === question._id.toString()
```

This is using the **id.$oid**, which is a string, from the saved items array, which looks like the ids in the following example:

```
"questions": {
  "saved": [
    {
      "id": { "$oid": "64192b927b7fbb7150674781" },
      "note": "I am not sure yet."
    },
```

```
    {
      "id": { "$oid": "64192b927b7fbb7150674798" },
      "note": "Make sure to check back on Chapter 12"
    }
  ]
}
```

The question's **_id** on the other hand, is a **ObjectId**. Therefore, we will need to use **.toString()** on to get a string so that we can compare, we do this like so: **question._id.toString()**.

Then, we can use the value of the note field to populate the text area on lines **23**-**25**, on load, or, to reiterate, if the user changes the value in the text area. We will handle that with the **handleNoteChange** function on lines **27**-**30**:

```
27    const handleNoteChange = React.useCallback((e) => {
28      let inputValue = e.target.value;
29      setCurrentNote(inputValue);
30    }, []);
```

You can review the rest of the code on your own and see how we are using the loading status, or the value of the text area compared to a saved note, to disable the **save** button, or show it in a loading state, and so on.

In the next tab, back in **<QuestionTabs>**, we will show a possible response, on lines **16**-**18**:

```
16    <TabPanel height="180px" overflow="auto">
17      <Text>{shortResponse}</Text>
18    </TabPanel>
```

We can see this rendered in *Figure 17.19*:

Figure 17.19: *Displaying question response text*

Lastly, the user can view the raw document itself, as you can see in *Figure 17.20*:

Figure 17.20: *Displaying question document*

We do this using the **<RawQuestion>** component, which you can find the code for in the **components/Question/RawQuestion.js** file or as follows:

```
1    import { useQuestionContext } from "../../context/QuestionCon-
   text";
2
3    const RawQuestion = () => {
4     const { question } = useQuestionContext();
5     return (
6       <div style={{ padding: 6, backgroundColor: "#EFEFEF", max-
   Height: 180, overflow: "auto"  }} >
7         <pre style={{ fontSize: 10, whiteSpace: "pre-wrap" }}>
8           {JSON.stringify(question, null, 1)}
9         </pre>
10      </div>
11    );
12  };
13
14   export default RawQuestion;
```

components/Question/RawQuestion.js

Here, we get the whole **question** document from the *Question Context* on line **4**, which we then use on line **8** along with the JavaScript **JSON.stringify** method

to turn the object into a formatted string, which again, we can see in the preceding *Figure 17.20*.

Question Navigation Component

The last component we will cover generates the buttons on the bottom right of the question and allows the user to mark the question as "done" by pressing the **`Got It!`** button and/or moves on to the next random question. They look like *Figure 17.21*:

Figure 17.21: *Question navigation buttons*

The code to generate them, and handle their actions, can be found in **`components/QuestionNav.js`** or as follows. By now, you are probably starting to understand the pattern and might have an idea how we did this.

We use both the *User Context* and the *Question Context* hook's on lines **`8`** and **`9`**. We then use the methods and statuses from these hooks to change the question's status to "answered" when the **`Got It!`** button is pressed, on lines **`25`**-**`28`**, and to change the state of that button to either loading or disabled on lines **`23`**-**`24`**:

```
1   import React from "react";
2   import { Button, Flex } from "@chakra-ui/react";
3   import { FaArrowCircleRight, FaCheckCircle } from "react-icons/
    fa";
4   import { useUser } from "../hooks/useUser";
5   import { useQuestionContext } from "../context/QuestionContext";
6   import { useUserContext } from "../context/UserContext";
7
8   const QuestionNav = () => {
9    const { setQuestionStatus } = useUserContext();
10   const { isLoadingQuestion, question, isAnswered, getNextQuesti
    on } =
11     useQuestionContext();
12   const { isLoading: isLoadingUserData } = useUser();
13
14   const [hasAnswered, setHasAnswered] = React.useState();
```

```
15
16      React.useEffect(() => setHasAnswered(isAnswered), [isAnswered]
    );
17
18      if (!question) return null;
19
20      return (
21        <Flex justifyContent="right" mr={6}>
22          <Button colorScheme="green" mr={4}
23            isDisabled={hasAnswered}
24            isLoading={isLoadingUserData}
25            onClick={() => {
26              setHasAnswered(true);
27              setQuestionStatus(question._id);
28            }}
29          >
30            {hasAnswered ? <FaCheckCircle /> : «Got It!»}
31          </Button>
32          <Button
33            colorScheme="blue"
34            isLoading={isLoadingQuestion}
35            loadingText="Loading Quest"
36            spinnerPlacement="end"
37            rightIcon={!isLoadingQuestion && <FaArrowCircleRight />}
38            onClick={getNextQuestion}
39          >
40            Next Question
41          </Button>
42        </Flex>
43      );
44    };
45
46     export default QuestionNav;
```

components/NavQuestion.js

We do the same sort of thing for the **`Next Question`** button on lines **`32`**-**`41`**, with a loading state, as well as calling the **`getNextQuestion`** function from the *Question Context* on click.

Wrap Up

There are a number of other components we did not cover here due the length of the chapter. However, make sure to check out the code in the *MongoQuest* repository:

https://github.com/learnmongo/mongoquest

Additionally, it should be noted that some of the code we made to query our collection could also be composed in Atlas Serverless Functions instead. To learn more about those, take a look at the following resource:

https://www.mongodb.com/docs/atlas/app-services/functions/

Conclusion

In this chapter, we showed how you can write a completely "serverless" application using Atlas App Services and a Realm SDK. We covered using a free Atlas cluster, configuring its access, linking it as a data source and querying it via JavaScript. We also briefly looked into Serverless Functions on Atlas, as well as some of the Authentication methods Atlas provides.

If you are looking for more adventure, feel free to get a copy of the code and edit the app by adding anything you think is cool, or useful. Additionally, you can play a hosted version at **https://mongoquest.com** the hosted version has a few features not included in the example app but is generally the same. All the interview-like questions in MongoQuest are the same ones that we will present in the next chapter, so you can use MongoQuest to practice, if you wish.

Chapter 18
Jobseeker – Interview Prep

"One important key to success is self-confidence. An important key to self-confidence is preparation."

— Arthur Ashe

"The important thing is not to stop questioning."

— Albert Einstein

Introduction

Well, my friends, we are near the end, and since one of the purposes of this book is to empower anyone looking for a job, or wanting to expand their role with knowledge of MongoDB, we would be remiss to not cover one of the most important aspects of learning new skills, that is, comprehension. Notice that we used the term *comprehension* over *memorization*. You can memorize an answer; but being able to reply, with a thoughtful response, using concepts you have truly comprehended, is much more important.

Thus, in this chapter, we will present a number of question for you to respond to, which you can use to test your comprehension.

Structure

In this chapter, we will discuss the following topics:

- MongoDB Questions and Response
- Questions round 1
- Questions round 2
- Questions round 3
- Questions round 4
- Questions round 5
- Questions round 6
- Questions round 7
- Questions round 8
- Questions round 9
- Questions round 10

Objectives

This chapter will present fifty different interview-like questions, at various levels of difficulty, as well as a *response*. These may not be the exact questions you would be asked in an interview, for which MongoDB skills are required, but you can expect to be asked some variation of them. For each question a response is offered, as well as a reference to the chapter we discussed the topic in general, or directly. We will discuss how to use this chapter's questions in the next section.

Even if you do not plan to have an interview anytime soon, or at all, these questions can be really useful as a review of this book's content, and to test your retention of the concepts you have learned. Why not challenge yourself, and see how well you do?

MongoDB Questions and Response

The rest of this chapter is entirely composed of interview-like questions, increasing in difficulty as we go on. The questions are broken into ten "rounds", like a boxing match, so as to not overwhelm you and give you time to go back and refresh yourself if you find you cannot give a good response. Otherwise, the questions are random in nature and do not go topic by topic, but rather bounce around. While it is unlikely in an interview questions would be so random, being able to be ready to have a response on a wide array of topics should help you have more confidence in general.

How to use these questions

First read the question and try to answer it yourself without looking for the response. Then read the response and see how your own response compares. We have purposely used the word "response" instead of "answer" since multiple, very different, responses may be valid for many of these questions. Use the printed response here as either a validation of your thoughts or a check on your response.

For each response, there is a chapter, and often, a section reference. Use these if you feel you were not able to think of a good response, or as a way to freshen up on the topic. Each response is roughly two sentences or less, to keep things succinct; your actual response may need to be more elaborate. If you feel like you would have a hard time elaborating if asked a follow up question, make sure to review the reference chapter and section, or do your own research.

MongoQuest

Of course, if you setup the *MongoQuest* app in *Chapter 17, MongoDB Atlas - Application Services,* you can use that! It has the same questions, responses, and references, with each question having its own document composing those items.

Since the format of a book makes these sort of *Questions and Responses* sometimes hard to follow, we highly recommend you use the *MongoQuest* app, or simply go to the website **https://mongoquest.com** which is a free online version of *MongoQuest* that has even more additional features, along with the features we already covered like saving notes. Within *MongoQuest*, you can find the same questions and responses as this chapter, but presented in a more a more interactive way. You can use *MongoQuest,* along with this book, to get the best experience by making use of the chapter references i*n MongoQuest* for each question.

Questions round 1

These questions should be fairly easy to respond to and demonstrate a solid understanding of the basics of MongoDB. Think about this as "round one" in a boxing match, or "level one" in a video game. The questions will get increasingly more challenging in each round. Good Luck!

Question 1: **Can you explain the concept of document embedding, or subdocuments, in MongoDB?**

Response 1: Document embedding is a way of storing related data within a single document, instead of splitting it across multiple collections or databases. This reduces the need for complex joins and improves read performance, at the cost of potentially increased write complexity and document size.

Reference: *Chapter 4, A Better Way to Store Data – Documents*

Question 2: **How do you add or remove fields in a document?**

Response 2: To add a new field to a document in MongoDB, you can use the `updateOne()` or `updateMany()` method with the `$set` operator. To remove a field from a document, you can use the `$unset` operator.

Reference: *Chapter 5, Let's Do It – Create, Update and Delete Documents*, Section: *Updating Documents*

Question 3: **What is the difference between an index and a collection?**

Response 3: A collection is the way MongoDB stores documents together; it is a little bit like a table in a relational database with the key difference being that not all the documents in a collection need to have the same structure, or schema. An index is a data structure that improves the speed of queries by allowing the database to quickly locate and retrieve documents in a collection based on the values of one or more fields.

Reference: *Chapter 9, Planning for Performance – Collections and Indexes*

Question 4: **What is the difference between the `findOne()` and `find()` methods?**

Response 4: The `findOne()` method returns the first document that matches the specified query, while the `find()` method returns a cursor with all the documents that match the query criteria. The `find()` method can be used with additional methods such as `sort()`, `limit()`, and `skip()` to control the number and order of documents returned.

Reference: *Chapter 5, Let's Do It – Create, Update and Delete Documents*, Section: *View Our New Document*

Question 5: **Can you explain the difference between a MongoDB document and a collection?**

Response 5: A document is a single unit of data storage within a collection, while a collection is a grouping of documents that share a similar structure and are stored together.

Reference: *Chapter 9, Planning for Performance – Collections and Indexes*, Section: *Collection Settings and Types*

Questions round 2

The next set of questions delve a little deeper into your knowledge of MongoDB, especially around querying and schema design.

Question 6: **What is the difference between a cursor and a result set?**

Response 6: A result set is the complete set of data that matches a query, returned to the client. A cursor is a pointer to the result set on the server, that allows you to retrieve and process the data in smaller, manageable chunks.

Reference: *Chapter 6, Getting What You Want – Querying*, Section: *Querying MongoDB*

Question 7: **Can you explain the difference between a document-oriented database and a relational database?**

Response 7: A document-oriented database is a type of NoSQL database that stores data in a flexible, document-like format, such as JSON or BSON, and does not require a fixed schema. A relational database stores data in tables with fixed columns and rows, and enforces a strict schema. In a document database, you will usually store related data together in a document, whereas in a relational database, you will store the data in separate tables and join them together based on their relationships.

Reference: *Chapter 1, Why MongoDB?*, Section: *Relational Databases vs Document Databases*

Question 8: **How do you handle schema design with MongoDB?**

Response 8: To handle schema design with MongoDB, it is important to understand your data and its relationships, and denormalize your data wherever appropriate, by storing data that goes together in the same document. Choose appropriate data types for your fields, such as strings for text, numbers for numerical data, and dates for time-related data.

Reference: *Chapter 4, A Better Way to Store Data – Documents*, Section: *What is a Document?*

Question 9: **Can you explain the difference between a BSON document and a JSON document?**

Response 9: BSON and JSON are both formats for representing data in a human-readable and machine-readable way, but BSON is a binary format while JSON is a text-based format. BSON supports additional data types, such as binary data and date objects, and is designed for efficient storage and manipulation of data in MongoDB.

Reference: *Chapter 4, A Better Way to Store Data – Documents,* Section: *What is a Document?*

Questions round 3

How are things going? Are you ready for the next round? We will continue with some more general questions about MongoDB. If you are feeling a bit unsure, that is okay! Make sure to review the referenced chapters and sections to help you feel more confident.

Question 10: How would you query for a document with a field named title, with the value Tacos?

Response 10: You would construct a query like this: **`db.collection.find({ 'title': 'Tacos' })`**. Note the **`'Tacos'`** is capitalized. A query for **`'tacos'`** would not match since MongoDB is case-sensitive.

Reference: *Chapter 6, Getting What You Want – Querying*, Section: *The MongoDB Query API*

Question 11: What is the 'projection' or 'project' part of a query?

Response 11: When querying a collection, you can specify a projection document as the second argument to the **`find()`** method. The projection document contains a list of field names that should be included or excluded from the result set. For example, if you wanted to query for document with the **`title`** 'Tacos' and only return that field, you would write a query like:

```
> db.collection.find({ 'title': 'Tacos' }, { title: 1, _id: 0 })
```

This will show the **`title`** field, and exclude the **`_id`**, which is otherwise already returned.

Reference: *Chapter 6, Getting What You Want – Querying*, Section: *Using Projection to Control Output*

Question 12: Does the casing of text matter in a query?

Response 12: Yes. For strings MongoDB is case sensitive, which means a search for 'Dr. Pepper' is not the same as 'dr. pepper' or 'DR. Pepper'. There are ways around this, such as using **`$regex`** queries or storing your data with search or shadow fields, or using aggregation queries.

Reference: *Chapter 6, Getting What You Want – Querying*, Section: *Query Case Sensitivity*

Question 13: In a Document, what is an ObjectId?

Response 13: In MongoDB, **`ObjectId`** is a built-in data type that represents a unique identifier for a document in a collection. It is a 12-byte value consisting of a 4-byte timestamp, a 5-byte random value, and a 3-byte incrementing counter. By default, MongoDB will assign an **`ObjectId`** to the documents **`_id`**.

Reference: *Chapter 4, A Better Way to Store Data – Documents,* Section: *Structure*

Questions round 4

In these next two rounds, the questions will get a bit more difficult, as they focus on common but more advanced aspects of queries and database administration.

Question 14: What is the difference between the `$and` and `$or` operators in MongoDB?

Response 14: The **`$and`** operator performs a logical 'AND' operation between two or more fields and returns the documents that match. The **`$or`** operator performs a logical 'OR' operation and returns the documents that match at least one of the expressions.

So, the **`$and`** operator requires all the conditions to be met while the **`$or`** operator requires at least one of the conditions to be met.

Reference: *Chapter 6, Getting What You Want – Querying*, Section: *Logical Operators*

Question 15: How do you optimize MongoDB performance?

Response 15: There are many ways to optimize MonogDB performance, the most common is analyzing the usage of indexes. Use tools such as **`.explain()`** and other performance monitoring tools to ensure you have proper indexes to match common, or slower running queries. In some cases, sharding your data can also help with performance.

Reference: *Chapter 9, Planning for Performance – Collections and Indexes*, Section: *Indexing Collections*

Question 16: Can you explain aggregation?

Response 16: Aggregation allows you to perform complex data processing tasks, such as filtering, grouping, sorting, and summarizing data using a set of stages. There are many stages available in the MongoDB aggregation pipeline, such as **`$match`**, **`$group`**, **`$sort`**, **`$limit`**, **`$skip`**, **`$project`** and more. You can use these stages in different combinations to perform complex data processing tasks.

Reference: *Chapter 8, The MongoDB Aggregation Framework,* Section: *Typical Aggregation Pipelines*

Question 17: How would you handle data archiving in MongoDB?

Response 17: You can use tools like **`mongoexport`** to achieve data archiving, or you could use the aggregation framework's **`$merge`** or **`$out`** stages to move documents to a another collection based off a query or other stages.

Reference: *Chapter 10, Getting In and Getting Out – Data Migration,* Section: *Transferring via the Aggregation Framework*

Question 18: Can you explain the use of the **`$regex`** operator?

Response: The **`$regex`** query operator is used to search for documents that match a specific pattern or regular expression. It can be used in queries to find data that matches a certain string pattern, allowing for more flexible and powerful searches including case-insensitive searches.

Reference: *Chapter 6, Getting What You Want – Querying*, Section: *Using Regex Queries*

Question 19: What is the difference between the **`$all`** and **`$elemMatch`** operators?

Response 19: The **`$all`** operator requires an exact match of all the elements in a specified list, while the **`$elemMatch`** operator only requires a match of at least one element in the array that meets the specified condition.

Reference: *Chapter 7, Complex Data, Made Simple,* Section: *Matching Multiple Array Values*

Question 20: What is a MongoDB Replica Set?

Response 20: In MongoDB, a Replica Set is a way to achieve database replication, and thus high availability, by maintaining multiple copies of the data in your MongoDB instance between multiple separate instances of MongoDB. There is a primary instance, and multiple secondary instances. Each instance will have a copy of all the data in the other instances, so that if the primary instance goes down, one of the secondaries can take over, with minimal downtime for the database, and perhaps none to the user.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding,* Section: *Reducing Risk with Replication*

Questions round 5

How are you doing? *This is not a race*; make sure you take the time to look into any questions you are having a harder time responding to or highlight them for further review later. Or maybe things are a breeze so far, great! We have a lot more questions for you.

Question 21: Can you explain the concept of indexes?

Response 21: An index is a data structure that improves the speed of data retrieval operations on a collection. When a query is executed, MongoDB can use an index to quickly locate the documents that match the query criteria, rather than scanning through the entire collection.

Reference: *Chapter 9*, *Planning for Performance – Collections and Indexes*, Section: *Indexing Collections*

Question 22: Can you explain the use of the **`$push`** operator?

Response 22: The **`$push`** operator adds one or more elements to an array field within a document, or to create a new array field if it does not already exist. The **`$push`** operator is often used with other operators, such as **`$each`**, **`$position`**, or **`$slice`** to modify exactly how elements are added to the array.

Reference: *Chapter 7*, *Complex Data, Made Simple*, Section: *Adding Array Items*

Question 23: What are the various data types supported by MongoDB?

Response 23: MongoDB's BSON format supports many common data types, such as string, integer, boolean, null, as well as more JSON like types such as arrays and objects. Other data types include decimal, dates, **`ObjectId`**, regular expressions and javascript code.

Reference: *Chapter 4*, *A Better Way to Store Data – Documents*, Section: *More About Types*

Question 24: What is a Capped Collection?

Response 24: A capped collection is a special collection type that will only ever have a certain number of documents at a time, or to be constrained by a set size on disk. You set these limits when you create the collection. The oldest documents in the collection will be continually and automatically deleted to keep the collection capped at the size set.

Reference: *Chapter 9*, *Planning for Performance – Collections and Indexes*, Section: *Capped Collections*

Question 25: What is the difference between the **`$in`** and **`$nin`** operators?

Response 25: The **`$in`** operator selects documents where the field matches any value in a provided array, while the **`$nin`** operator selects documents where the field does not match any value in the provided array.

Reference: *Chapter 6, Getting What You Want – Querying*, Section: *Comparison Operators*

Question 26: When using the **`$push`** operator, what is the difference between **`$slice`** and **`$position`**?

Response 26: The **`$slice`** operator limits the number of elements that can be inserted into an array, which you can use to limit an array to a certain size. The **`$position`** operator specifies the location in the array where the new element should be inserted.

Reference: *Chapter 7, Complex Data, Made Simple,* Section: *Removing Array Items*

Questions round 6

While having responses, and knowing facts is important in an interview situation, being able to communicate well is even more important. How do you feel you are doing in that respect? Can you explain your responses easily, and clearly? Can you perhaps state the same thing a couple different ways? Give it a try with some of the questions you have read already or with the ones in the next section.

Question 27: What is the difference between a single-field index and a compound index in MongoDB?

Response 27: A single-field index is an index on one particular field of the documents in a collection, while a compound index is an index on multiple fields of the documents. Single-field indexes can be used to optimize queries that filter or sort on a single field, while compound indexes optimize queries which perform operations on multiple fields within a query.

Reference: *Chapter 9, Planning for Performance – Collections and Indexes*, Section: *Indexing Collections*

Question 28: What is the difference between the **`$and`** and **`$nor`** operators?

Response 28: The **`$and`** operator in MongoDB performs a logical 'AND' operation whereas the **`$nor`** operator performs a logical 'NOR' operation, meaning it returns the documents that do not match any of the expressions. In short, the **`$and`** operator requires all conditions to be

met while the **`$nor`** operator requires that none of the conditions are met.

Reference: *Chapter 6, Getting What You Want – Querying*, Section: *Logical Operators*

Question 29: What is the difference between the **`$pull`** and **`$pop`** operators?

Response 29: The **`$pull`** operator removes all instances of a value from an array in a document, based on a query. The **`$pop`** operator removes the first (or last) element from an array a document.

Reference: *Chapter 7, Complex Data, Made Simple,* Section: *Removing Array Items*

Question 30: What is the difference between a TTL index and a capped collection?

Response 30: A **Time To Live** (**TTL**) index is linked to a specific date field in a document and an expire date (in seconds). When the expire date is met on the date field, that document will be deleted. In a capped collection documents are deleted when the size on disk is exceed, or optionally, a count of documents. When either the size or the count of documents is exceeded, the oldest documents will be deleted from the collection.

Reference: *Chapter 9, Planning for Performance – Collections and Indexes*, Section: *Capped Collections*

Question 31: What is the difference between the **`mongodump`** and **`mongoexport`** commands?

Response 31: The **`mongodump`** and **`mongoexport`** commands are both used for exporting data from a MongoDB database, but they differ in the format of the exported data.

`mongodump` creates a binary backup of the entire database or a specific collection in a binary BSON format, which can be used to restore the data to another MongoDB instance using the **`mongorestore`** command.

`mongoexport` exports data from a MongoDB collection in JSON, CSV, or TSV format. This command is typically used to export data for use in other applications or systems that do not use MongoDB. Unlike **`mongodump`**, **`mongoexport`** only exports the data from a single collection at a time and does not include indexes or other metadata.

Reference: *Chapter 10, Getting In and Getting Out – Data Migration,* Section: *Using Database Export Tools*

Question 32: How can you tell if you are on the primary node of a Replica Set?

Response 32: If you are connected via the MongoDB shell, this will generally be in your shell prompt with something like **`:PRIMARY>`** or **`:SECONDARY>,`** or you can use the **`rs.status()`** or **`rs.hello()`** commands and look for **`primary`** key which will have an object with this information.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding,* Section: *Changing a Primary to a Secondary*

Questions round 7

These last few sections comprise the most difficult questions, but you should still have a pretty good response for them after reading this book.

They are by no means the most advanced questions you might be asked for a role needing a lot of experience with MongoDB, but if you can answer these, you probably have a pretty solid understanding of MongoDB and should feel confident in your skills.

Question 33: Can you explain the role of a replica set arbiter in MongoDB?

Response 33: The role of a replica set arbiter in MongoDB is to assist in the election of a primary node in the event of a network partition or the failure of one of the nodes. The arbiter does not store any data, and its sole purpose is to participate in the election process by casting a tie-breaking vote if necessary.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding,* Section: *Arbiters*

Question 34: How do you monitor MongoDB performance?

Response 34: You can use commands like **`db.serverStatus()`** or **`db.dbStats()`** in the MongoDB shell to monitor database metrics such as memory, CPU, and network usage. Additionally, the **`mongostat`** command line tool can be used to monitor various performance metrics of a MongoDB server in real-time.

Reference: *Chapter 11, Make It Great – Configuration and Monitoring,* Section: *Monitoring MongoDB*

Question 35: How can you ensure data security with MongoDB?

Response 35: Data security with MongoDB can be ensured through various mechanisms such as authentication, authorization, encryption, auditing, and following best practices for database security.

Reference: *Chapter 13, Being Proactive – Security and Backups*

Question 36: What is the role of a secondary node in a replica set?

Response 36: The role of a secondary node is to maintain a copy of the primary's data, and if configured to so, vote in elections to elect a new primary and take over a primary, if elected. It can also receive reads, depending on the settings of the connection and the replica set configuration.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding,* Section: *Replica Sets*

Questions round 8

Great job, how are you feeling? Again, make sure to use these questions as a means to review what you have learned and think about how you would explain these concepts in a way that feels comfortable to you; that is why this is a question and a response, not a question and answer. Make your responses your own.

Let us continue.

Question 37: Can you explain the difference between a primary key and a shard key in MongoDB?

Response 37: A primary key is a unique identifier for each document in a collection, while a shard key is a field or set of fields that is used to break up data across multiple shards in a sharded cluster.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding,* Section: *Sharding Data and Shard Keys*

Question 38: How would you handle failover in MongoDB?

Response 38: To handle failover in MongoDB, you can set up a replica set with multiple nodes, where one node is designated as the primary and the others as secondaries. If the primary node fails, one of the secondaries will be automatically elected as the new primary, and the application can continue to operate without interruption. You can also manually handle failover by using the `rs.stepDown()` command and/or changing the priority of nodes.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding,* Section: *Changing a Primary to a Secondary*

Question 39: What is sharding in MongoDB, and how does it work?

Response 39: Sharding is the concept of breaking up the data in your collection across multiple different servers. It works by choosing a field to break up your data on, called a shard key. Using this key, MongoDB will automatically break up and balance your data between different servers, which should be using Replica Sets.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding*, Section: *Scaling with Sharding*

Question 40: How can you configure and extend the MongoDB Shell, mongosh?

Response 40: You can extend the shell in a number of ways including configuring and adding custom configuration and code to your **`mongoshrc`** file. There you can configure things such as your prompt, custom functions and even node.js code and imports.

Reference: *Chapter 15, Tools for Success – MongoDB Shell and Compass UI*, Section: *MongoDB Shell*

Questions round 9

We have arrived at the next to last round. Some of these concepts might be a bit less reliant to your role or position, so feel free to skip things if you are sure, but it is recommended to at least having a decent idea of how to respond to these questions.

Having a response along the lines of *"I have general idea about that, such as …"* is much more impressive than no response.

Question 41: What is the difference between a replica set and a sharded cluster?

Response 41: A Replica Set has members which each have a copy of the primary server's data. A sharded cluster has data split up between different replica sets, so no one replica set has a full copy of the server's data, but rather the slice, or shard of the data they are responsible for storing. These slices are automatically balanced by MongoDB based on a shard key.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding*, Section: *Scaling with Sharding*

Question 42: How do you back up and restore MongoDB data?

Response 42: There are many ways to backup and restore MongoDB data. The simplest is simply backing up the folder on disk with the database

files. However, more than likely, you will want to use a tool such as **`mongodump`** or **`mongoexport`** along with **`mongorestore`** or **`mongoimport`**. You can use those tools, along with custom scripts to backup databases and collections on a regular schedule as well restore from backups.

Reference: *Chapter 13, Being Proactive – Security and Backups.* Section: *Backup Strategies*

Question 43: How do you implement role-based access control in MongoDB?

Response 43: You can implement role-based access control by setting the authentication and authorization settings in your MongoDB config file or by using the **`--auth`** option. You will also need to create an administrator user account. After that, you can create more users using **`createUser()`**, create and define roles using **`createRole()`** and lastly, use **`grantRolesToUser()`** to assign roles.

Reference: *Chapter 13, Being Proactive – Security and Backups.* Section: *Enabling Access Control*

Question 44: How do you handle query optimization in MongoDB?

Response 44: To optimize queries in MongoDB, you can use the **`.explain()`** method to get information about how a query is executed and identify potential performance issues. You can also create indexes on fields that are frequently used in queries or use the aggregation pipeline to optimize complex queries.

Reference: *Chapter 9, Planning for Performance – Collections and Indexes,* Section: *Query Plans*

Question 45: Can you explain the concept of document validation in MongoDB?

Response 45: Schema validation enforces a schema on all documents inserted into the collection. For example, you can force a field to only ever be a number, or string, or required. You can provide a **`$jsonSchema`** object to the validator option when you create your collection to configure the collection's schema.

Reference: *Chapter 9, Planning for Performance – Collections and Indexes,* Section: *Document Schema Validation*

Questions round 10

You made it! This is the last round. Do you feel you have already knocked out everything, or this decision might go to judges? Let us finish this strong, with the last few questions.

Question 46: What is the **`oplog`** in MongoDB?

Response 46: The **`oplog`** in MongoDB is a special system collection that records all the write operations that occur on a MongoDB server in chronological order. The **`oplog`** is used in MongoDB's replication and sharding features, in order to ensure that all replica set members or shards have the same data as the primary member or shard.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding,* Section: *The oplog Collection*

Question 47: Can you explain the difference between a horizontal and vertical scaling using MongoDB?

Response 47: Vertical scaling involves adding or upgrading to more resources to a single server, such as CPU, RAM or disk. Horizontal scaling involves adding more servers or nodes to distribute the load across multiple servers.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding*

Question 48: Can you explain the concept of a Read Preference in MongoDB?

Response 48: *Read Preference* refers to the strategy used to determine which member in a Replica Set should be used for read operations. It can be specified as an option when performing read operations such as a **`find()`** query to determine how a replica set should distribute read operations across the members in the replica set. This includes options like **`Primary`**, **`Secondary`**, **`PrimaryPreferred`**, **`SecondaryPreferred`** and **`Nearest`**.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding,* Section: *Fail-safes*

Question 49: Can you explain the role of the primary node in a replica set?

Response 49: The primary node in a Replica Set is responsible for all write operations and manages all client connections to the Replica Set. It replicates all its data and any changes to the secondary nodes in the replica set.

Reference: *Chapter 12, Seamless Scaling – Replication and Sharding*, Section: *Replica Set*

Question 50: Can you explain the use of the `$group` operator?

Response 50: The **`$group`** operator is used to group documents in a collection, based on a specified set of fields, and then perform aggregate functions on the grouped data. The **`$group`** operator takes an aggregation pipeline stage, and is typically used in conjunction with other pipeline stages such as **`$match`**, **`$project`**, and **`$sort`**.

It can also take one or more accumulator expressions, which are used to perform aggregate functions on the grouped data. Some examples of accumulator expressions are **`$sum`**, **`$avg`**, **`$min`**, **`$max`**, and **`$push`**. The result of the **`$group`** operator is a new collection that contains the grouped and aggregated data, which can then be further processed or analyzed as needed.

Reference: *Chapter 8, The MongoDB Aggregation Framework*, Section: *Grouping and Sorting Stages*

Conclusion

You made it! We covered a lot of area of MongoDB, and it is just fine if you feel like you missed some things. Make sure you review any questions you did not feel you could respond to well. It is also fine if you skipped some, but make sure to try your best to at least have a general idea how to respond when asked.

If you are in the "questing" mood, remember you can "play" the *MongoQuest* app either with your own setup, or via **https://mongoquest.com**

Either way, take your time and build up your confidence either for an interview, or just a fun way to test your comprehension of what you learned in this book. In the next chapter, we will wrap things up and touch on a few topics related to MongoDB you might be interested in learning more about.

Chapter 19
Conclusion

"You can't go back and change the beginning, but you can start where you are and change the ending."

— C.S. Lewis

Introduction

In this final chapter, we will wrap up a few concepts, as well as touch upon a couple of areas that we did not cover in detail in the previous chapters, but that might be of interest to you nonetheless. For example, how can we handle the concept of database transactions in MongoDB? How can we easily convert SQL type queries in MongoDB queries? Or how can we store files directly within the database? While we will not be going in-depth on these topics, a few examples will be provided along with links to find out more.

Structure

We will discuss the following topics:

- Change Streams
- Transactions
- Storing files with GridFS
- SQL to MongoDB

Objectives

The goal of this chapter is to briefly introduce some more advanced MongoDB topics that did not quite fit into other chapters. By the end of this chapter, you should have a high level idea of how to look out for changes in your database, known as *Change Streams*, and code side effects to happen when those changes occur, as well as how to use database transactions to provide ACID compliance, and how to store files in MongoDB using GridFS. Lastly, we will discuss and show some examples of how to translate SQL queries into MongoDB queries.

For the following sections, we will generally provide partial code samples, meaning most of the code will not include things like connecting to the database and so on, for brevity. However, there are a number of more robust example in the book's GitHub repository.

Change Streams

Change Streams in MongoDB allow you to listen for changes that occur in a MongoDB database or a specific collection. It provides a way to monitor the database in real-time and receive notifications when changes happen, such as document inserts, updates, deletes, or even changes in the database configuration.

Change Streams use the MongoDB's **`oplog`**, the same internal collection used in replica sets to keep a record of all changes happening in a MongoDB database. By leveraging the **`oplog`**, Change Streams can provide a reliable and efficient way to track changes and propagate them to applications or services.

Subscribing to Database Changes

Following are a few examples, coded in Node.JS, to use Change Streams. In short, they are small bits of code that you can run, as an ongoing program, which will connect to your MongoDB server and watch for changes. To keep things succinct, we will show, for the most part, we have shown only the portion of code needed to subscribe to the database's Change Stream. However, you can find fuller examples in the book's GitHub repository. As a very basic example, here is how we can watch for any changes within our collection:

```
1   const collection = db.collection('recipes');
2   const changeStream = collection.watch();
3
4   changeStream.on('change', (change) => {
5     console.log({ change });
6   });
```

This code creates a Change Stream on the **`recipes`** collection and listens for any changes that occur. The **`changeStream.on('change')`** callback is triggered for each change that happens in the collection. If we want to be more particular about what sort of changes, we watch for we can modify the code slightly, as shown:

```
1   const collection = db.collection('recipes');
2
3   const changeStream = collection.watch([
4     { $match: { operationType: 'insert' } }
5   ]);
6
7   changeStream.on('change', (change) => {
8     console.log({ change });
9   });
```

This will "filter" the changes we are watching for; in this case **`insert`** operations, using the "pipeline" format just like the *Aggregation Framework*, by using **`$match`** on line **`4`**.

Triggering Actions with Change Streams

In the examples so far, we have simply been outputting what the change was, which is not very useful. However, subscribing to these change streams can be very powerful when paired with code that can trigger other actions. For example, you could update a separate document, or you could trigger sending an email, or you could make a call to an internal or external API.

In the following, slightly more complex example, we are matching both on the type of **`insert`** and on a field value in the document that is being inserted, on line **`7`**.

```
1    const users = db.collection('users');
2
3   changeStream.on('change', (change) => {
4     const { operationType, fullDocument } = change;
5
6     // If a new breakfast recipe is inserted
7     if (operationType === 'insert' && fullDocument.type === 'break-
    fast') {
8       const { userId } = fullDocument;
```

```
9
10        // Update the user document with the new favorite break-
    fast recipe
11        users.updateOne(
12          { _id: userId },
13          { $set: { favoriteBreakfast: fullDocument.title } }
14        )
15        .then(() => {
16        console.log('Favorite breakfast recipe updated for user:', use-
    rId);
17        });
18      }
19    });
```

Using this, we can update a document in the users collection when a new document is inserted in the recipes collection. This is not limited to simply updating one document; you can perform more complex operations, like in the following example where we are inserting a user's id in an array on our recipe document, when the user adds that recipe's id to their liked recipes array in the user document.

```
1     const users = db.collection('users');
2     const recipes = db.collection('recipes');
3
4     const changeStream = users.watch([
5      { $match: { operationType: "update" } },
6       { $match: { "updateDescription.updatedFields.likes": { $ex-
    ists: true }}},
7     ]);
8
9     changeStream.on("change", async (change) => {
10      const { fullDocument, updateDescription } = change;
11
12      const userId = fullDocument._id;
13      const recipeId = updateDescription.updatedFields.likes;
14
```

```
    const result = await recipes.updateOne(
      { _id: recipeId },
      { $addToSet: { likes: userId } }
    );
});
```

This can be a very powerful way to keep documents in other collections up-to-date automatically. Another great way to use Change Streams is to use the change to make a call to an API, like the following example:

```
1   const axios = require('axios');
2
3   changeStream.on('change', (change) => {
4     const { fullDocument } = change;
5
6     // If a recipe is deleted, trigger an API call
7     if (change.operationType === 'delete') {
8       const recipeId = fullDocument._id;
9
10      // Make an API call to notify the external API
11      axios.post('https://api.example.com/track/deletes', { recipeId })
12        .then(() => {
13          console.log('Notification sent for deleted recipe: ', recipeId);
14        })
15    }
16  });
```

Notice we are watching, on line **7** for delete operations and sending a request to an API endpoint on lines **11**-**14**.

Running Change Streams

As mentioned earlier, you can run a script as an ongoing program that watches for a change stream event, as we are in the following example:

```
const { MongoClient } = require('mongodb');

async function runChangeStream() {
 try {
    const client = await MongoClient.connect('mongodb://local-
host:27017');
    const db = client.db('cookbook');
    const collection = db.collection('recipes');

    // Change Stream for inserts
    const changeStream = collection.watch(
      [{ $match: { operationType: 'insert' } }]
    );

    changeStream.on('change', async (change) => {
      const newTitle = change.fullDocument.title;
      await collection.updateOne(
        { _id: '123' },
        { $push: { new_recipes: newTitle } }
      );
    });

    // Wait indefinitely
    await new Promise(() => {});

    changeStream.close();
    client.close();
  } catch (error) {
    console.error('Error:', error);
  }
}

// Run the change stream
runChangeStream();
```

This will watch for newly inserted documents in the recipes collection, on lines 10-12 and when a new document is inserted, take the title of that document and append it to an array called **`new_recipes`** in a document with the **`_id`** of **`123`** on lines **`14`**-**`20`** using **`$push`**.

To run the Change Stream, assuming you have node and the drivers installed, simply run:

```
$ node myChangeStream.js
```

Since we are awaiting a **`Promise`** indefinitely on line **`23`**, this script will keep running as long as the connection is maintained, and the process running the script is not stopped. You can find out a lot more about Change Streams at the following link:

https://www.mongodb.com/docs/manual/changeStreams/

Transactions

If you have worked with databases before, you may be familiar with the concept of transactions. Transactions, simply stated, are a way to group some related actions together and ensure they all happen as a single unit of work. If anything goes wrong while attempting those operations, all the operations in that group can be "rolled back" or canceled. This process ensures that a number of database concepts are adhered to: **atomicity, consistency, isolation, and durability** (**ACID)**.

This can be very important for applications were a batch of operations need to happen together, such as two documents being inserted as a pair, or updating a document when another document is inserted, as well as handling things like retrying the query operations if they fail.

Transactions in mongosh

There are a number of syntactical ways to accomplish this in your code, or even from **`mongosh`**, since it is a JavaScript shell. For example, if we needed to insert two documents at the same time, we could use a transaction within **`mongosh`** like the following query:

```
1    const session = db.getMongo().startSession();
2
3   session.startTransaction();
4
5    try {
6     const collection = session.getDatabase('cookbook').recipes;
```

```
7
8       collection.insertOne({ title: 'Carrot Soup' });
9       collection.insertOne({ title: 'French Toast' });
10
11
12      session.commitTransaction();
13
14    } catch (error) {
15      session.abortTransaction();
16
17      print('Transaction aborted:', error);
18    } finally {
19      session.endSession();
20    }
```

Notice, from the very beginning, we have created and started a **`session`**, on lines **1** and **2**. Then, we start a try/catch/finally block where we run two inserts for new recipe documents, and then let MongoDB know that we are ready to try to "commit", or run this transaction with **`commitTransaction()`** on line **12**. In essence, we have queued up the two insert operations and now will run them, together.

If there is a problem, we can handle that on lines **14**-**17** and finally, either way, end our session on line **19**.

Transactions in Code

When using a support programming language, transactions work much the same. For the next examples, we will use Node.JS, as shown in the following transaction code:

```
1     const { withTransaction } = require('mongodb');
2
3     const session = client.startSession();
4
5     await withTransaction(session, async () => {
6     const collection = session.client.db('cookbook').collection('rec-
    ipes');
7
```

```
    await collection.insertOne({ title: 'Carrot Soup' });
    await collection.insertOne({ title: 'French Toast' });
});

session.endSession();
```

Note the use of **`withTransaction()`** on line **5**, which handles things such as "committing" the transaction for us. Next, to create a more complex transaction, we can again use a try/catch/finally as in the following example program:

```
const { MongoClient } = require("mongodb");

async function runTransaction() {
  const uri = "mongodb://localhost:27017";
  const client = new MongoClient(uri);

  try {
    await client.connect();
    const session = client.startSession();

    await session.withTransaction(async () => {
      const recipes = session.client.db("cookbook").collection("recipes");

      // Check if a document exists in the collection
      const existing = await recipes.findOne({title: "Chicken Tikka"});

      if (existing) {
        console.error("Transaction aborted: Document already exists");
        session.abortTransaction();
        return;
      }

```

```
23          // If there is no error, perform operations
24          await recipes.insertOne({ title: "Carrot Soup" });
25          await recipes.insertOne({ title: "French Toast" });
26        });
27      } catch (error) {
28        console.error(error);
29      } finally {
30        session.endSession();
31        await client.close();
32      }
33    }
34
35    runTransaction();
```

With this logic, which we put inside a function, the code in our **`try`** block on lines **`14`**-**`21`** is checking to see if a document already exists with a particular **`title`**, and if so, aborts the transaction and exits the block. This will cause the **`insertOne`** operations on lines **`24`**-**`25`** to not run. Otherwise, they will insert as planned.

There are many other options for transactions, and they can be used with inserts, updates, deletes and so on. To find out more about transactions, check out the official docs.

https://www.mongodb.com/docs/manual/core/transactions/

Storing Files with GridFS

As mentioned previously, documents themselves have a size limit of 16 megabytes. But that does not mean we cannot store data larger than 16 megabytes in MongoDB. To get around this limit, MongoDB has a file store system called GridFS which is built on top of a normal MongoDB database.

GridFS allows you to store and retrieve files as chunks, breaking them into smaller pieces for efficient storage and retrieval. It does this by using two collections, one for the file's *metadata* and another for the *chunks* of data that make up the file.

Files are divided into small, evenly sized chunks and stored as separate documents in the **`chunks`** collection. The so called "metadata" about the file, such as the file name, type, and other optional information, is stored in the **`files`** collection. Refer to *Figure 19.1:*

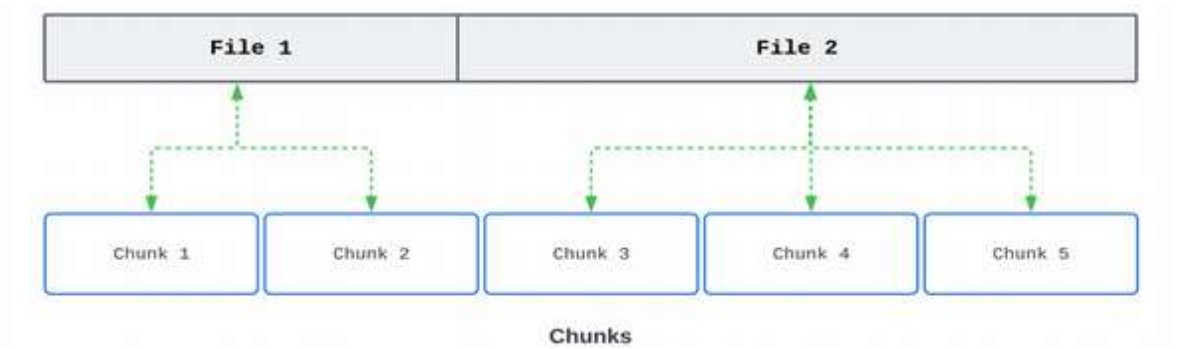


Figure 19.1: *GridFS File Chunks*

GridFS automatically handles the separation, storage, and streaming back of your files, meaning that you will not need to deal with that complexity yourself.

By using GridFS, you can efficiently store and retrieve files of virtually any size in MongoDB. It is particularly useful for scenarios such as storing image or video files, large documents, or any type of data that exceeds the 16-megabyte document size limit.

Using mongofiles

The most straight forward way to use GridFS is by using the **`mongofiles`** binary, which is available as part of the MongoDB Database Tools which we discussed earlier in this book. If you did not already install the Database Tools, you can find out more at the following link:

https://www.mongodb.com/docs/database-tools/

The command works by accepting a parameter with the database that holds the files, as well as other parameters as subcommands. For example, to add a file to GridFS, you can use the **`put`** subcommand as shown:

```
$ mongofiles --db myFiles put example.jpg
```

This would add a file called **`example.jpg`** to a database called **`myFiles`**. You can use whatever database you wish, but it is generally recommended to use a separate database for GridFS specifically. To list the files in your GridFS database, you can use the **`list`** subcommand:

```
$ mongofiles --db myFiles list
```

To write back out of the data you can use the get subcommand:

```
$ mongofiles --db myFiles get example.jpg
```

Lastly, to remove a file, you can use delete, which will remove the file and all its chunks from the database:

```
$ mongofiles --db myFiles delete example.jpg
```

Using Code

More than likely, you will want to work with GridFS with a programming language, often times as part of an application. Most of the drivers for programming languages, which we have covered in the previous chapters, support GridFS. For example, using Python and the **pymongo** driver, you can add and receive a file with a few lines of code, as shown:

```
1   from pymongo import MongoClient
2   from gridfs import GridFS
3
4   # Connect to MongoDB
5   client = MongoClient('mongodb://localhost:27017')
6   database = client['images']
7   fs = GridFS(database)
8
9   # Store image in GridFS
10  with open('example.jpg', 'rb') as file:
11      file_id = fs.put(file, filename='example.jpg')
12
13  print(f"Image uploaded, ID: {file_id}")
14
15  # Retrieve image from GridFS
16  out_file = fs.get(file_id)
17  with open('retrieved.jpg', 'wb') as file:
18      file.write(out_file.read())
```

We are setting our GridFS database on line **7**, and then adding a file on lines **10**-**11**. Lastly, we can retrieve the file out and write it to disk on lines **16**-**18**.

A common use case might be to store and retrieve images from GridFS; you can get an image file out of GridFS and output as a file image file with code, as shown:

```
1   <?php
2
3   $mongo = new MongoDB\Driver\Manager("mongodb://localhost:27017");
4
5   $filename = "example.jpg";
```

```
6    $options = ["bucketName" => 'fs'];
7
8    $bucket = new MongoDB\GridFS\Bucket($mongo, 'files, $options);
9    $stream = $bucket->openDownloadStreamByName($filename);
10
11   // Output the file as an image
12   header("Content-Type: image/jpeg");
13   fpassthru($stream);
14   exit;
15
16   ?>
```

Notice how we retrieved the file on lines **8**-**9** and then set the header on line **12**, before outputting the data of the file on like **13**. MongoDB will automatically handle streaming back the chunks of the file for you, it is that simple. You can read more about GridFS at the following link:

https://www.mongod.com/docs/manual/core/gridfs/

SQL to MongoDB

Following is a very brief table of SQL to MongoDB examples, which you can use to review what you have learned, or as a reference. There are more examples and explanations if you follow the given link:

https://www.mongodb.com/docs/manual/reference/sql-comparison/

SQL	MongoDB
`SELECT * FROM recipes`	`db.recipes.find()`
`SELECT title, desc FROM recipes`	`db.recipes.find({}, { _id: 0, title: 1, status: 1} )`
`SELECT * FROM recipes WHERE title = 'Toast'`	`db.recipes.find({ title: 'Toast' })`
`SELECT COUNT(*) FROM recipes`	`db.recipes.find().count()`
`SELECT * FROM recipes LIMIT 10`	`db.recipes.find().limit(10)`

SQL	MongoDB
`SELECT * FROM recipes WHERE cook_time < 30`	`db.recipes.find({ cook_time: { $lt: 30 }})`
`SELECT * FROM recipes WHERE title LIKE 'taco%'`	`db.recipes.find({ title: { $regex: /^taco/ }})`
`INSERT INTO recipes(title, cook_time)` `VALUES ("Tacos", 45)`	`db.recipes.insertOne({ title: 'Tacos', cook_time: 45 })`

`UPDATE recipes`
`SET type = 'Second Breakfast'`
`WHERE type = 'Breakfast'`
	`db.recipes.updateMany({ type: 'Breakfast' }, { $set: { type: 'Second Breakfast' }})`

`CREATE TABLE recipes (`
`  id INT NOT NULL AUTO_INCREMENT,`
`  title VARCHAR(100),`
`  cook_time SMALLINT,`
`  PRIMARY KEY (id)`
`)`
	`db.recipes.insertOne({ title: 'Tacos', cook_time: 45 })`

`DROP TABLE recipes`	`db.recipes.drop()`

Table 19.1: *SQL to MongoDB examples*

Conclusion

We have covered a lot of topics in this book, and yet there is so much more to learn! Thank you for following along with us, and we hope you have learned a lot. Good luck on your journey, be it a quest for a new job or position, or just learning a new skill. Stay curious!

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

A

administration, replica set 213
  disaster recovery 215, 216
  existing replica set, changing 213
  fail-safe 213-215
  maintenance 215
  monitoring 216
Aggregation Framework 111
  case sensitive searching and sorting 131, 132
  pipeline 112
  uses 131
aggregation pipelines
  aggregations, in MongoDB Compass 114-116
  building 112
  fields, projecting 113, 114
  making readable 122, 123
  matching documents, filtering 119-121
  MongoDB Compass pipeline features 116
  operators, combining 117, 118
  stages 112
  stages, grouping 123
  stages, sorting 123-125
  stages, using multiple times 121, 122
aggregation pipelines, for better searches
  documents, updating 136, 137
  stages 132-136
application-level encryption 237, 238
arrays 52, 90
  array order 99, 100
  example documents 97
  mixed data type arrays 100, 101
  multiple array values, matching 100

querying 99
query operators, using 101, 102
using 96
array update operators 104
array items, adding 105
array items, removing 107-110
array items, sorting 106, 107
example documents 104
multiple items, appending 105, 106
Atlas App
anonymous Realm users 320, 321
creating 318-320
database user access 320
User Create function 321-325
Atlas Database 316, 317
data, importing 317, 318
question example 316, 317
Atomic Operation 87
authentication 222, 223
access control, enabling 223
authorization, on Docker 224
localhost exception 223, 224
logging in, with user 226, 227
replica sets 228
types 227
users, creating 224, 225
users, on different databases 227, 228
authorization 228, 229
privileges 229
roles 229, 230
user-defined roles 230, 231

B

backups, restoring 235
example script 236
MongoDB Database tools 235, 236
via MongoDB data files 235
backup strategies 232
example backup script 233, 234
filesystem backups 232, 233
MongoDB Database tools 233
MongoDB services 235
Bash 272
business intelligence analyst 23

C

C# 265, 266
C++ 268
capped collection 153
case insensitive index
creating 147, 148
Change Streams 378
actions triggering with 379-381
database changes, subscribing 378, 379
running 381-383
charts 308
creating 309-311
embedding 312
sharing 312
Client-Side Field Level Encryption (CSFLE) 238
cloud tools
Atlas CLI 308
Data Lake 307
device sync 307
Search 304-306
triggers 306
Codespace 240
collection maintenance 159
collections, deleting 161

collection statistics 159-161
collections
capped collection 153, 154
files, storing with GridFS 155
indexing 140
settings 152
time-series collection 154, 155
types 152, 153
common options, configuration file
logging options 185, 186
network options 184, 185
process management options 186
replication 186, 187
security options 187
storage options 183, 184
Windows service options 187
complex documents
creating 64
MongoDB Shell, using as JavaScript Shell 64, 65
complex pipelines 125, 126
creating 126
example 127, 128
stages 129-131
compound indexes 143
computer databases 6, 7
configuration file
common options 183
externally sourced config 188
file format 182
server defaults 182
configuration, MongoDB server
binary data 180
command line 181, 182
data files 181
Context Hook 334
CRUD (Create Read Update Delete) 59
cursor 76

D

data history 4
database administrator 6, 23
Database as a Service (DBaaS) 296
database encryption 237
application-level encryption 237, 238
network encryption 237
database export tools 173
database import tools 166
MongoDB database tools 166
databases 4
computer databases 6, 7
of clay 4, 5, 6
relational databases 7
Database Services 296
backups 303
cluster 296-298
connecting, to Atlas cluster 301
Data API 301-303
data, editing 298
data, viewing 298
Multi-Cloud Database Services 296
network access 299, 300
serverless 298
users 299
data engineer 22
data export
database export tools, using 173
mongodump command, using 174
mongoexport command, using 173, 174

mongorestore command, using 174
performing 172
via MongoDB Compass 172
data import
bulk inserts 172
database import tools, using 166
into non-empty collections 169
JSON, importing from API 169-171
MongoDB database tools, using 166
mongoimport command, using 167-169
performing 164
via MongoDB Compass 164-166
Data Lake 307
data scientists 23
data transfer
documents, archiving 175, 176
performing 175
via Aggregation Framework 175
dates 50, 51
epoch dates 51
importing/exporting 52
dbStats command 189
Device Sync 307
DevOps engineer 23
document databases
versus, relational databases 7-9
Documents 43, 47, 48
complex documents, creating 64
considerations 48
creating 60
creating, MongoDB Shell used 60-62
deleting 71
example documents, importing 44
examples 52
fields, adding 67, 68
fields, removing 68
finding, by Document Field 63
finding, by ObjectId 63
importing on command line, with mongoimport 47
importing, with MongoDB Compass 44-46
inserting 62
MongoDB Shell commands 50
multiple documents, inserting 65
multiple documents, inserting with Compass 66, 67
multiple documents, updating 68, 69
one document, finding 63, 64
querying 63
structure 49
updating 67
updating, with Compass 70, 71
upsert 69, 70
viewing 62, 63
document schema validation 156
basic schema validation 156, 157
in MongoDB Compass 158
Documents, examples 52
home sale listing 57, 58
recipe 55, 56
stock data 53, 54
user profile 54

E

element operators 89
field type considerations 89, 90
embedded documents
exact matches 102, 103
matching 103, 104

querying 102
using 96
epoch dates 51
example documents
importing 74

F

field update operators 85, 86
files storing, with GridFS 386, 387
code, using 388, 389
mongofiles, using 387
full-stack developer 22

G

geospatial indexing 150
Go 263-265
GridFS 386

H

hierarchical database 6

I

indexes
case insensitive indexes 147, 148
compound indexes 143
creating 141
creating, in MongoDB Compass 142
deleting 152
example 141
explain(), using with 145, 146
geospatial indexes 150
hiding 151
maintaining 150, 151
modifying 152
naming 142, 143
on arrays 143
query plans 144, 145
Time To Live (TTL) indexes 149
unique indexes 143, 144
wildcard indexes 148, 149
interview questions 359-375

J

job roles 21
business intelligence analyst 23
database administrator 23
data engineer 22
data scientists 23
DevOps engineer 23
full-stack developer 22
technical consultant 24
technical writer 24
jobs
example interview questions 24
JSON
importing, from API 169

K

Kotlin 267

L

language examples
Bash 272
C# 265, 266
C++ 268
Go 263-265
Kotlin 267
Perl 271
Ruby 270
Rust 268
Scala 272
Swift 270
logical operators 87, 88

M

member roles, replica set 210
  arbiters 212
  delayed nodes 212
  hidden nodes 211
  priority 210, 211
  voting nodes 211
MongoDB 20, 21
  charts 308
  features 1, 15
  future jobs 24
  installing 28, 29
  monitoring 188
  prerequisites 28
  programming 240
  querying 76
  setting up, on MongoDB Atlas Cloud 39-42
MongoDB Compass 285
  advanced connection options 285, 286
  Aggregation Framework 288, 289
  Aggregation pipelines 288, 289
  collections, creating 287
  Explain Plan tab 291, 292
  performance monitoring 292, 293
  queries 75, 287
  queries, exporting 288
  schema 290, 291
MongoDB on Docker
  connecting 37
  connecting, via MongoDB Compass 38
  connecting, via MongoDB Shell mongosh 37, 38
  database data files, persisting 37
  installing 36
  server, running 36, 37
  server, running via Docker Compose 38, 39
MongoDB on macOS
  installation 33
  server, connecting via Compass 35, 36
  server, installing 33
  server, running 34, 35
  shell, installing 34
  tools, installing 34
MongoDB on Windows
  Compass, installing 31
  installation 29
  server, configuring 32, 33
  server, installing 30, 31
  tools, installing 32
MongoDB Query API 76, 77
  count(), using 82
  filter, for matching documents 77
  limit(), using 83
  projection, for controlling output 77-79
  skip(), using 83
  sort(), for ordering output 79-81
  variables, using in queries 81, 82
MongoDB Query Operators
  Atomic Operations 87
  comparison operators 83, 84
  element operators 89, 90
  field update operators 85, 86
  logical operators 87, 88
  using, in updating queries 85
MongoDB Server configuration 180

MongoDB Server operations
  binary 179
  ports 179, 180
  starting 178
  stopping 179
MongoDB Shell 276
  commands 50
  configuration 276
  editor mode 276, 277
  Node.JS Scripting 277-282
  queries 74
  snippets 282
  versus, MongoDB Compass 74
mongodump command 174
mongoexport command 173
mongoimport command 167
MongoQuest 361
mongorestore command 174
mongostats command 193, 194
mongotop command 192, 193
monitoring, MongoDB 188
  command line tools 192-194
  database statistics 189-192
  software monitoring 194, 195

N

Near Earth Objects (NEO) 170
network encryption 237
Node.JS and MongoDB 247
  Aggregation Framework 252-254
  database connection, with Node.JS 247-249
  document, inserting 251, 252
  MongoDB Node.JS driver, installing 247
  Mongoose, using with MongoDB 254, 255
  query options 249-251
numbers 50

O

objects 52, 90

P

Perl 271
PHP and MongoDB 256
  Aggregation Framework 260, 261
  database connection, with PHP 257, 258
  document inserting 259, 260
  MongoDB driver, installing 256
  MongoDB Laravel package, using 262, 263
  MongoDB PHP library, autoloading 256
  query options 258, 259
programming, with MongoDB 240
  code examples 240
  Codespace 240, 241
Python and MongoDB
  Aggregation Framework 246, 247
  database connection 242
  document, inserting 244-246
  PyMongo library, installing 241
  queries, performing 243
  query options 243
  URL 241

Q

Queryable Encryption 238
query case sensitivity 91
  casing issues, dealing with 91

regex queries, using 91
shadow fields, maintaining 92, 93
shadow fields, storing as objects 93, 94
quotes 96, 97

R

React App 325
coding 327
creating 325, 326
dependencies 326
Realm Web SDK, using 326
React App components
Context Hooks 334, 335
core components 327-332
question component 346-349
Question Context 342-345
question navigation component 356-358
questions, displaying 345
question tabs component 349-355
Realm context 332-334
useAggregate Hook 340-342
User Context 335-337
useUser Hook 337-340
recipes, as data 2, 3
query, constructing 4
relational databases 7
versus, document databases 7-9
replica set 199
automatic failover 199-201
configuration 201-203
connecting to 204-207
elections 199
initiation 201, 204
local database 207
oplog collection 208, 209
primaries 199
primary, changing to secondary 209, 210
secondaries 199
setting up 199
replication
risk reducing with 198
Rust 268

S

Salted Challenge Response Authentication Mechanism (SCRAM) 227
Scala 272
scaling, with sharding 216
config servers 217, 218
data, sharding 217
mongos process 217, 218
replica set considerations 218, 219
sharding configuration 219
shard keys 217
Secure Sockets Layer (SSL) 237
serverStatus command 189-191
SQL to MongoDB examples 389, 390
Stack overflow 2022 developer survey graph 20
strings 50
Structured Query Language (SQL) 7
Swift 270

T

Team Document 15
breakdown 13, 14
technical consultant 24
technical writer 24

time-series collection 154, 155
Time To Live (TTL) index
  creating 149
transactions 383
  in code 384-386
  in mongosh 383, 384
Transport Layer Security (TLS) 237

U

unique indexes 143, 144
Universal Unique Identifier (UUID) 49
useAggregate Hook 340-342
useUser Hook 337

V

Visual Studio Code 283
  MongoDB extension 283
  MongoDB playgrounds 284

W

wildcard index
  creating 148, 149

MongoDB for Jobseekers

DESCRIPTION

MongoDB for Jobseekers serves as the ultimate companion, providing assistance and support throughout your entire MongoDB learning journey. Whether you are an experienced professional exploring new career paths or an aspiring jobseeker looking to enhance your opportunities, this comprehensive guide is specifically designed to cater to your needs.

From the basics to advanced concepts, MongoDB for Jobseekers offers a well-structured approach to understanding the intricacies of this powerful NoSQL database. The book then delves into subjects like schema modeling, querying, indexing, and scalability, and discovers the reasons behind MongoDB's widespread popularity. Through clear and practical examples, the book will swiftly help you grasp the fundamental concepts and techniques required to work with MongoDB in real-life scenarios. This extensive guide will not only help establish a strong foundation in MongoDB but also unlock numerous job opportunities.

Upon completing this book, you will acquire the necessary confidence and expertise to excel in your job search and embark on a rewarding career path.

KEY FEATURES

- Master the fundamental principles of Schema Design, Querying, and Database Administration.
- Explore advanced topics, including Aggregation, Replication, and Sharding.
- Develop a fully functional application utilizing MongoDB Cloud Services.

WHAT YOU WILL LEARN

- Gain a comprehensive understanding of MongoDB's architecture and data model.
- Learn to perform CRUD operations (Create, Read, Update, Delete) in MongoDB.
- Understand indexing strategies for optimizing query performance.
- Discover MongoDB's aggregation framework for complex data analysis.
- Learn about MongoDB's high availability and scalability features.
- Explore integration with programming languages and frameworks.

WHO THIS BOOK IS FOR

Whether you are a novice starting from scratch or a seasoned professional aiming to enhance your database skills, this book is for individuals who aspire to learn about MongoDB, the contemporary "NoSQL" database.

BPB PUBLICATIONS
www.bpbonline.com

ISBN 978-93-5551-825-5