PRO ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

bhv®

Spring и Spring Boot

Разработка облачных приложений на Java

- Аспектно-ориентированное программирование
- Хранение и обновление настроек приложения
- Модуль Spring Core
- Фреймворк Spring Security
- Внедрение зависимостей
- Упаковка приложений в Java
- Создание приложений в стиле REST
- Тестирование Spring-приложений

Федор Урванов

Федор Урванов

Spring и Spring Boot

Разработка облачных приложений на Java

В книге рассмотрено актуальное состояние технологий Spring и Spring Boot, помогающих шире раскрыть возможности языка Java и аспектно-ориентированного программирования. Пошагово объяснено, как самостоятельно написать и развернуть облачный проект под управлением Spring. Рассказано о координации микросервисов на Java с применением Spring и модуля Spring Core, способах внедрения зависимостей, аннотировании кода. На примере сквозного веб-приложения описаны важнейшие приемы работы с данными в стиле REST, тестирование данных, обеспечение согласованности, безопасности и долговременного хранения данных в приложении.

Для Java-программистов

Оглавление

Введение

В современном мире практически все корпоративные приложения создаются с использованием каких-либо фреймворков. Существует множество фреймворков разной степени популярности, предназначенных для работы с различными языками программирования. Одни фреймворки имеют версии для нескольких языков с тем или иным уровнем поддержки, а другие — созданы специально для конкретного языка программирования. Обычно в компании работают с двумя-тремя наиболее популярными фреймворками, среди которых и производят выбор какого-либо одного, руководствуясь наличием у нее разработчиков, знакомых с ним, его особенностями и общим подходом, принятым в компании.

Применительно к языку программирования Java практически все новые системы пишутся с использованием Spring Framework, а в последнее время — с активным применением его компонента Spring Boot, значительно упрощающего инициализацию и конфигурацию Spring Framework.

Зачем нам нужны все эти фреймворки? Почему нельзя вести разработку на чистом языке программирования без них? На самом деле можно, но это будет гораздо сложнее просто потому, что фреймворки берут на себя большую часть заботы об основных стандартных функциях, которые есть практически в каждой корпоративной системе: работа с различными протоколами передачи данных, работа со слоем хранения данных, управление транзакциями, аутентификация, авторизация, интеграция с различными системами, связь разрозненных сторонних библиотек между собой, хоть какая-то стандартизация общей архитектуры и многое другое. Наличие фреймворка существенно облегчает введение нового сотрудника в команду разработки, так как он уже примерно понимает, что где искать и чего можно ожидать от проекта.

До Spring Framework (или просто Spring) разработка на Java чаще всего велась с использованием спецификации Enterprise JavaBeans, являющейся частью платформы Java EE (в 2018 году переименована в Jakarta EE). При желании можно и сейчас разрабатывать системы на последних версиях спецификаций Jakarta EE, задействовав сервер приложений GlassFish и совершенно не используя Spring. И такой подход даже вполне будет работать.

Тем не менее практически во всех случаях новые проекты на Java сейчас ведутся с использованием Spring Framework. К Jakarta EE, скорее всего, прибегают для поддержки старых проектов, которые по какой-либо причине переводить на Spring нет смысла, либо слишком сложно и ресурсоемко. Начинать новый проект на Jakarta EE вряд ли целесообразно, поскольку это приведет к сложностям в поиске разработчиков, да и в целом добавит проблем (но работать такой проект будет).

На самом деле, как бы там ни было, но нам в любом случае необходимо знать Jakarta EE хотя бы в общих чертах, так как Spring Framework работает далеко не сам по себе и активно использует эту спецификацию, да и запускается он все равно на одном из серверов приложений.

ГЛАВА 1

Теория

1.1. Краткая история Spring

Термин «Spring» может иметь разный смысл в разных контекстах. Он может означать:

- проект Spring Framework;
- всю экосистему проектов, построенных поверх Spring Framework.

В этой книге термин «Spring» используется для обозначения не только самого проекта Spring Framework, но и всей связанной с ним экосистемы проектов.

Spring появился в 2002 году. На тот момент основным способом разработки веб-приложений на Java было использование спецификации Java EE, которая в тот момент расшифровывалась как Java 2 Enterprise Edition (или сокращенно JEE). Как уже было отмечено ранее, спецификация эта была переименована в 2018 году в Jakarta EE.

Основной идеей Spring было упрощение чрезмерной сложности ранних спецификаций JEE.

> ***JEE называют по-разному***
>
> Java EE; JEE; J2EE; Java 2 Enterprise Edition; Java Platform, Enterprise Edition — это всё одна и та же спецификация, которая сейчас называется Jakarta EE.

JEE позволяла и до сих пор позволяет создавать высоконагруженные приложения, в которых необходима масштабируемость, надежность и гибкость.

Некоторые рассматривают Spring как конкурента Jakarta EE и современного ее преемника. В реальности это не совсем так — Spring не конкурирует с Jakarta EE, а, скорее, расширяет ее и активно использует часть ее спецификаций, таких как:

- внедрение зависимостей (JSR 330);
- основные аннотации (JSR 250);
- Servlet API (JSR 340);
- WebSocket API (JSR 356);

- Concurrency Utilities (JSR 236);
- JSON Binding API (JSR 367);
- Bean Validation (JSR 303);
- JPA (JSR 338);
- JMS (JSR 914);
- JTA/JCA, если это необходимо.

В первых версиях приложения на Spring разворачивались на сервере приложений. Сейчас благодаря Spring Boot приложения могут содержать встроенный в них сервер приложений и запускаться как обычные приложения, поддерживая все возможности, которые необходимы для запуска в облачной среде и для поддержки DevOps. При этом встроенный сервер приложений легко меняется на другой.

Начиная со Spring Framework 5, приложения WebFlux не используют Servlet API на прямую и могут разворачиваться на сервере приложений, не поддерживающем контейнер сервлетов, — например, на Netty.

1.2. Альтернативные фреймворки

Spring Framework не является единственным в мире Java — проектов для построения приложений на основе внедрения зависимостей и со сквозной поддержкой всех частей стандартного приложения учета достаточно много.

- Jakarta EE вполне можно использовать вместе с одной из реализаций этой спецификации: WebSphere, WildFly, GlassFish, Apache Tomcat, Jetty и других.
- Google Guice[1] — легковесный фреймворк, разработанный компанией Google. Построен на принципах внедрения зависимостей и использует аннотации из Jakarta EE.

В свое время развивались и другие фреймворки, основанные на внедрении зависимостей, — наподобие JBoss Seam Framework и PicoContainer, но для большинства из них жизненный цикл закончен, и практически все проекты на Java используют именно Spring Framework.

1.3. Внедрение зависимостей

Современные приложения, написанные с использованием объектно-ориентированных языков программирования, состоят из большого количества классов, реализующих бизнес-логику. Все они взаимодействуют друг с другом, вызывают методы, считывают и устанавливают свойства, создают новые экземпляры и т. д. Теоретический пример того, как может выглядеть часть подобных связей, показан на рис. 1.1.

---

[1] См. **https://github.com/google/guice**.

Рис. 1.1. Абстрактная схема связей классов бизнес-слоя приложения

Экземпляры классов должны каким-либо образом получать ссылки друг на друга, взаимодействовать между собой, передавая результаты своей работы и инициируя дальнейшую их обработку. Схема связей классов, приведенная на рис. 1.1, имеет весьма абстрактный характер, но достаточно хорошо показывает, что различных классов может быть довольно много.

Каким образом экземпляр **Класс1** может узнать о существовании экземпляра **Класс2**? Каким образом **Класс4** должен узнать о существовании **Класс7**? Кто обязан создавать экземпляры всех этих классов?

Для облегчения понимания сути сведем проблему к более конкретным классам и более понятным их функциям. Предположим, что у нас есть два класса:

- `CollectionsShelf` — обработка сбора игроком коллекционных предметов: статуэток, наклеек, наборов брони и т. д.;
- `Journal` — работа с журналом игрока: статусы заданий, записи о том, что в них нужно сделать, информация о мире, услышанные истории.

Пока эти два класса никак не связаны между собой (рис. 1.2), каждый занимается своей работой, и никаких проблем нет.

Введем дополнительный класс `QuestEngine`, который по замыслу должен работать как движок заданий, выдаваемых игроку, — по мере выполнения заданий движок заданий добавляет записи в журнал игрока. За выполнение заданий выдаются различные предметы из собираемых коллекций. В конечном итоге `QuestEngine` для выполнения указанной функциональности должен быть связан и с `Journal`, и с `CollectionsShelf` (рис. 1.3).

Для того чтобы `QuestEngine` мог добавлять элементы в коллекции `CollectionsShelf` и записи в журнал `Journal`, он должен каким-то образом узнать об их существовании.

Рис. 1.2. Классы `CollectionsShelf` **и** `Journal` **никак не связаны**

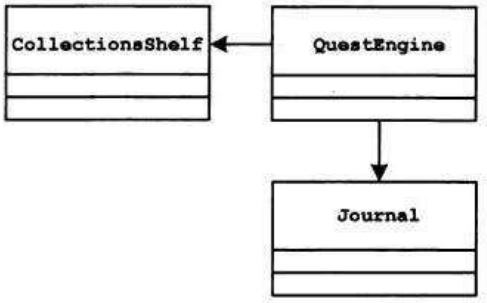


Рис. 1.3. Класс `QuestEngine` **и его связи**

В самом простом варианте `QuestEngine` может создавать необходимые экземпляры `CollectionsShelf` и `Journal` самостоятельно, но в этом случае он жестко завязывается на конкретную реализацию этих классов. Однако классы `CollectionsShelf` и `Journal`, их конкретная реализация и управление их жизненным циклом — это уже не ответственность движка квестов. Согласно *принципу единственной ответственности* класс `QuestEngine` должен заниматься только деятельностью, связанной с обработкой квестов.

> ***Принцип единственной ответственности***
>
> **Каждый объект должен иметь только одну ответственность, и эта ответственность должна быть полностью инкапсулирована в класс.**

Поскольку сам `QuestEngine` не может заниматься созданием экземпляров `CollectionsShelf` и `Journal`, значит, это должен делать какой-то другой класс, или группа классов, или еще какой-нибудь другой механизм, который, в свою очередь, будет предоставлять классу `QuestEngine` готовые ссылки на уже созданные и инициализированные объекты. Подобный процесс называется *внедрением зависимости* (Dependency Injection, DI).

> ***Внедрение зависимости***
>
> **Внедрение зависимости, инъекция зависимости — процесс предоставления внешней зависимости программному компоненту. Сам объект при этом пассивен и не принимает действий по вычислению и отбору своих зависимостей, а только предоставляет для этого методы установки значений, конструкторы или какой-либо другой способ, как показано на рис. 1.4.**

До текущего момента всё объяснялось с помощью графиков и схем. Далее показано, как внедрение зависимостей может выглядеть в программном коде Java.

Листинг 1.1. Achievements.java

```
/**
 * Обработка достижений игрока и получение за них наград.
 */
class Achievements {
    // ...
}
```

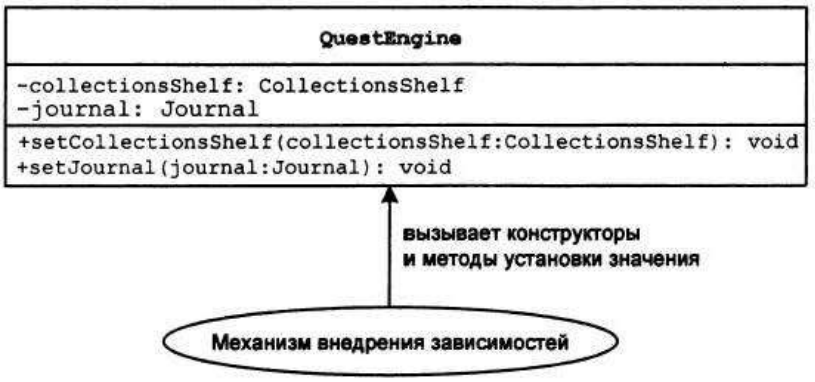


Рис. 1.4. Внедрение зависимостей

Листинг 1.2. CollectionsShelf.java

```
/**
 * Собираемые коллекции предметов.
 */
class CollectionsShelf {
    // ...
}
```

Листинг 1.3. Journal.java

```
/**
 * Дневник с описанием заданий, текущим прогрессом выполнения и т. д.
 */
class Journal {
    // ...
}
```

Внедрение зависимостей может осуществляться:

- через конструктор;
- через методы установки значений;
- через прямой доступ к полям класса;
- через интерфейс;
- другими способами.

При внедрении зависимостей через конструктор очевидно, что этот конструктор должен принимать их в качестве параметров (листинг 1.4).

Листинг 1.4. QuestEngine.java

```
class QuestEngine {
    private CollectionsShelf collectionsShelf;
    private Journal journal;
    private Achievements achievements;
```

```
    public QuestEngine(
            CollectionsShelf collectionsShelf,
            Journal journal,
            Achievements achievements) {
        super();
        this.collectionsShelf = collectionsShelf;
        this.journal = journal;
        this.achievements = achievements;
    }
```

При внедрении зависимостей через методы установки значений (сеттеры) объект предоставляет соответствующие методы, но при этом так же, как и в случае внедрения зависимостей через конструктор, не заботится об их создании и получении. Этим обычно занимается фреймворк. Например, наш класс работы с заданиями игрока `QuestEngine`, готовый для внедрения зависимостей через методы установки значений, мог бы выглядеть так (листинг 1.5).

Листинг 1.5. QuestEngine.java

```
class QuestEngine {
    private CollectionsShelf collectionsShelf;
    private Journal journal;
    private Achievements achievements;

    public void setCollectionsShelf(CollectionsShelf collectionsShelf) {
        this.collectionsShelf = collectionsShelf;
    }

    public void setJournal(Journal journal) {
        this.journal = journal;
    }

    public void setAchievements(Achievements achievements) {
        this.achievements = achievements;
    }
```

При внедрении зависимостей через интерфейс класс предоставляет специальный интерфейс, с помощью которого внедряет зависимость от себя в любой другой класс. Например, для `QuestEngine` и `CollectionsShelf` это может выглядеть примерно так (листинг 1.6).

Листинг 1.6. Пример внедрения зависимостей через интерфейс

```
class CollectionsShelf {
    // ...
}

interface CollectionsShelfSetter {
    void setCollectionsShelf(CollectionsShelf collectionsShelf);
}
```

```
class CollectionsShelfInjector {
    private CollectionsShelf collectionsShelf = new CollectionsShelf();

    public void inject(CollectionsShelfSetter collectionsShelfSetter) {
        collectionsShelfSetter.setCollectionsShelf(collectionsShelf);
    }
}

class QuestEngine implements CollectionsShelfSetter{
    private CollectionsShelf collectionsShelf;
    // ... аналогично с Journal и Achievements...

    public void setCollectionsShelf(CollectionsShelf collectionsShelf) {
        this.collectionsShelf = collectionsShelf;
    }
}
```

1.4. Инверсия управления

Внедрение зависимостей, описанное в *разд. 1.3*, — это лишь один из возможных подходов к реализации *инверсии управления*, который рассматривается в этом разделе.

Обычно ход выполнения программы полностью контролируется программистом, как показано на рис. 1.5.

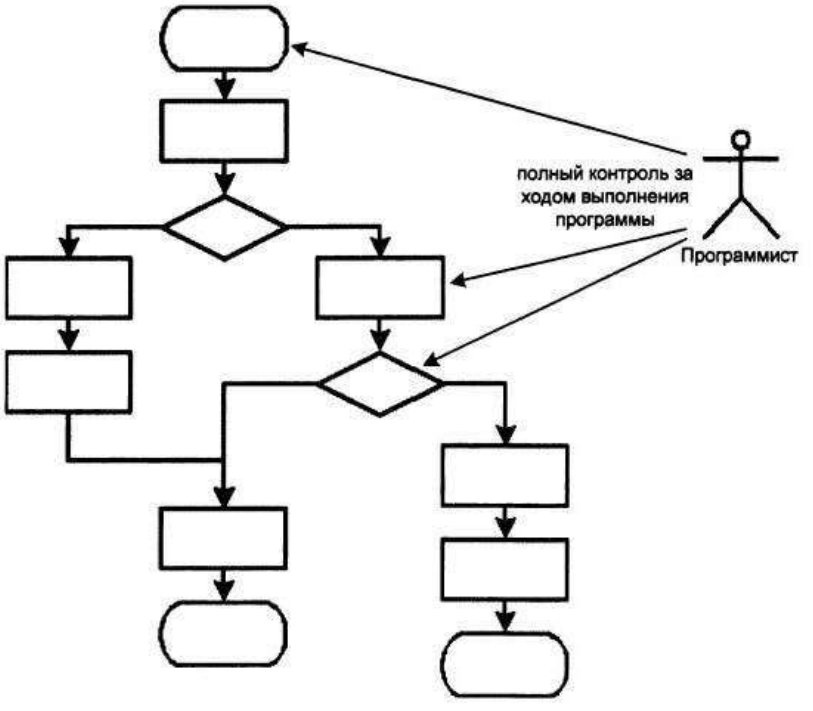


Рис. 1.5. Обычная программа

При инверсии управления программист не управляет напрямую ходом исполнения программы — он предоставляет обработчики событий, которые позволяют выполнить его код в определенных ее местах (рис. 1.6).

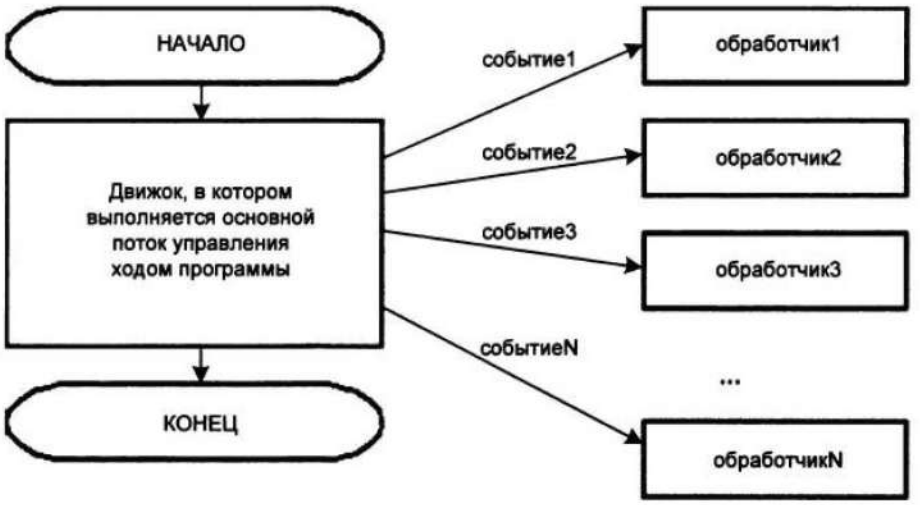


Рис. 1.6. Инверсия управления

Один из хороших примеров инверсии управления — разработка экранных форм. Практически во всех современных операционных системах, библиотеках и средах программистом описываются обработчики событий нажатия на кнопки, изменения размеров форм, состояния, изменения значений в полях ввода и т. п., но он не контролирует напрямую саму логику экранной формы, — этим обычно занимается операционная система.

Вот отличный пример — экранная форма входа в игру виртуальных питомцев, по которой написана эта книга (рис. 1.7). Сам код клиентской части не входит в рассматриваемую тему, т. к. не использует Spring, но в экранной форме присутствуют:

- обработчик кнопки **Зарегистрироваться**;
- обработчик кнопки **Войти**;
- обработчик кнопки **Я забыл(а) пароль**.

Однако в коде экранной формы входа нет последовательной логики обработки формы — этим занимается движок браузера совместно с операционной системой.

К этому моменту мы познакомились с двумя главными принципами, на которых построено ядро Spring Framework:

- инверсия управления;
- внедрение зависимостей (описывается в *разд. 1.3*).

Инверсия управления

Инверсия управления (Inversion of Control, IoC) — принцип объектно-ориентированного программирования, используемый для уменьшения связанности. Он заключается

в том, что код программиста вызывает фреймворк или библиотека — например, с помощью интерфейсов и / или callback'ов, — здесь и происходит инверсия управления. Такой подход отличается от стандартного, при котором контроль над потоком выполнения программы полностью лежит на программисте.

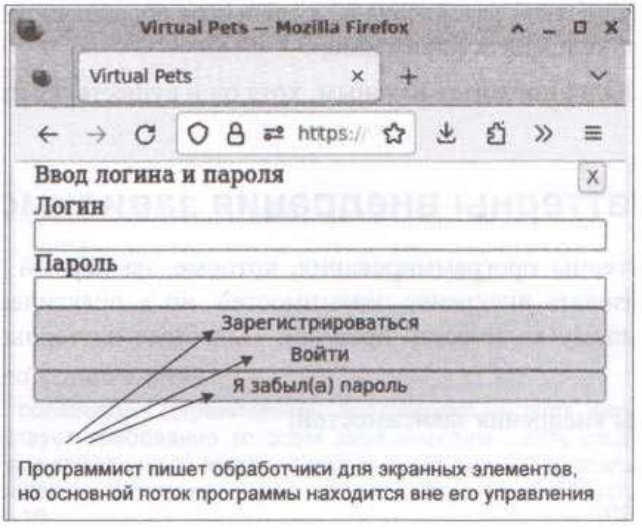


Рис. 1.7. Экранная форма и обработчики

В соответствии с этими двумя принципами наш класс `QuestEngine` из *разд. 1.3* мог бы выглядеть примерно так (листинг 1.7).

Листинг 1.7. QuestEngine

```
class QuestEngine {
    private CollectionsShelf collectionsShelf;
    private Journal journal;
    private Achievements achievements;

    public void setCollectionsShelf(CollectionsShelf collectionsShelf) {
        this.collectionsShelf = collectionsShelf;
    }

    public void setJournal(Journal journal) {
        this.journal = journal;
    }

    public void setAchievements(Achievements achievements) {
        this.achievements = achievements;
    }

    // ... some methods with logic
}
```

Создавать экземпляр класса `QuestEngine` и устанавливать в него зависимости с помощью методов `setCollectionsShelf`, `setJournal` и `setAchievements` будет сам фреймворк — наш класс не должен задумываться о том, откуда они берутся и как создаются.

Инверсия управления имеет свои определенные недостатки:

- логика взаимодействия разбросана по отдельным обработчикам событий и классам, что может усложнить понимание;
- поток управления становится неявным, хотя он и существует на самом деле.

1.5. Антипаттерны внедрения зависимостей

Существуют паттерны программирования, которые, на первый взгляд, будто бы позволяют реализовать внедрение зависимостей, но в практическом применении приводят к большому количеству проблем. Подобные паттерны называют *антипаттернами*.

Вот антипаттерны внедрения зависимостей:

- Control freak;
- Bastard injection;
- Constrained construction;
- Service locator;
- нарушение принципа единственной ответственности.

Control freak

Control freak (руководитель-наркоман) — все зависимости контролируются напрямую. Возникает тогда, когда мы создаем внутри класса изменяемую зависимость. Пример этого антипаттерна приведен в листинге 1.8.

Листинг 1.8. Control freak

```
class QuestEngine {
    private CollectionsShelf collectionsShelf;
    private Journal journal;

    QuestEngine() {
        collectionsShelf = new CollectionsShelf();
        journal = new Journal();
    }
}
```

Bastard injection

Bastard injection (внебрачная зависимость) — наличие конструктора по умолчанию, который создает необходимые объекты и передает их в свой параметризованный вариант. Пример этого антипаттерна приведен в листинге 1.9.

Листинг 1.9. Bastard injection

```
public class QuestEngine {
    private CollectionsShelf collectionsShelf;
    private Journal journal;

    public QuestEngine() {
        this(
                new CollectionsShelf(),
                new Journal());
    }

    public QuestEngine(
            CollectionsShelf collectionsShelf,
            Journal journal) {
    }
}
```

Constrained construction

Constrained construction (ограниченное построение) — этот антипаттерн возникает, когда существует требование ко всем зависимостям иметь особенный конструктор. Оно может возникнуть из-за желания получить возможность создания объектов однотипным образом — например, через reflection. Пример этого антипаттерна приведен в листинге 1.10.

Листинг 1.10. Constrained construction

```
// Из каких-либо файлов настроек считываются
// typeFromSettings, nameFromSettings, priceFromSettings.
Class<?> clazz = Class.forName(typeFromSettings);
Constructor<?> constructor = clazz.getConstructor(
        String.class, int.class);
Object collectionShelfItem = constructor.newInstance(
        nameFromSettings, priceFromSettings);
```

Service locator

Service locator (локатор служб) — получение зависимостей через единый локатор служб, который создает зависимости либо возвращает уже созданные ранее. Пример кода локатора служб может выглядеть так, как показано в листинге 1.11.

Листинг 1.11. Класс ServiceLocator

```
class ServiceLocator {
    private static Set<Object> services = new HashSet<>();

    @SuppressWarnings("unchecked")
    public static <T> T getService(Class<T> clazz) {
        return (T) services.stream()
            .filter(v -> v.getClass().isAssignableFrom(clazz))
            .findFirst()
            .orElseThrow(() -> new IllegalArgumentException(""));
    }
```

```
    public static void registerService(Object service) {
        services.add(service);
    }
}

class CollectionsShelf {
    // ...
}

class Journal {
    // ...
}

class Achievements {
    // ...
}

class QuestEngine {

    public void completeQuest() {
        Journal journal = ServiceLocator.getService(Journal.class);
        // Использование journal.

        System.out.println("Quest completed. ");
    }
}

public class ServiceLocatorExample {
    public static void main(String[] args) {
        // Добавляем все наши зависимости
        ServiceLocator.registerService(new CollectionsShelf());
        ServiceLocator.registerService(new Journal());
        ServiceLocator.registerService(new Achievements());
        ServiceLocator.registerService(new QuestEngine());

        QuestEngine questEngine = ServiceLocator.getService(QuestEngine.class);
        questEngine.completeQuest();
    }
}
```

Подобный подход уже выглядит гораздо лучше подходов, описанных ранее, но при этом все классы будут дополнительно зависеть от класса `ServiceLocator`, что само по себе увеличит связанность, затруднит тестирование, да и логически создает дополнительную зависимость для каждого класса, которая им не особо нужна. Многие ошибки при таком подходе будут появляться уже на этапе выполнения, хотя вполне могли бы быть отловлены еще на этапе компиляции.

Метод `getService` не обязательно должен принимать тип класса в качестве параметра — локатор служб может быть создан на основе текстовых идентификаторов сервисов, а не типов.

В целом подход с локатором служб на текущий момент все же считается антипаттерном, поскольку приводит к сокрытию зависимостей класса. При чтении сигнатуры конструкторов и методов невозможно определить, использует ли класс какие-либо зависимости, а если использует, то какие именно.

1.6. Многоуровневая архитектура

Многоуровневая, или многослойная, архитектура — это шаблон проектирования, при котором приложение разделяется на несколько слоев.

Существуют различные способы разделения приложение на слои:

- например, Model-View-Controller (Модель-Представление-Контроллер) — сокращенно MVC:
  - Model (модель) — данные и методы работы с ними. Модель не знает ничего о других слоях, она только изменяет данные по запросам из слоя контроллеров и сохраняет их в базу данных. Зачастую именно здесь и содержится бизнес-логика;
  - View (представление) — отображение данных пользователю и обработка событий пользовательского интерфейса. Представление не обрабатывает данные пользователя — этим занимаются другие слои;
  - Controller — связывает два слоя воедино. Преобразует ввод пользователя в методы модели или в события представления.

  MVC часто используется в веб-программировании, где представление — это код HTML, CSS и JavaScript, который загружается и используется браузером для формирования интерфейса пользователя, а контроллер и модель — это код на сервере;
- Model-View-Presenter (MVP) — это другой популярный вариант трехуровневой архитектуры, производный от Model-View-Controller. Обычно он используется при проектировании интерфейсов пользователя. Presenter в нем содержит обработчики событий пользовательского интерфейса.

MVC и MVP — лишь частные случаи многоуровневой архитектуры. Существуют и другие способы разделения на слои, и ни один из них не идеален, поэтому в разных приложениях могут лучше подойти и другие способы.

Количество слоев в приложении может быть любым — в зависимости от целей, которых требуется достигнуть. Часто выделяются следующие слои:

- слой постоянства;
- слой бизнес-логики;
- слой контроллеров;
- слой презентации;
- слой безопасности;
- слой пакетных заданий.

Слой постоянства отвечает за сохранение данных приложения. Обычно данные сохраняются в какую-нибудь базу данных — чаще всего в реляционную. В современном мире очень часто используется технология ORM (Object-Relational-Mapping), которая облегчает отображение содержимого баз данных на классы Java. В большинстве случаев в случае с Java используется Hibernate в качестве реализации этого подхода, но сам стандарт Jakarta EE уже содержит в себе интерфейсы и аннотации для работы с ORM.

Слой бизнес-логики отвечает за все расчеты, преобразования данных и т. п. Он каким-либо образом взаимодействует со слоем постоянства, вызывая методы сохранения данных и загрузки данных (рис. 1.8).

Рис. 1.8. Слой бизнес-логики взаимодействует со слоем постоянства, вызывая его методы

При этом сам слой постоянства состоит из большого числа компонентов, различных для каждого способа хранения данных (рис. 1.9). Это могут быть, например:

1. Драйвер JDBC.
2. Классы приложения, работающие с JDBC, вызывающие методы `Statement` или `PreparedStatement`, либо методы ORM, либо другие методы используемого хранилища. Подобные классы зачастую создаются максимально независимыми от остальных частей слоя постоянства и называются классами Data Access Object, DAO. Бизнес-слой при этом взаимодействует только с классами из Data Access Object, не взаимодействуя напрямую с остальными частями слоя постоянства, так что остальные части можно менять, не заботясь об их влиянии на бизнес-логику.
3. Используемая в приложении ORM — например, Hibernate.
4. Схема базы данных с таблицами, представлениями, последовательностями, триггерами, хранимыми процедурами и функциями.
5. Механизм проверки версии схемы базы данных, обновления этой схемы и приведения ее в состояние, необходимое новой версии приложения.
6. База данных, развернутая на каком-либо сервере и доступная к использованию по определенному адресу и порту в сети.

Слой презентации отвечает за взаимодействие с пользователем — здесь могут быть различные экранные формы, десктопное приложение, веб-приложение, мобильное приложение, RSS-ленты и т. д. В нашем примере игры с виртуальными питомцами слой презентации для страниц сайта представлен в виде JSPX-страниц — для варианта на Spring Framework и шаблонов страниц Thymeleaf — для варианта на Spring

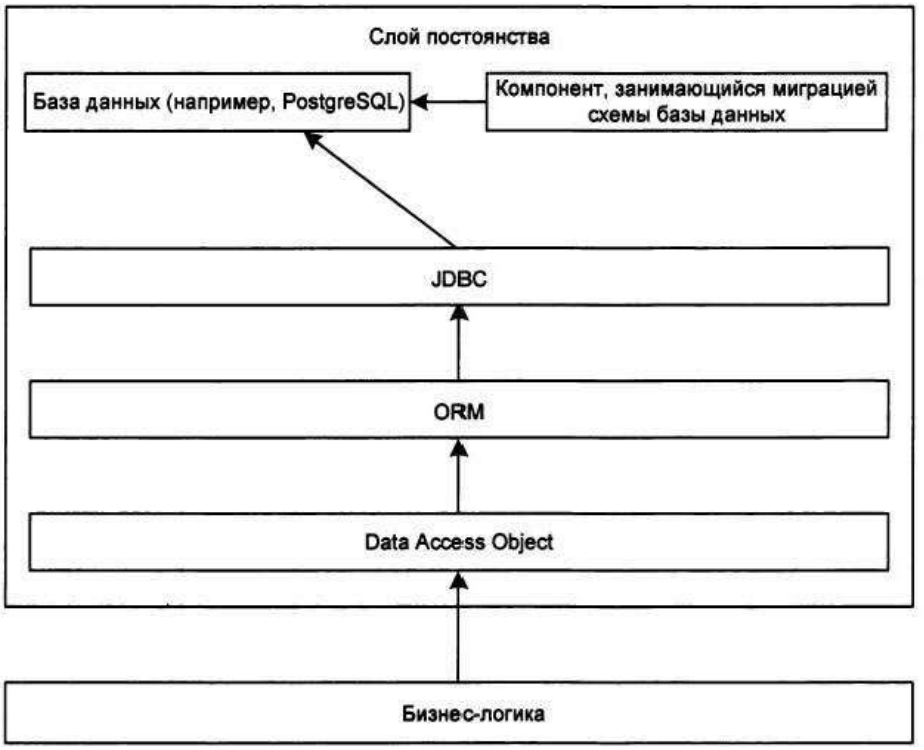


Рис. 1.9. Слой постоянства и бизнес-логика

Boot. Сама клиентская часть игры со всей логикой представляет собой Progressive Web Application на JavaScript.

Между слоем презентации и слоем бизнес-логики часто находится *слой контроллеров*, которые преобразуют пользовательский ввод в методы бизнес-слоя (рис. 1.10).

Слой безопасности ограничивает доступ к защищенным ресурсам — например, по ролям пользователей или по каким-нибудь другим настройкам. В нашем приложении игры виртуальных питомцев этот слой будет построен с помощью Spring Security. Слой безопасности — это не просто отдельный самостоятельный слой. Он проходит через все остальные слои, обеспечивая безопасность всего приложения (рис. 1.11). Нельзя просто в слое презентации скрыть кнопки и экранные формы и считать, что с безопасностью закончено. Проверки допустимости выполнения тех или иных действий должны осуществляться зачастую и в бизнес-слое.

Слой пакетных заданий отвечает за обработку длительных задач, которые запускаются по расписанию: различные выгрузки, синхронизации, отправки отчетов и прочее (рис. 1.12). В Spring для этого обычно используется Spring Batch.

В приложениях также могут быть другие слои в зависимости от требуемой функциональности приложений.

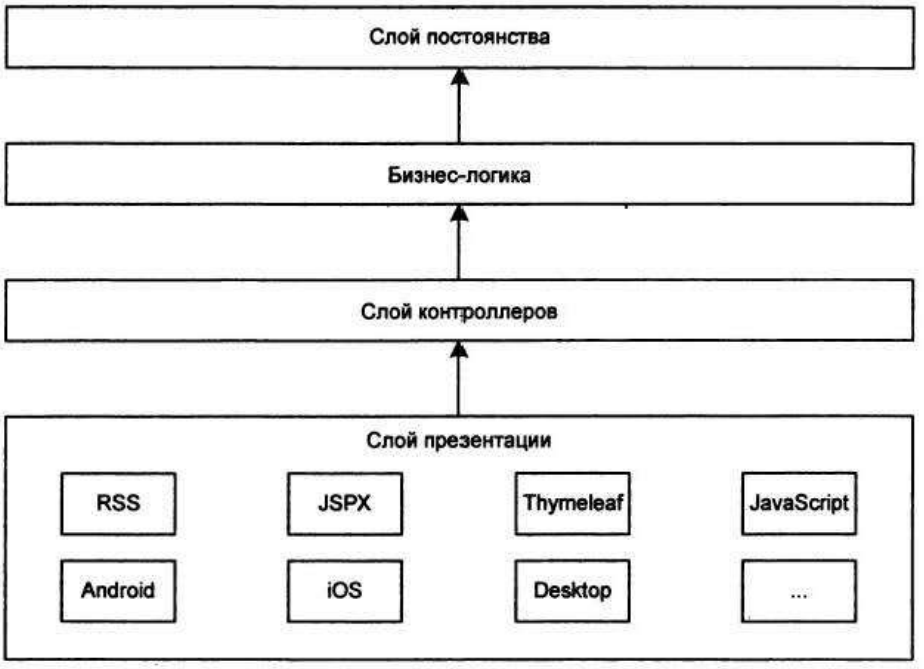


Рис. 1.10. Слой контроллеров преобразует пользовательский ввод со слоя презентации в методы слоя бизнес-логики

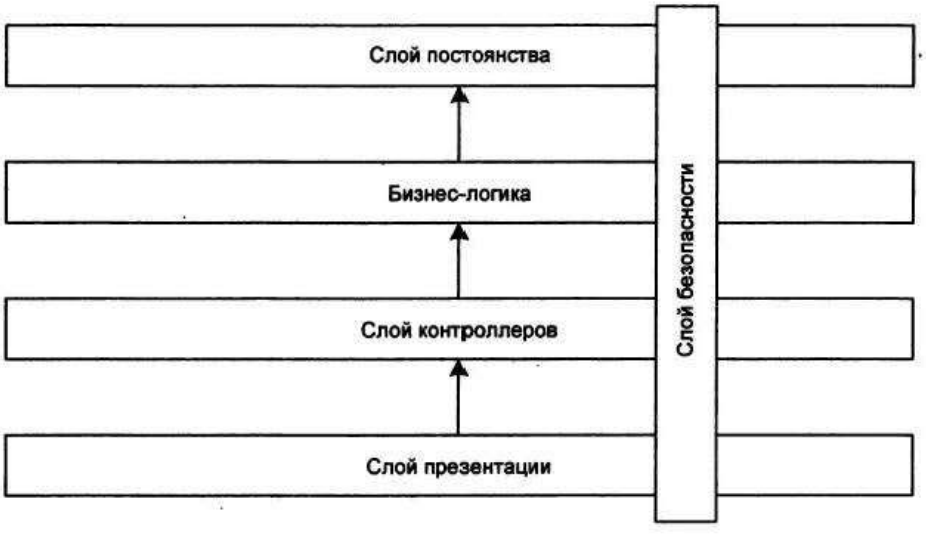


Рис. 1.11. Слой безопасности проходит через все другие слои

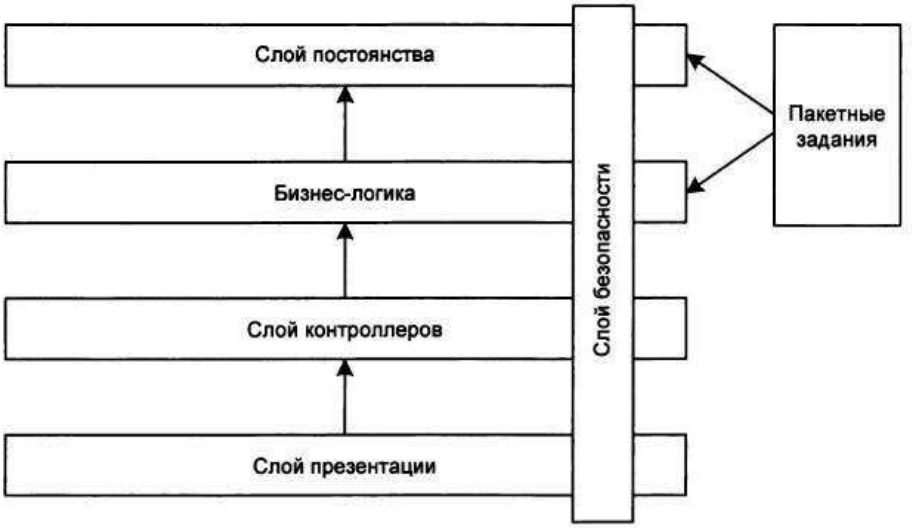


Рис. 1.12. Слой пакетных заданий

1.7. Аспектно-ориентированное программирование

В предыдущих разделах описывалось внедрение зависимостей и инверсия управления — как основных принципов, на которых построена платформа Spring Framework. Все эти технологии задействуются для уменьшения связности. В дополнение к ним Spring Framework предлагает аспектно-ориентированное программирование.

Аспектно-ориентированное программирование (Aspect-Oriented Programming, AOP, АОП) — технология для обеспечения сквозной функциональности, повседневно используемая программистами, работающими с Spring Framework. Хорошими примерами АОП могут, например, служить запись в журнал лога событий обращения к какому-либо ресурсу, управление транзакциями и какие-либо другие функции, которые напрямую не связаны с тем или иным конкретным объектом или бизнес-процессом, но должны осуществляться по всему приложению. Суть аспектно-ориентированного программирования заключается в том, что разработчик не добавляет логику, связанную со сквозным процессом, в методы бизнес-процессов, а разрабатывает ее отдельно вместе с описанием точек, в которых она должна выполняться.

Рассмотрим вызов метода `addEntry` класса `Journal` из класса `QuestEngine` (рис. 1.13).

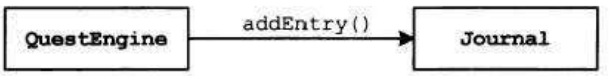


Рис. 1.13. Обычный вызов метода `addEntry`

Предположим, что нам при добавлении записей в журнал необходимо выполнить дополнительное действие — например, подсчитать добавленные в журнал записи, чтобы по достижении определенного их количества отображать получение достижения игроком. Всё это можно сделать в самом методе `addEntry`, но, допустим, что мы хотим вынести из класса `Journal` всю логику сборки подобной статистки и работы с системой достижений. В этом случае на помощь приходит аспектно-ориентированное программирование, которое позволяет вклиниться в цепочку и добавить дополнительную логику (совет) в точки (срез), где вызывается метод `addEntry`, но при этом не изменяя код самого метода `addEntry` (рис 1.14).

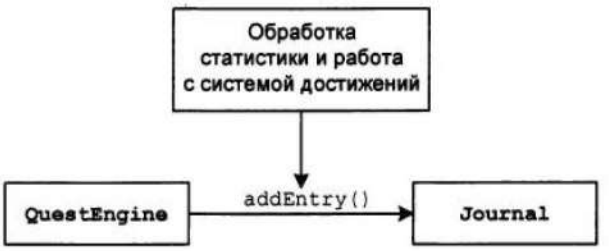


Рис. 1.14. Обработка статистики в точках вызова метода `addEntry`

В парадигме АОП используются следующие термины:

- *аспект* (aspect) — это комбинация советов и срезов. Аспект изменяет поведение остального кода, применяя совет в точках соединения, определенных некоторым срезом;
- *совет* (advice) — кусок кода, который должен выполниться в точке соединения, перед точкой соединения или после нее;
- *точка соединения* (join point) — четко определенная точка в выполняемой программе, где следует применить совет. Примеры: обращение к методу, обращение к полю, инициализация класса, создание экземпляра объекта, возникновение исключения;
- *срез* (pointcut) — набор точек соединения;
- *связывание* (weaving) — представляет собой процесс вставки аспектов в определенную точку кода;
- *цель* (target) — объект, поток выполнения которого изменяется через АОП.
- *введение* (introduction) — изменение структуры класса или иерархии. В процессе введения в класс могут, например, добавляться поля и методы, необходимые для реализации какого-либо интерфейса, но без изменения исходного класса.

АОП бывает двух видов:

- *статическое*.

  При статическом АОП связывание происходит во время компиляции приложения. Конечный байт-код будет просто обычным байт-кодом Java и не станет требовать каких-либо дополнительных действий. Пример реализации: AspectJ;

- *динамическое*.

  Связывание происходит во время выполнения. К недостаткам такого подхода можно отнести низкую производительность, но обычно накладные расходы на это не столь существенны, как, например, обращение к внешним ресурсам. К преимуществам подхода можно отнести то, что при изменении аспектов не требуется перекомпилировать код всего приложения. Пример реализации динамического АОП: Spring AOP, которое будет рассмотрено подробнее далее.

1.8. Резюме

Spring предоставляет современный способ создания приложений на Java. Существуют другие фреймворки, разработанные по сходным и отличающимся принципам, но в современном мире они не так популярны. Внедрение зависимостей и инверсия управления — это две основные концепции, на которых построена разработка приложений с помощью Spring.

Разрабатываемое на Spring приложение зачастую делится на слои, в которые помещаются классы в соответствии со своей функцией. Обычно выделяют:

- слой DAO;
- слой бизнес-логики;
- слой представления.

При разработке сквозной функциональности принято использовать аспектно-ориентированное программирование, но следует быть осторожным, иначе это может затруднить понимание основной логической цепочки обработки запросов.

ГЛАВА 2

Микросервисы

2.1. Для чего нужны микросервисы?

Микросервисы — это небольшие, слабо связанные, легко изменяемые сервисы, разрабатываемые в какой-нибудь предметной области и выпускаемые независимо. Обычно микросервисы общаются между собой с помощью обмена JSON-сообщениями в стиле REST, но, разумеется, это не единственно возможный вариант. Вполне могут использоваться обмены сообщениями с помощью брокера сообщений наподобие IBM MQ, Apache Kafka, RabbitMQ, либо взаимодействие может быть организовано через gRPC (система удаленного вызова процедур с HTTP/2 и Protocol Buffers).

Очень часто микросервисы и микросервисную архитектуру рассматривают как противоположность монолиту.

> ***Монолит***
>
> *Монолит* — один большой сервис, внутри которого зашита вся логика приложения. Его отдельные модули вызывают методы друг друга напрямую, т. к. находятся внутри одной кодовой базы и одного приложения.

Сразу стоит отметить, что не стоит рассматривать микросервисы как «серебряную пулю». Далеко не каждому проекту они нужны, а даже если нужны, то степень разбиения его на микросервисы может быть совершенно разной. Да и сам монолит далеко не всегда можно встретить в его чистом виде — зачастую какие-нибудь отдельные функции даже у него уже могут быть вынесены в отдельный сервис.

Плюсы микросервисной архитектуры:

- легкость масштабирования — любой микросервис можно запустить в таком количестве экземпляров, который необходим для текущей нагрузки;
- каждый микросервис можно переписать на другом стеке технологий, более подходящим под задачу конкретно этого микросервиса;
- отказоустойчивость — если произойдет сбой в каком-либо одном микросервисе, то это не приведет к падению всей системы. К тому же при падении микросервиса можно настроить его перезапуск;

- частые релизы — каждый микросервис можно выводить в продакшен отдельно от остальных сервисов.
- Минусы микросервисной архитектуры:
- дополнительные расходы на сетевое взаимодействие между сервисами;
- увеличение общей сложности системы — каждый микросервис сам по себе может быть довольно простым, но понимание работы всей системы целиком может оказаться затрудненным;
- отсутствие стандартизации в форматах сообщений микросервисов;
- усложнение интеграционного тестирования — микросервисная архитектура предполагает наличие большого количества маленьких сервисов. И для ее полноценного интеграционного тестирования на тестовом стенде нужно развернуть все необходимые микросервисы, предоставив им необходимые настройки, ресурсы процессора и т. д.

2.2. Основные компоненты микросервисной архитектуры

Микросервисам необходимо каким-либо образом отправлять сообщения в другие микросервисы. В предыдущем разделе уже упоминалось, что чаще всего в этих целях используется обмен JSON-сообщениями через REST-подобные API. Для того чтобы обратиться к другому микросервису, необходимо, кроме формата сообщения, знать адрес, на который отправлять запросы к нему. В соответствии с этим микросервисная архитектура обычно содержит:

- базу данных service registry;
- механизм service discovery;
- единую входную точку API gateway;
- интеграционный слой service mesh.

Service registry — это база данных, хранящая все адреса и порты сервисов. Каждый микросервис регистрируется в service registry, добавляя свой актуальный адрес и порт. При необходимости обращения к другому сервису исходный сервис обращается к service registry, получает из него актуальный порт и адрес необходимого сервиса либо их список и по нему делает запрос.

Service discovery — механизм обнаружения сервисов и выбора экземпляра из их множества для отправки запросов. При микросервисной архитектуре может быть запущено несколько экземпляров каждого сервиса. Периодически могут подниматься новые экземпляры и останавливаться уже запущенные. Соответственно, при взаимодействии сервисов между собой необходимо каким-то образом выбрать один экземпляр из списка запущенных и зарегистрированных в service registry.

Существуют два вида service discovery:

- server-side service discovery;
- client-side service discovery.

При *server-side service discovery* сервис, инициирующий запрос, отправляет его не напрямую конечному сервису, а на балансировщик нагрузки (load balancer), который уже выбирает по какому-либо алгоритму конечный запущенный экземпляр сервиса, которому предстоит обрабатывать запрос.

Преимущества server-side service discovery:

- проще код клиента сервиса, отправляющего запрос, т. к. он содержит только отправку запроса, без дополнительного кода по выбору экземпляра запущенного сервиса;
- многие облачные среды уже содержат свой компонент, реализующий балансировку и выбор экземпляра конечного сервиса.

При *client-side service discovery* логика выбора экземпляра сервиса из списка сервисов, полученных от service registry, находится внутри самого сервиса, отправляющего запрос.

Преимущества client-side service discovery:

- меньшее количество сетевых запросов, т. к. отсутствует отдельный сервис балансировщика нагрузки (load balancer), занимающийся балансировкой;
- при переносе между разными облачными средами логика балансировки остается прежней.

API Gateway (рис. 2.1) — единая входная точка для внешних сервисов (мобильных клиентов, веб-клиентов, десктопных приложений и т. п.), которая скрывает за собой различные внутренние сервисы, собирает один ответ клиентскому приложению из ответов нескольких сервисов, преобразует данные из внутреннего представления в формат, необходимый клиентскому, а также выполняет дополнительную логику по логированию, сбору статистики, авторизации и аутентификации.

Service mesh (рис. 2.2) — интеграционный слой, обеспечивающий маршрутизацию сообщений между микросервисами, обнаружение сервисов, балансировку нагрузки,

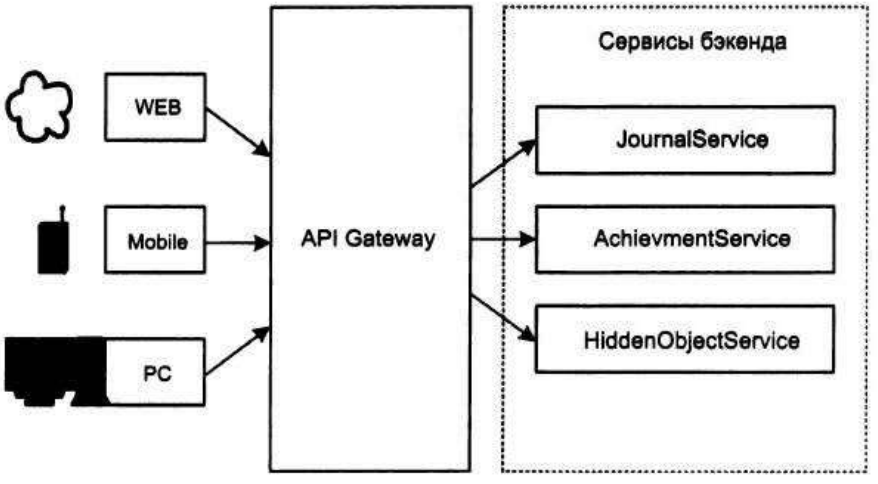


Рис. 2.1. API Gateway

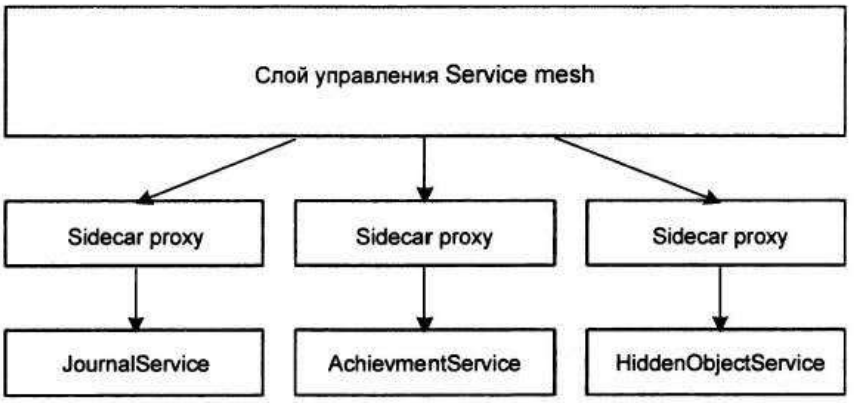


Рис. 2.2. Service mesh

логирование, авторизацию, аутентификацию и прочее. Обычно он реализуется с помощью добавления каждому сервису прокси-сервиса, называемого sidecar proxy.

2.3. Spring Cloud

Spring содержит около десятка проектов Spring Cloud. Они не обязательны для использования — можно разрабатывать микросервисы на Spring без Spring Cloud, но зачастую эти проекты могут облегчить разработку, т. к. возьмут на себя некоторую часть однотипной функциональности, необходимой для работы микросервисной архитектуры.

- Spring Cloud Commons — реализует индикацию состояния сервиса (health indiators), базовый механизм Service Discovery с поддержкой разных реализаций другими проектами Spring Cloud, базовый интерфейс Service Registry с поддержкой разных реализаций другими проектами Spring Cloud, RestTemplate с балансировщиком нагрузки на клиентской стороне;
- Spring Cloud Netflix — реализует поддержку Eureka Service Discovery на основе механизма из Spring Cloud Commons;
- Spring Cloud Zookeeper — позволяет интегрировать Spring Cloud с Apache Zookeeper, реализующим Service Discovery, Service Registry, распределенную конфигурацию и балансировщик нагрузки;
- Spring Cloud Consul — позволяет интегрировать Spring Cloud с Consul, реализующим Service Discovery, балансировщик нагрузки, API Gateway вместе с Spring Cloud Gateway, распределенную конфигурацию и обмен сообщениями через Consul Events;
- Spring Cloud Config — реализует поддержку внешней конфигурации для сервисов;
- Spring Cloud Bus — реализует поддержку обмена сообщениями через брокер сообщений AMQP или Kafka;

- Spring Cloud Contract — позволяет использовать Consumer Driven Contract (CDC) при разработке микросервисов, а также дает возможность имитировать среду production и провести реальные тесты между микросервисами;
- Spring Cloud Gateway — помогает реализовать API Gateway;
- Spring Cloud Azure — интеграция с облачной платформой Microsoft Azure;
- Spring Cloud for Amazon Web Services — интеграция с облаком Amazon Web Services;
- Spring Cloud Alibaba — интеграция с Alibaba Cloud;
- Spring Cloud Kubernetes — помощь в интеграции с Kubernetes. На самом деле, можно развернуть («задеплоить») приложение Spring Boot в Kubernetes и без этого проекта, но он предоставляет некоторые дополнительные возможности;
- Spring Cloud OpenFeign — интеграция с OpenFeign для реализации декларативных REST-клиентов с помощью аннотаций на интерфейсах.

2.4. Kubernetes

Современное приложение обычно содержит большое количество сервисов, которые нужно не только развернуть, но и поддерживать в рабочем состоянии, следить за их работоспособностью, перезапускать и масштабировать при необходимости.

> ***Сложная тема...***
>
> Kubernetes — это сама по себе отдельная тема, которая может потянуть на полноценную книгу. В этом разделе просто описано, как можно запустить тестовое приложение в Kubernetes без слишком сильного углубления в детали, чтобы не запутать читателя. Если вы только начинаете изучать Spring Framework, то, возможно, не стоит огорчаться из-за того, что многое в этом разделе будет вам непонятно. На начальном этапе это вам не пригодится.

Основные понятия Kubernetes, которые нам понадобятся в этом разделе:

- Node — это физическая машина в кластере Kubernetes;
- Pod — контейнер, его можно рассматривать как один небольшой виртуальный компьютер с установленной операционной системой, нашим приложением и необходимыми зависимостями;
- Persistence Volume — постоянное хранилище, которое могут использовать несколько pod'ов;
- Service — абстракция, которая логически объединяет несколько pod'ов и доступ к ним;
- ReplicaSet — абстракция, поддерживающая указанное количество копий pod'ов в рабочем состоянии, уничтожая и создавая новые при необходимости;
- Deployment — описание желаемого состояния, которое Kubernetes будет поддерживать;

- Kubectl — консольный клиент Kubernetes, с помощью которого мы будем им управлять.

Minikube — это локальный Kubernetes, который обычно используется для разработки и изучения. Именно его вам и нужно будет установить по инструкции для вашей операционной системы[1].

После установки и настройки Minikube и kubectl у вас должны работать команды его запуска и остановки:

```
$ minikube start
$ minikube stop
```

А следующая команда должна отображать версию kubectl:

```
$ kubectl version
```

Также нам будет необходим пакетный менеджер Helm[2].

Наш проект на Spring Boot уже содержит зависимость, которая нужна ему для работы внутри Kubernetes:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Actuator добавляет конечные точки (endpoint): /actuator/health, /actuator/health/liveness и /actuator/health/readiness, по которым Kubernetes отслеживает состояние приложения внутри pod'a. В самом простейшем случае достаточно лишь этой зависимости — никаких дополнительных настроек не требуется.

Вы можете запустить приложение (проект virtualpets-server-springboot) и увидеть статус по **http://localhost:8080/actuator/health**:

```
{
    "status": "UP"
}
```

Поскольку Docker Registry и Minikube хранят пакеты docker image в разных местах, которые никак не связаны, Minikube не сможет просто забрать docker image из того места, где их обычно создает Docker. Поэтому нам нужно настроить Docker так, чтобы он собирал свои docker image в Minikube:

```
$  minikube docker-env
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.49.2:2376"
export DOCKER_CERT_PATH="/home/fedor/.minikube/certs"
export MINIKUBE_ACTIVE_DOCKERD="minikube"

# To point your shell to minikube's docker-daemon, run:
# eval $(minikube -p minikube docker-env)
```

---

[1] См. **https://minikube.sigs.k8s.io/docs/start**.

[2] См. **https://helm.sh**.

Вывод команды `minikube docker-env` подсказывает команду, которую необходимо выполнить:

```
$ eval $(minikube -p minikube docker-env)
```

> ***Настройка Docker на сборку docker image в Minikube***
>
> **Обратите внимание, что команду** `eval $(minikube -p minikube docker-env)` **нужно выполнять в каждом запущенном терминале, в котором вы собираетесь собирать образы Docker.**

Собираем образ Docker с помощью плагина Maven от Spring:

```
./mvnw spring-boot:build-image -Dspring-boot.build-image.imageName=urvanovru/virtualpets-
server-springboot -Dspring-boot.build-image.docker-host=$DOCKER_HOST -Dmaven.test.skip=true
```

Затем необходимо создать `namespace` (пространство имен), в котором мы будем разворачивать сервисы нашего проекта:

```
$ kubectl create namespace urvanovru
```

Все элементы: pod, service, deployment, replicaSet, persistent volume — будут размещаться в этом пространстве имен, для чего к каждой команде необходимо добавлять `--namespace urvanovru`.

Установим PostgreSQL в наш `namespace` с помощью пакетного менеджера Helm:

```
$ helm install pg-minikube --set auth.postgresPassword=postgres bitnami/postgresql --namespace
urvanovru
```

Если мы проверим работающие pod'ы в нашем пространстве имен, то увидим pod с PostgreSQL:

```
$ kubectl get pods --namespace urvanovru

NAME                      READY   STATUS    RESTARTS   AGE
pg-minikube-postgresql-0  1/1     Running   0          38s
```

Для работы сервера виртуальных питомцев необходимо в этом экземпляре PostgreSQL создать схему `virtualpets_server_springboot`, поэтому следует пробросить порт 5432 из pod'а, чтобы была возможность к нему подключиться:

```
$ kubectl port-forward --namespace urvanovru svc/pg-minikube-postgresql 5432:5432
```

Порт 5432 pod'а с `postgresql` будет доступен с нашего компьютера, пока мы не прервем выполнение этой команды с помощью комбинации клавиш <Ctrl>+<C>.

Далее нужно подключиться к базе данных с помощью Eclipse Data Tools Platform, Database tool из IntelliJ IDEA, pgAdmin или DBeaver, используя данные:

```
url: jdbc:postgresql://pg-minikube-postgresql.urvanovru.svc.cluster.local:5432/postgres
username: postgres
password: postgres
database: postgres
```

Для создания схемы `virtualpets_server_springboot` необходимо выполнить команду SQL:

```
CREATE SCHEMA IF NOT EXISTS virtualpets_server_springboot;
```

Всё готово к развертыванию сервера виртуальных питомцев в Minikube. Обратите внимание на каталог kubernetes в каталоге с исходными кодами сервера виртуальных питомцев — именно содержащиеся в нем файлы и будут использоваться для следующих двух команд:

```
$ kubectl apply -f kubernetes/configmap.yaml --namespace urvanovru
$ kubectl apply -f kubernetes/deployment.yaml --namespace urvanovru
```

Если все пройдет без ошибок, то появится новый pod с развернутым приложением.

Если теперь пробросить порт 8080 из нашего pod'а, то на нашем компьютере будет доступна страница **http://localhost:8080/site/home**.

Правильное название pod'а нужно узнать с помощью следующей команды:

```
$ kubectl get pods --namespace urvanovru
```

Скопируйте имя pod'а и подставьте его в команду вместо `virtualpets-server-springboot-85fc79b488-nqndk`:

```
$ kubectl port-forward --namespace urvanovru virtualpets-server-springboot-85fc79b488-nqndk
8080:8080
```

Если что-то не получилось, то вы можете удалить наш `deployment`:

```
$ kubectl delete deployment virtualpets-server-springboot --namespace urvanovru
```

после чего исправить проблемы, а затем снова попытаться его развернуть:

```
$ ./mvnw spring-boot:build-image -Dspring-boot.build-image.imageName=urvanovru/virtualpets-
server-springboot -Dspring-boot.build-image.docker-host=$DOCKER_HOST -Dmaven.test.skip=true
$ kubectl apply -f deployment.yaml --namespace urvanovru
```

> ***Kubernetes необходимо изучать, но не в самом начале***
>
> Нужно понимать, что Kubernetes — это полезная технология, но если вы читаете эту книгу, то, возможно, вам пока не стоит слишком глубоко в него погружаться. Для начала будет полезно хотя бы просто разрабатывать приложения и изучать сам Spring Framework и Spring Boot.

2.5. Резюме

Микросервисная архитектура упрощает масштабирование приложения за счет создания новых экземпляров сервисов на новых компьютерах, pod'ах или контейнерах. К основным недостаткам микросервисной архитектуры можно отнести дополнительное усложнение архитектуры. Spring полностью поддерживает разработку приложений с использованием микросервисной архитектуры.

ГЛАВА 3

Примеры приложения

Предыдущие главы книги были посвящены вопросам преимущественно теоретическим, но одну только теорию без практики усвоить сложно, поэтому все последующие главы, в которых, собственно, и рассматривается Spring Framework и его модули, мы изучим с бóльшим упором на практику.

В рамках этой книги в качестве примера нам послужит приложение сервера игры виртуальных питомцев. Исходные коды его расположены в двух репозиториях на GitHub:

- репозиторий с примером приложения на Spring Framework[1]. Далее в книге этот репозиторий и проект в нем мы будем для краткости называть virtualpets-server-springframework;
- репозиторий с примером приложения на Spring Boot[2] — его при дальнейших упоминаниях мы будем называть virtualpets-server-springboot.

Исходные коды клиентской части приложения написаны на JavaScript и также расположены на GitHub[3].

3.1. Пример приложения на Spring Framework

В этом разделе мы рассмотрим — для дальнейшей разработки и изучения — развертывание и запуск примера приложения из репозитория virtualpets-server-springframework на локальном компьютере. Вы можете задействовать Java IDE: Eclipse, IntelliJ IDEA, NetBeans или любую другую среду. Использование Spring Framework требует от разработчика довольно обширных знаний о мире информационных технологий, поэтому вам нужно хорошо понимать, как работать в вашей IDE с обычными приложениями, — только тогда вы сможете полноценно изучать Spring и разрабатывать на нем.

---

[1] См. **https://github.com/urvanov-ru/virtualpets-server-springframework**.

[2] См. **https://github.com/urvanov-ru/virtualpets-server-springboot**.

[3] См. **https://github.com/urvanov-ru/virtualpets-client-js**.

Какую бы среду вы ни выбрали, первоначально вам необходимо клонировать проект из репозитория на GitHub и импортировать его на свой компьютер в качестве Maven-проекта. Импорт проекта подробно описан мною в книге «Java. Состояние языка и его перспективы»[1]. Вы также можете найти примеры импорта проекта с GitHub в популярные IDE на моем сайте **https://urvanov.ru**.

Для запуска проекта вам необходимо дополнительно установить Docker Desktop с сайта **https://docker.com**. Приложение Docker Desktop при разработке необходимо всё время держать запущенным, потому что контейнеры Docker будут использоваться как тестами, так и самим запущенным приложением.

Когда Docker Desktop запущен, и код приложения открыт в IDE, вам необходимо либо запустить контейнер с PostgreSQL, описанный в файле docker-compose.yml, находящемся в подкаталоге docker корневого каталога проекта, либо самостоятельно установить PostgreSQL и создать новую схему virtualpets_server_springframework, которую будет использовать наш тестовый проект с сервером виртуальных питомцев.

Лучше всего воспользоваться именно файлом docker-compose.yml (листинг 3.1), поскольку это гарантированно запустит как раз ту версию PostgreSQL, с которой проект сможет работать, хотя он должен запускаться практически со всеми версиями этой базы данных, т. к. не использует никаких специфичных конструкций из SQL.

Листинг 3.1. Файл docker-compose.yml

```
version: '3'
services:
  postgres:
    image: postgres:16.1
    ports:
      - 5432:5432
    environment:
      POSTGRES_DB: "postgres"
      POSTGRES_USER: "postgres"
      POSTGRES_PASSWORD: "postgres"
      PGDATA: "/var/lib/postgresql/data/pgdata"
      POSTGRES_INITDB_ARGS: "--encoding=UTF-8"
    volumes:
        - ./init.sql:/docker-entrypoint-initdb.d/init.sql
        - ./data/postgres:/var/lib/postgresql/data
```

Как видите, в этом файле указан `image` нашего PostgreSQL версии 16.1, настроен проброс порта 5432 наружу, чтобы сервер виртуальных питомцев смог до него достучаться, когда мы будем его запускать из IDE, а затем передаются переменные окружения, в которых задаются название базы данных, логин и пароль пользователя, каталог `pgdata` для хранения данных и кодировка по умолчанию.

---

[1] См. **https://bhv.ru/product/java-sostoyanie-yazyka-i-ego-perspektivy/**.

Скрипт `init.sql`, который «прокидывается» в создаваемый контейнер, создает схему virtualpets_server_springframework в базе данных postgres. С этой схемой virtualpets_server_springframework будет работать как тестовое приложение.

Итак, находясь в каталоге с файлом docker-compose.yml, выполните команду:

```
docker compose up
```

Она выкачает `image` с PostgreSQL и запустит его согласно описанию, приведенному в этом файле.

После выполнения описанных действий у вас должен быть запущен PostgreSQL-сервер внутри docker-контейнера с проброшенным портом 5432 наружу — так что мы можем к нему подключиться, например, с помощью Eclipse Data Tools Platform, Database tool из IntelliJ IDEA, pgAdmin, DBeaver или какой-нибудь другой утилиты для работы с базой данных.

Для нашего приложения нужен экземпляр Apache Tomcat 10. Скачайте архив с ним с официального сайта[1] и распакуйте его в какой-нибудь каталог.

Необходимо также настроить пул соединений, который будет использоваться тестовым приложением. Внутри каталога с Apache Tomcat 10 найдите подкаталог conf — в нем содержатся конфигурационные файлы, и именно их использует IntelliJ IDEA.

Пул соединений представляет собой кеш соединений с базой данных. Дело в том, что открытие нового соединения для каждого запроса или каждого пользователя ресурсоемко, а работа с пулом соединений позволяет брать уже существующее свободное соединение, использовать его, а потом возвращать обратно в пул, не закрывая, что увеличивает суммарную производительность приложения, т. к. при этом отсутствуют накладные расходы на открытие новых соединений и закрытие старых.

Подробную информацию о настройке Apache Tomcat можно найти на его официальном сайте — нам же для нашего примера потребуется внести два изменения в файл context.xml.

> ***В реальности настройки Apache Tomcat могут храниться отдельно***
>
> Apache Tomcat не обязательно содержит конфигурационные файлы в том же каталоге, что и свои исполняемые файлы. Он поддерживает запуск нескольких экземпляров, каждый из которых имеет свои настройки, но при этом сам Apache Tomcat может находиться в другом каталоге. На каталог с исполняемыми файлами Apache Tomcat указывает переменная окружения CATALINE_HOME, а на каталог с конфигурационными файлами экземпляра — переменная окружения CATALINA_BASE. При этом конфигурационные файлы для каждого запущенного экземпляра Apache Tomcat будут находиться в своем каталоге CATALINE_BASE, как показано на рис. 3.1. Значения этим переменным окружения обычно присваиваются в скриптах запуска.

Настройку пула соединений рекомендуется осуществлять в конфигурационных файлах экземпляра — т. е. по пути CATALINA_BASE. Подробно настройка пула

[1] См. **https://tomcat.apache.org**.

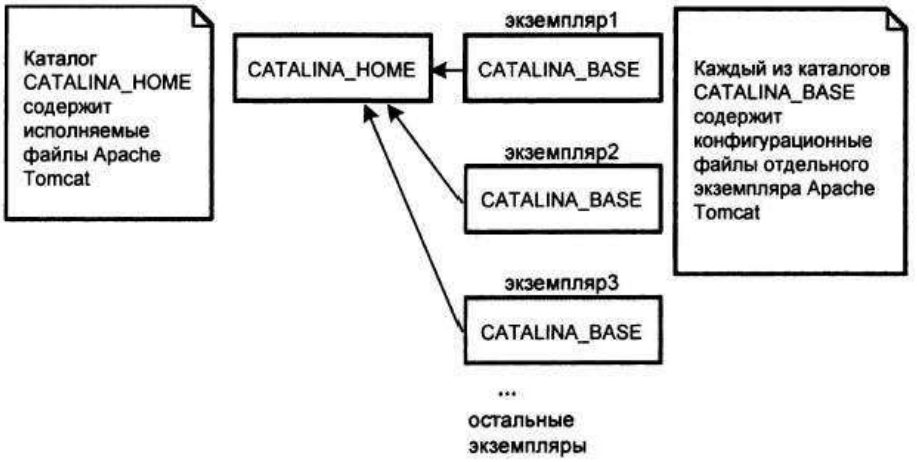


Рис. 3.1. Несколько экземпляров Apache Tomcat и переменные окружения CATALINA_HOME и CATALINA_BASE

соединений описана далее отдельно для Eclipse и отдельно для IntelliJ IDEA. Надо также учесть, что схемы запуска проекта в разных IDE различаются.

3.1.1. Настройка пула соединений для Eclipse

В Eclipse или Spring Tool Suite необходимо найти представление Servers (рис. 3.2).

Рис. 3.2. Представление **Servers** в Eclipse

> ***Открытие представления Servers***
>
> Если представления **Servers** нет либо вы его закрыли, то его можно открыть из горизонтального меню **Window | Show View | Other**, после чего выбрать **Servers**.

В представлении **Servers** (см. рис. 3.2) щелкните левой кнопкой мыши на надписи **No servers are available. Click this link to create a new server...** либо щелкните правой кнопкой на пустом месте в нем и выберите в открывшемся контекстном меню **New | Server**.

В поле ввода типа сервера открывшегося окна **New Server** (рис. 3.3) введите `Tomcat` (поз. 1), выберите из списка **Tomcat v10.1 Server** (поз. 2), введите имя экземпляра сервера `tomcat10` (поз. 3) и нажмите кнопку **Next >**. Введенное вами имя `tomcat10` — это имя экземпляра сервера, на этот экземпляр и будет указывать переменная CATALINA_BASE.

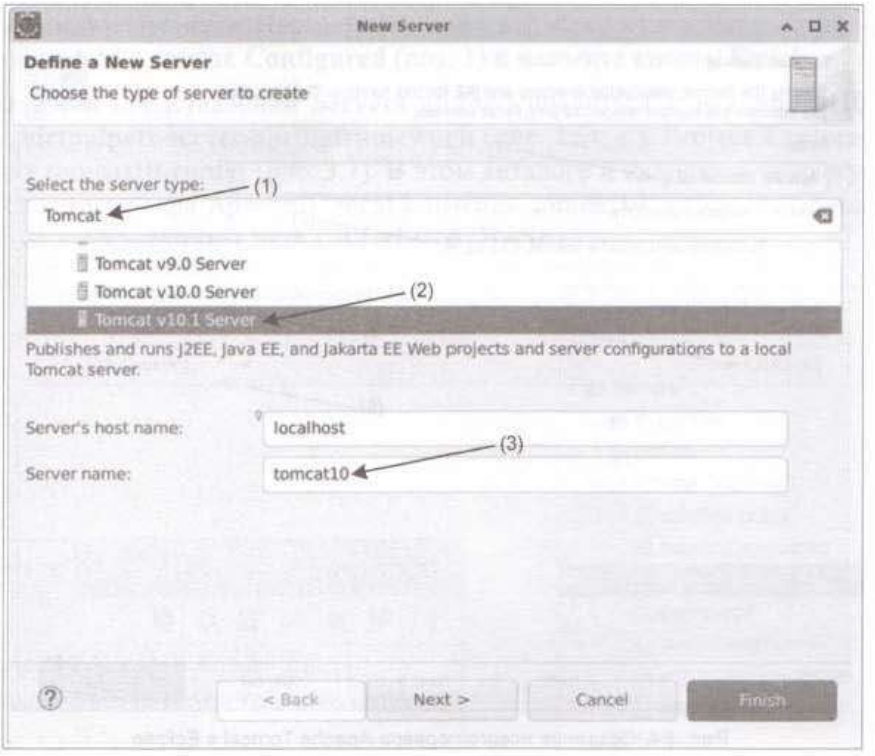


Рис. 3.3. Создание нового экземпляра сервера Apache Tomcat в Eclipse: выбор типа сервера

В открывшемся окне создания нового сервера Apache Tomcat (рис. 3.4) введите имя сервера `Apache Tomcat 10.1` (поз. 1), выберите каталог, в который установлен Apache Tomcat (поз. 2), версию JDK, с которой будет запускаться создаваемый экземпляр (поз. 3), и нажмите кнопку **Next >**. Сервер Apache Tomcat — это тот сервер, на основе которого создается наш экземпляр сервера, — на него будет указывать переменная CATALINA_HOME.

> ***Apache Tomcat можно скачать на этом шаге***
>
> Скачать Apache Tomcat можно с помощью кнопки **Download and Install** (поз. 4 на рис. 3.4), если вы не сделали этого самостоятельно с сайта **https://tomcat.apache.org**.

На самом деле, версию Java можно оставить как Workspace Default JRE, поскольку последние версии Eclipse уже работают на подходящей версии, но вы можете скачать и установить любую версию, а затем добавить ее в список с помощью **кнопки Installed JRE's** (поз. 5 на рис. 3.4), после чего эту версию Java можно будет выбрать в списке.

Введенное вами имя **Apache Tomcat 10.1** — это имя сервера, на его основе можно создать несколько экземпляров серверов с разными именами, подобно тому, как мы ранее создали экземпляр сервера с именем **tomcat10** (см. рис. 3.3).

В открывшемся по нажатию кнопки **Next >** окне добавления приложений для развертывания на создаваемый экземпляр сервера tomcat10 (рис. 3.5) выберите из

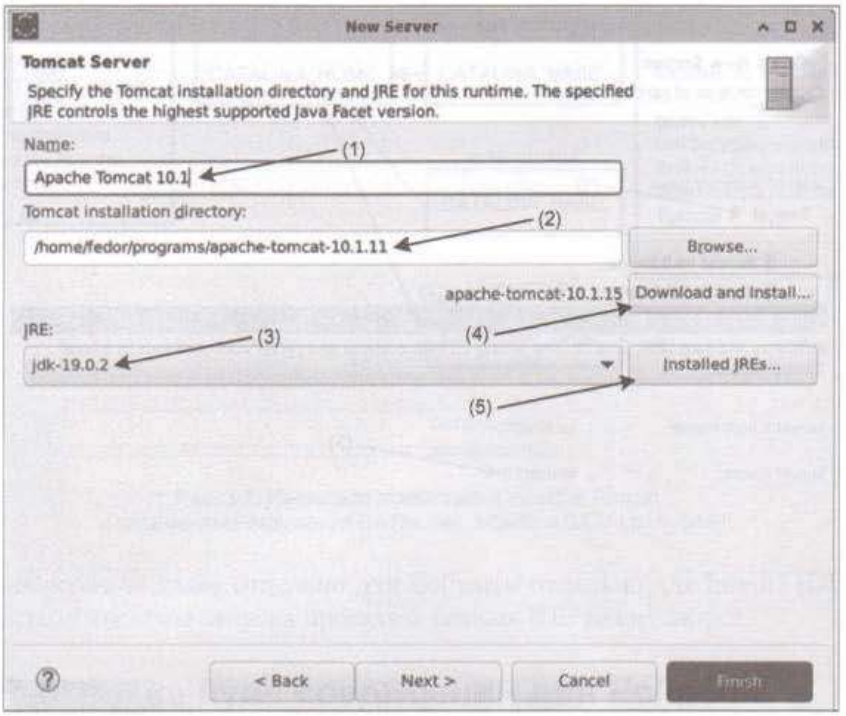


Рис. 3.4. Создание нового сервера Apache Tomcat в Eclipse

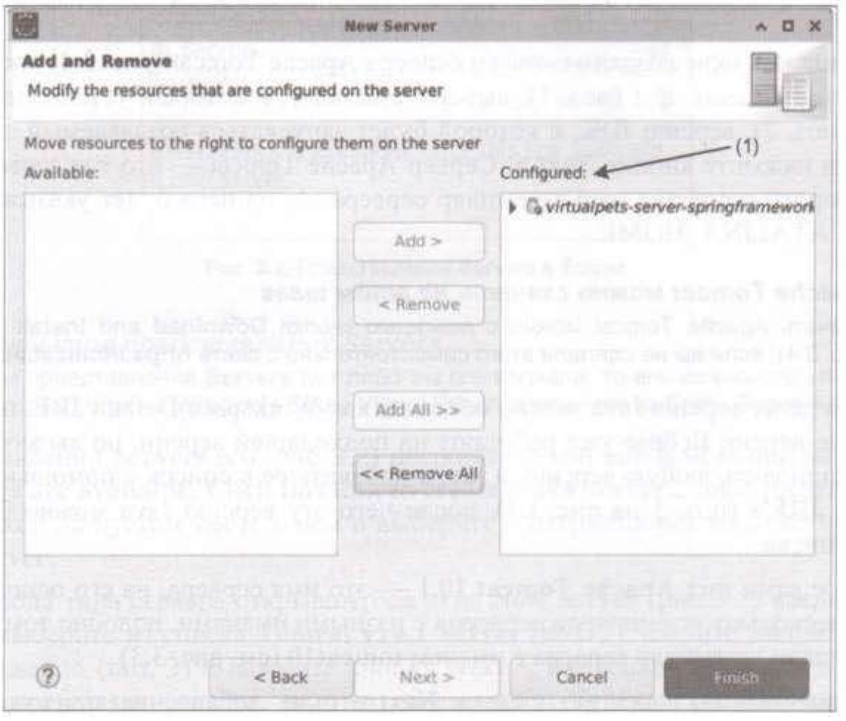


Рис. 3.5. Выбор приложений для развертывания на экземпляр Apache Tomcat 10.1 с именем tomcat10

списка **Available** запись **virtualpets-server-springframework**, добавьте ее с помощью кнопки **Add** в список **Configured** (поз. 1) и нажмите кнопку **Finish**.

В результате в представлении **Servers** должен появиться пункт **tomcat10** с подпунктом **virtualpets-server-springframework** (рис. 3.6), а в **Project Explorer** — новый пункт **tomcat10-config** (рис. 3.7). В этом каталоге и находятся конфигурационные файлы экземпляра Apache Tomcat с именем **tomcat10**, который мы только что создали (на него указывает путь CATALINA_BASE).

Рис. 3.6. Созданный экземпляр сервера tomcat10 с развернутым приложением virtualpets-server-springframework

Рис. 3.7. Расположение файла context.xml, используемого в Eclipse экземпляром Apache Tomcat с именем tomcat10

Осталось прописать в файле context.xml настройки пула соединений, как это было отмечено ранее, — добавьте узел `Resource` в блок `Context` этого файла (листинг 3.2).

Листинг 3.2. Добавление узла `Resource` в блок `Context` файла context.xml

```
...
<Context>
    ...
    <Resource
        auth = "Container"
        driverClassName = "org.postgresql.Driver"
        maxIdle = "10"
        maxTotal = "20"
        maxWaitMillis = "-1"
        name = "jdbc/virtualpetsDB"
        password = "postgres"
        type = "javax.sql.DataSource"
        url = "jdbc:postgresql://localhost:5432/postgres?currentSchema=
                                                    virtualpets_server_springframework"
        username = "postgres"/>
    ...
</Context>
```

Здесь мы описали JNDI-ресурс с именем `jdbc/virtualpetsDB`, который будем использовать в нашем тестовом приложении.

> ***Java Name and Directory Interface (JNDI)***
> JNDI-ресурс позволяет ассоциировать наименование объекта с экземпляром и получать этот объект по его имени.

Основные используемые в узле `Resource` параметры:

- `url` — интернет-адрес (URL) подключения к базе данных;
- `username` — логин к базе данных;
- `password` — пароль к базе данных;
- `driverClassName` — класс драйвера JDBC;
- `maxTotal` — максимальное количество соединений в пуле.

Более подробную информацию о JNDI-ресурсах можно получить в документации Apache Tomcat, расположенной на его официальном сайте.

Тестовому приложению сервера виртуальных питомцев также понадобится драйвер PostgreSQL JDBC[1] — скачайте его и поместите скачанный файл в подкаталог lib к остальным jar-файлам Apache Tomcat, чтобы загрузчик классов смог его найти при запуске.

На этом настройка пула соединений для Eclipse завершена.

Запуск в режиме отладки выбранного экземпляра сервера tomcat10 и остановка этого экземпляра осуществляются соответственно кнопками  (поз. 1) и  (поз. 2) на панели инструментов в представлении **Servers** (рис. 3.8).

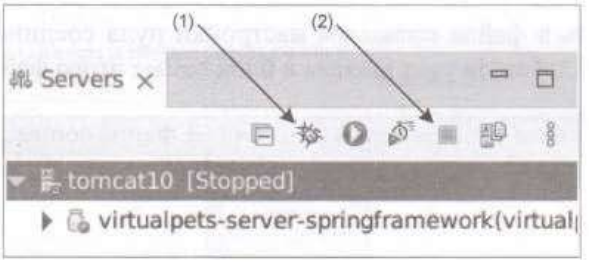


Рис. 3.8. Кнопки запуска и остановки экземпляра tomcat10

Для запуска выберите **tomat10** и щелкните на кнопке с изображением жука (поз. 1) на панели инструментов.

Если всё пройдет успешно, то в консоли в IDE сообщения об ошибках не появятся, а по локальному адресу[2] должна открываться та же страница, что доступна в Интернете[3].

---

[1] См. **https://jdbc.postgresql.org/**.

[2] См. **http://localhost:8080/virtualpets-server-springframework/site/home**.

[3] См. **http://virtualpets.urvanov.ru/virtualpets-server-framework/site/home**.

3.1.2. Настройка пула соединений для IntelliJ IDEA

В IntelliJ IDEA для добавления конфигурации запуска проекта с Apache Tomcat выполните в горизонтальном меню команду **Run | Edit configurations**, в открывшемся диалоговом окне **Run/Debug Configurations** (рис. 3.9) щелкните на крестике в его левом верхнем углу и в открывшейся панели, показанной на рис. 3.10, выберите команду запуска на локальном сервере: **Tomcat Server | Local**.

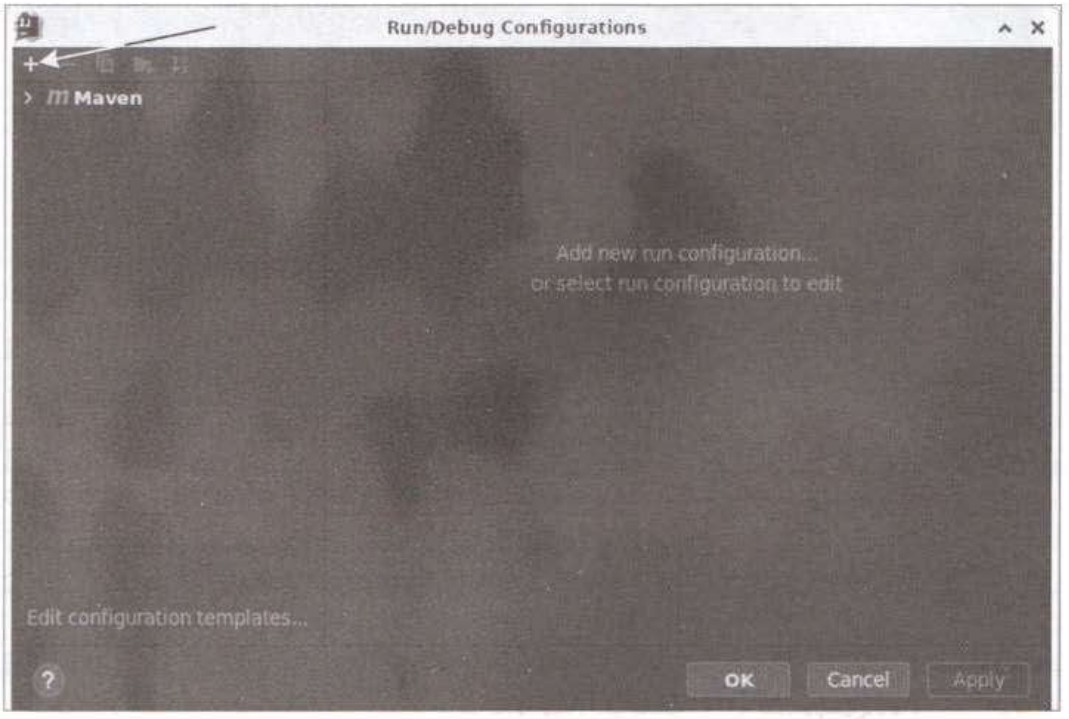


Рис. 3.9. Диалоговое окно **Run/Debug Configurations** в IntelliJ IDEA

Чтобы настроить расположение Apache Tomcat, в открывшейся в правой части окна **Run/Debug Configurations** панели (рис. 3.11) нажмите на кнопку **Configure** и в открывшемся окне **Application Servers** (рис. 3.12) выберите пути к каталогам с настроенным ранее Apache Tomcat, имея в виду следующее:

- путь, прописанный в поле **Tomcat Home** (поз. 1), — это CATALINA_HOME;
- путь, прописанный в поле **Tomcat base directory** (поз. 2), — это CATALINA_BASE.

Завершите выбор путей щелчком на кнопке **OK**.

При этом в каталог, указанный в поле **Tomcat base directory**, необходимо скопировать каталог conf из каталога установки Apache Tomcat так, чтобы внутри каталога CATALINA_BASE появился подкаталог conf со всеми конфигурационными файлами.

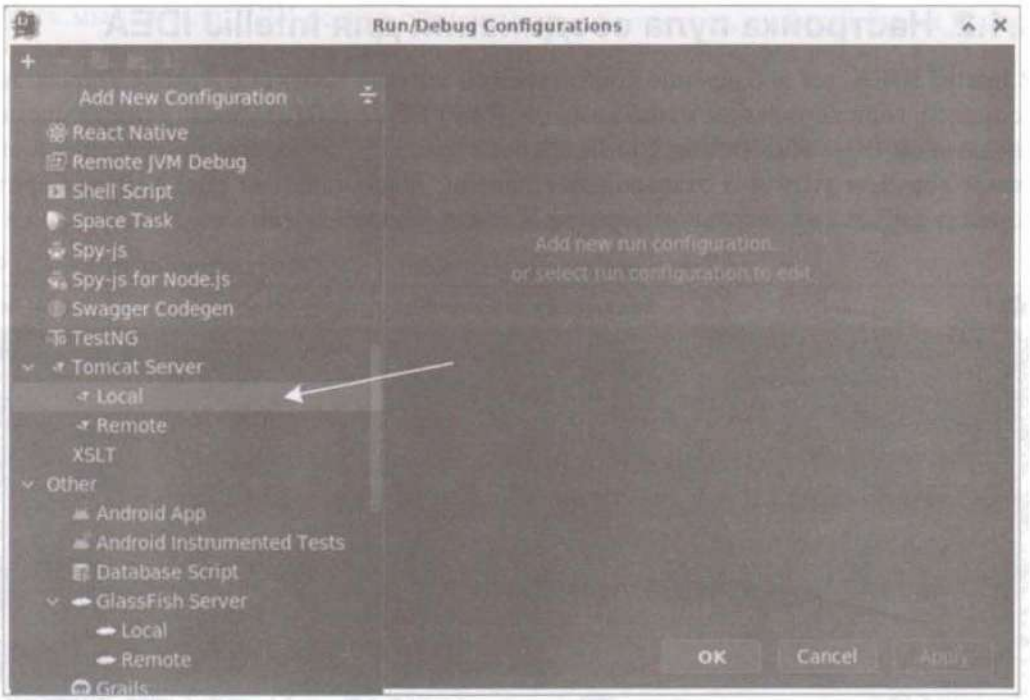


Рис. 3.10. Добавление конфигурации Tomcat Server Local

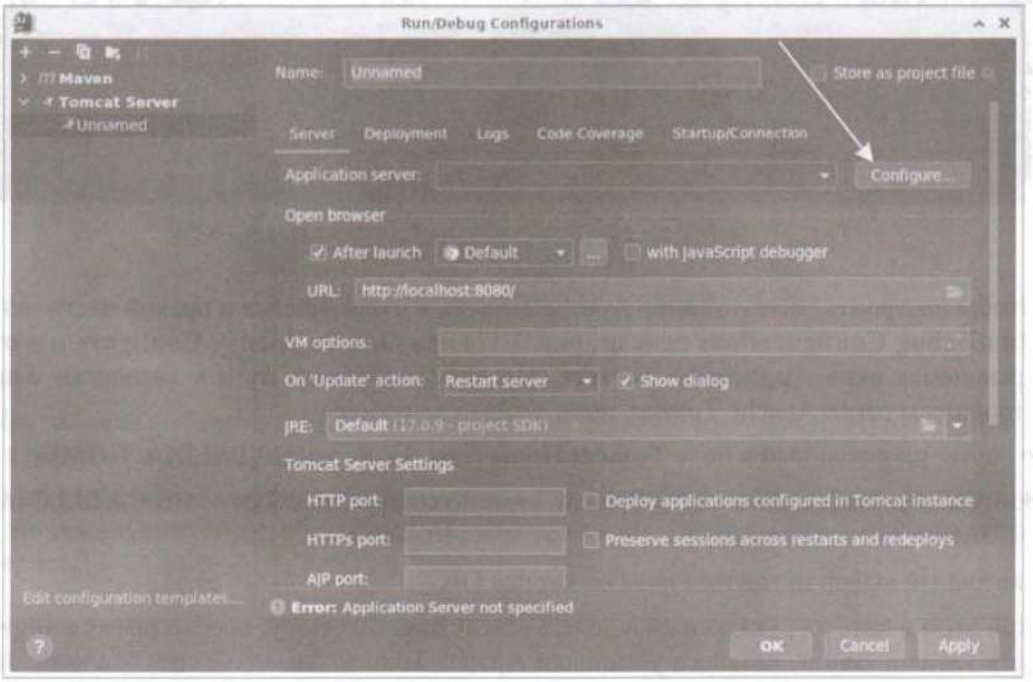


Рис. 3.11. Нажмите здесь на кнопку Configure

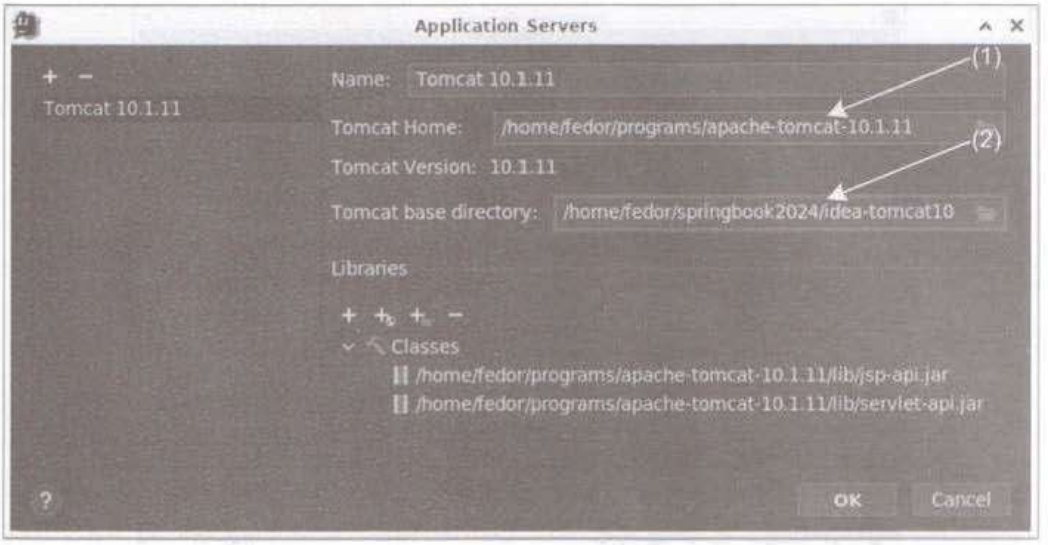


Рис. 3.12. Настройка путей CATALINA_HOME и CATALINA_BASE в IntelliJ IDEA

В скопированном подкаталоге conf найдите файл context.xml и пропишите в нем настройки пула соединений так же, как это было указано для файла context.xml в описании настройки Eclipse и Spring Tool Suite (см. *разд. 3.1.1*).

Вернитесь теперь в окно **Run/Debug Configurations** (рис. 3.13) и выполните там следующие действия:

1. Введите название конфигурации запуска: `tomcat10` (поз. 1).
2. Убедитесь (поз. 2), что в поле **Application Server** выбран **Tomcat 10.1.11** из предыдущего шага.
3. Для открытия проекта в браузере после запуска выберите в поле **URL** интернет-адрес локального сервера[1] (поз. 3).
4. Выберите JRE обязательно версии 17 или выше (поз. 4).

Далее перейдите в окне **Run/Debug Configurations** (рис. 3.14) на вкладку **Deployment** (поз. 1), нажмите в разделе **Deploy at the server startup** кнопку «плюс» (поз. 2) и добавьте в открывшийся список артефактов для развертывания (поз. 3) артефакт **virtualpets-server-springframework:war exploded**. Фрагмент **exploded** здесь означает, что для Tomcat будет указан не war-файл, а его распакованное содержимое, что облегчит обновление кода при его изменении.

И обязательно укажите в поле **Application context** значение **/virtualpets-server-springframework**, как показано на рис. 3.15.

В завершение настройки нажмите в окне **Run/Debug Configurations** на кнопку **OK**, после чего новая конфигурация запуска должна быть доступна в панели инструментов (рис. 3.16).

---

[1] **http://localhost:8080/virtualpets-server-springframework/site/home**.

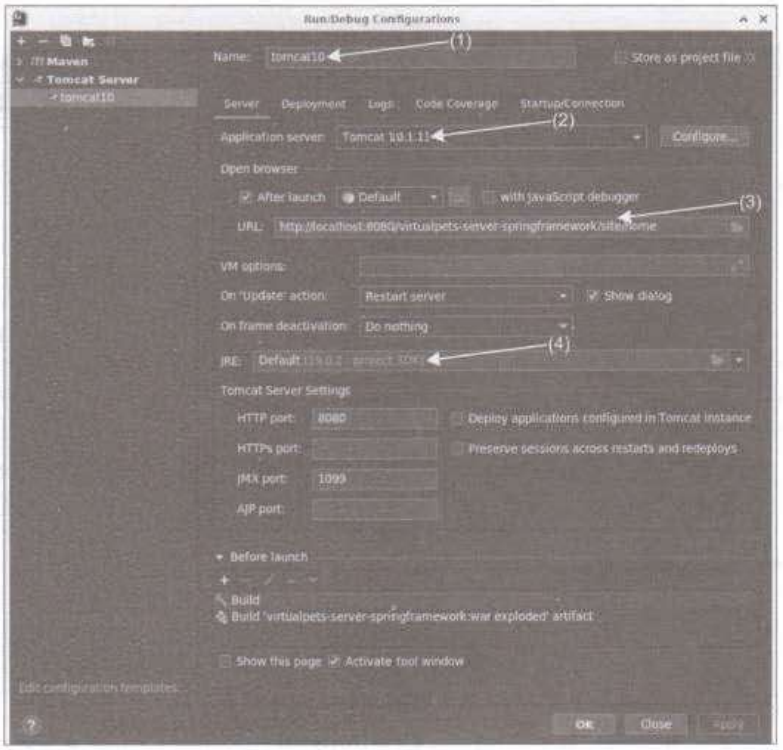


Рис. 3.13. Параметры конфигурации tomcat10

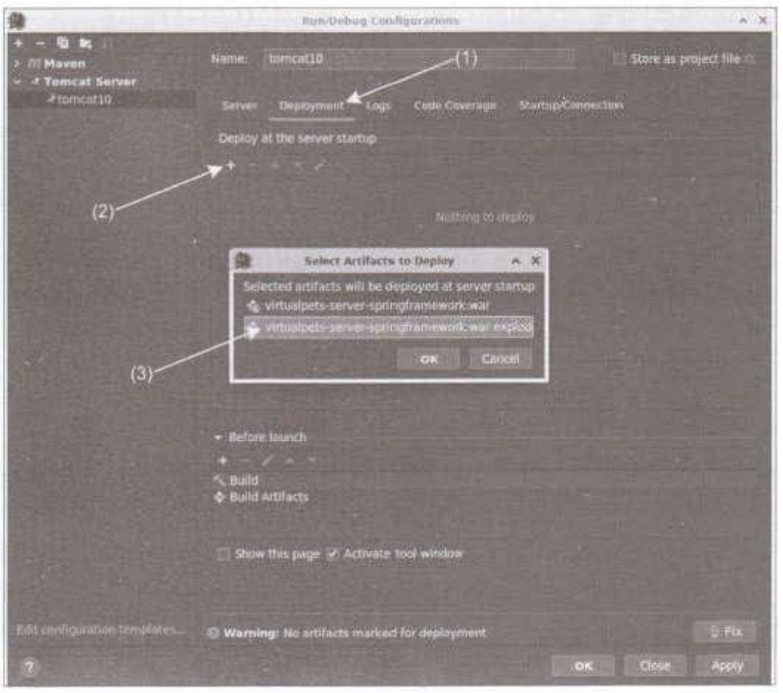


Рис. 3.14. Выбор артефакта для деплоя

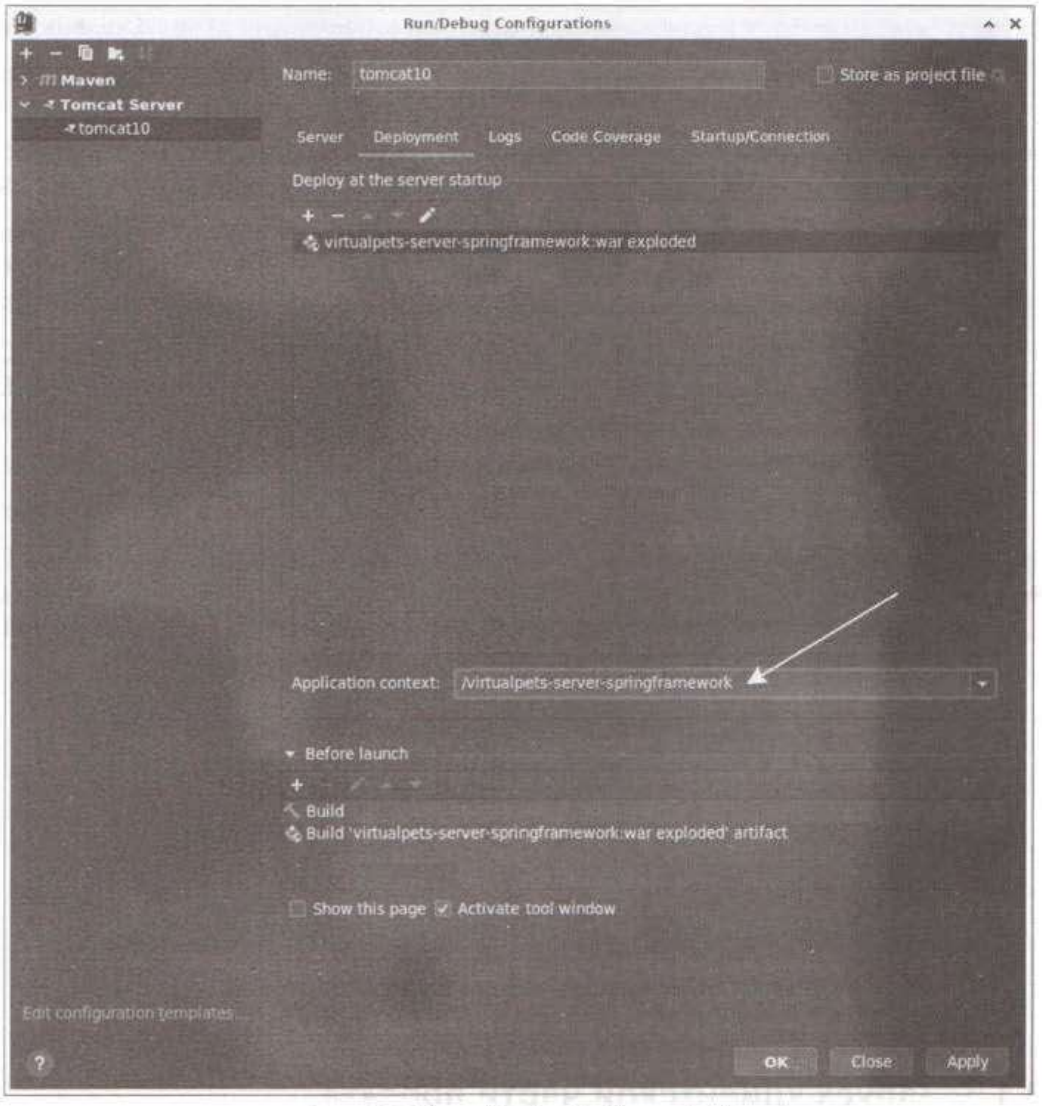


Рис. 3.15. Настройка **Application context**

Рис. 3.16. Панель инструментов управления конфигурациями запуска

Если запуск конфигурации пройдет успешно, то по локальному адресу[1] должна открываться та же страница, что доступна в Интернете[2] (рис. 3.17).

---

[1] См. **http://localhost:8080/virtualpets-server-springframework/site/home**.

[2] См. **http://virtualpets.urvanov.ru/virtualpets-server-framework/site/home**.

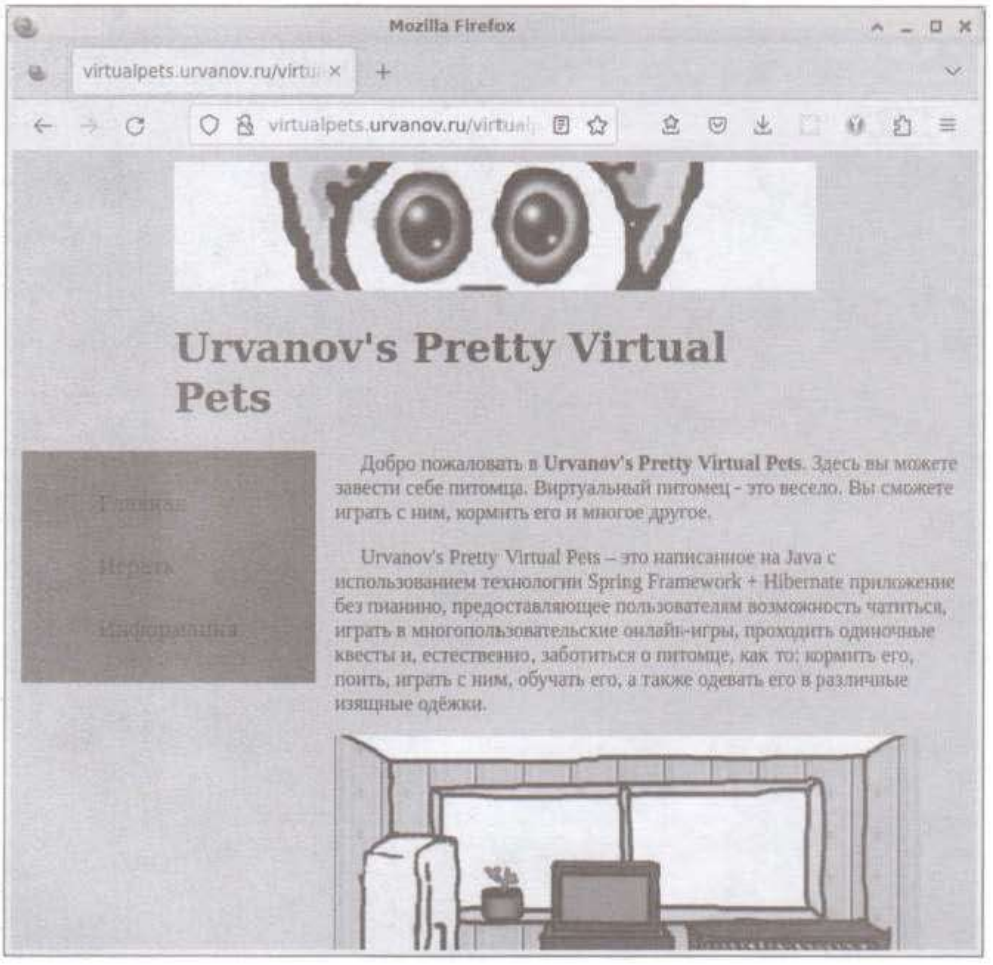


Рис. 3.17. Страница сайта сервера виртуальных питомцев

3.1.3. Запуск клиентской части проекта

После успешного запуска бэкенда осталось только запустить клиентскую часть с игрой, чтобы полностью убедиться в работоспособности тестового приложения. Для запуска клиентской части необходимо установить Node.js, т. к. для сборки потребуется пакетный менеджер npm, входящий в его состав. Последнюю версию Node.js можно найти на его официальном сайте[1].

> ***JavaScript, Node.js и пр.***
>
> В рамках этой книги работа с JavaScript, Node.js, npm и другими технологиями, связанными с клиентской частью приложений, рассматриваться не будет. Просто выполните указанные далее команды.

[1] См. **https://nodejs.org/**.

Установив Node.js, склонируйте репозиторий с исходными кодами на JavaScript:

```
git clone https://github.com/urvanov-ru/virtualpets-client-js.git
cd virtualpets-client-js
```

В каталоге со склонированным репозиторием должен находиться файл package.json. Из каталога, где содержится этот файл, запустите сборку проекта командами:

```
npm install
npm run build-development-springframework
```

Эти команды выкачают необходимые зависимости и соберут проект в каталоге dist.

Обратите внимание на файл docker-compose.yml, находящийся в том же каталоге. В нем описан контейнер docker, который запустит nginx с содержимым из каталога dist и откроет доступ к нему по адресу **http://localhost:8081**.

Запустим nginx с клиентской частью:

```
docker compose up
```

После запуска откройте браузер и перейдите по адресу **http://localhost:8081**. У вас должна открыться страница с выбором языка и кнопкой **Играть**, аналогичная странице, доступной в Интернете[1] (рис. 3.18).

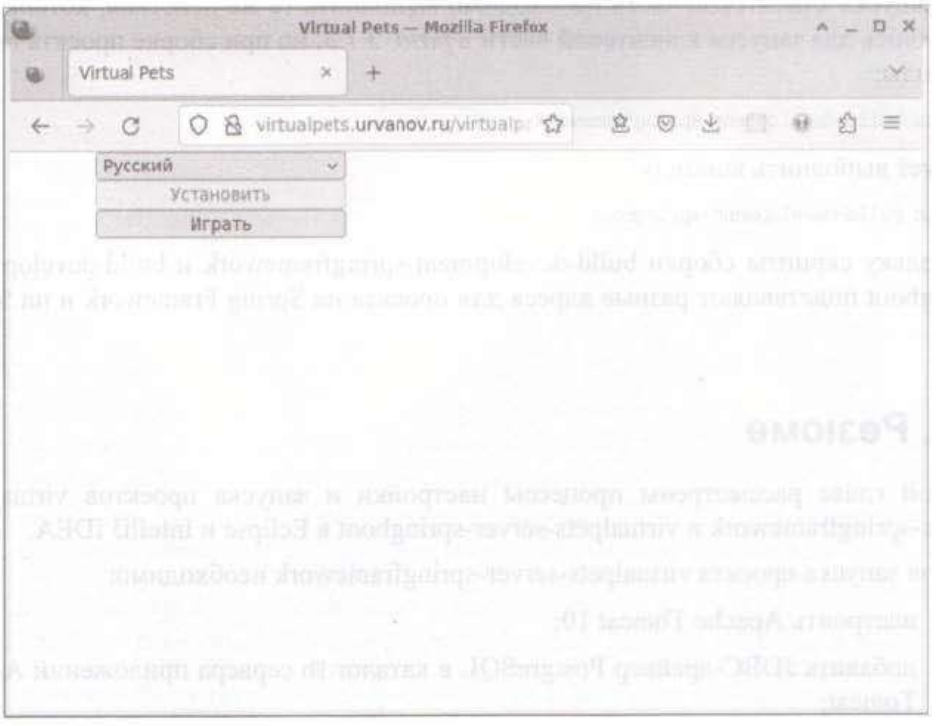


Рис. 3.18. Первый экран клиентской части игры на JavaScript

[1] См. **http://virtualpets.urvanov.ru/virtualpets-client**.

По умолчанию существует пользователь с логином admin и паролем 123 — его вам вполне хватит для тестирования, но вы всегда можете зарегистрировать нового пользователя с помощью соответствующей кнопки на странице логина.

3.2. Пример приложения на Spring Boot

Пример на Spring Boot, как уже отмечалось в начале главы, находится в репозитории на GitHub[1], напомню также, что в дальнейшем этот репозиторий и проект в нем для краткости мы будем называть virtualpets-server-springboot.

Приложение на Spring Boot запускается точно так же, как и обычное Java-приложение, но сначала вам нужно перейти в каталог docker и запустить базу PostgreSQL:

```
$ cd docker
$ docker compose up
```

Когда в консоли появится сообщение об успешном запуске, щелкните правой кнопкой мыши на virtualpets-server-springboot в дереве проектов и выберите вариант **Debug As... Java application.**

Для запуска клиентской части необходимо выполнить те же действия, которые выполнялись для запуска клиентской части в *разд. 3.1.3*, но при сборке проекта вместо команды:

```
npm run build-development-springframework
```

следует выполнить команду:

```
npm run build-development-springboot
```

поскольку скрипты сборки build-development-springframework и build-development-springboot подставляют разные адреса для проекта на Spring Framework и на Spring Boot.

3.3. Резюме

В этой главе рассмотрены процессы настройки и запуска проектов virtualpets-server-springframework и virtualpets-server-springboot в Eclipse и IntelliJ IDEA.

- Для запуска проекта virtualpets-server-springframework необходимо:
  - настроить Apache Tomcat 10;
  - добавить JDBC-драйвер PostgreSQL в каталог lib сервера приложений Apache Tomcat;
  - задействовать JDK версии 17 или выше;

[1] См. **https://github.com/urvanov-ru/virtualpets-server-springboot**.

- воспользоваться PostgreSQL (можно установить в контейнере docker);
- настроить пул подключений к PostgreSQL в Apache Tomcat;
- применить IDE на выбор: Eclipse, IntelliJ IDEA либо любую другую.

♦ Для запуска проекта virtualpets-server-springboot необходимо:

- задействовать JDK версии 17 или выше;
- воспользоваться PostgreSQL (можно установить в контейнере docker);
- применить IDE на выбор: Eclipse, IntelliJ IDEA либо любую другую.

♦ Для запуска JavaScript-клиента необходимо воспользоваться PostgreSQL (можно установить в контейнере docker).

ГЛАВА 4

Первые шаги

4.1. Spring и контейнер бинов

Spring Framework управляет жизненным циклом приложения (созданием, удалением), внедрением зависимостей, аспектами, а также осуществляет всю свою остальную функциональность с помощью *контейнера бинов* Spring IoC, базовым интерфейсом которого является `BeanFactory` из пакета `org.springframework.beans.factory` и его дочерний интерфейс `ApplicationContext` из пакета `org.springframework.context`.

> ***Бины***
>
> *Бином* называется любой класс, помещенный в контейнер Spring IoC.

> ***Контейнеры...***
>
> Контейнер бинов, контейнер Spring, контейнер бинов Spring, контейнер Spring IoC, контейнер бинов Spring IoC — это всё одно и то же.

Инициализация контейнера может производится двумя способами:

- используя XML-файлы с конфигурацией;
- с помощью Java-конфигурации.

В традиционном варианте инициализация производится именно с помощью XML-конфигурации — как это сделано в проекте virtualpets-server-springframework, рассматриваемом в качестве примера для этой книги.

Новые приложения в современном мире обычно используют Java-конфигурацию, пример которой можно увидеть в примере проекта virtualpets-server-springboot.

> ***Spring Boot и Spring Framework не привязаны к Java и XML***
>
> И Spring Framework, и Spring Boot поддерживают конфигурацию как в XML, так и в Java-варианте. Если вы создаете приложение на Spring Boot, то это еще не значит, что вы не можете использовать XML-файлы с конфигурациями. Аналогично вы можете использовать как Java, так и XML-конфигурацию и для варианта со Spring Framework без Spring Boot.

Коммерческое приложение на Spring практически всегда представляет собой веб-приложение, но это не значит, что с помощью Spring нельзя создавать консольные или десктопные приложения.

Инициализировать контекст Spring IoC можно самостоятельно:

```
// Инициализация контекста Spring IoC
ApplicationContext context
        = new ClassPathXmlApplicationContext("context.xml");
// Получение бина petService из контекста
PetService service = context.getBean("petService", PetService.class);
```

Имейте в виду, что переменная `service` будет ссылаться не на сам экземпляр класса `PetService`, а именно на бин. При вызове методов будут вызываться не методы класса напрямую, а методы бина, что позволяет контейнеру Spring IoC осуществлять дополнительные действия до, после или даже вместо вызова этого метода.

Файл context.xml может выглядеть примерно так:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:context = "http://www.springframework.org/schema/context"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
    ">
    <bean id = "petService"
        class = "ru.urvanov.virtualpets.service.PetServiceImpl">

    </bean>
...
</beans>
```

Следующая инструкция:

```
service.method1();
```

приведет к вызову метода бина, который после выполнения своих действий вызовет метод класса `PetService`.

В реальной жизни вам никогда не придется это делать, поскольку с большой долей вероятности вы будете разрабатывать веб-сервисы, предоставляющие API для фронтенда или для других сервисов, где инициализация контекста происходит не с помощью самостоятельной инициализации `ApplicationContext`, а на основе конфигурации в файле web.xml или автоконфигурации Spring Boot.

Каждый бин имеет свое имя либо несколько имен. Имя бина задается либо через атрибут `id`, либо через атрибут `name`. Атрибут `id` позволяет задать только одно имя, тогда как через атрибут `name` можно указать несколько имен, разделяя их запятой, точкой с запятой либо пробелом.

> ***Соглашение об именовании бинов***
>
> Имя бина может содержать любые символы, но согласно соглашению об именовании бинов для их имен действуют те же правила, что и при именовании переменных. Примеры: `petService`, `moneyService`, `accountTransactionService`.

Если не указывать имя бина, то Spring сгенерирует его автоматически, но если вы собираетесь ссылаться на бин по имени при связывании его с другим бином, то рекомендуется указывать имя бина вручную, а не полагаться на автосгенерированное.

4.2. Простой сервис на Spring Framework

4.2.1. Скачайте исходные коды

В большей части следующих глав книги будет рассматриваться пример сервера виртуальных питомцев, но сразу приступать к изучению большого проекта очень сложно, и в этой главе мы рассмотрим самый минимальный проект на Spring Framework, который вы уже можете создать самостоятельно.

Исходные коды проекта доступны на GitHub[1], но вы также можете скачать ZIP-архив из раздела моего сайта, посвященного этой книге[2]:

4.2.2. Пояснения к исходному коду

Проект имеет название simple-spring-webmvc и использует Maven в качестве менеджера зависимостей.

Подключена одна основная зависимость:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${org.springframework-version}</version>
</dependency>
```

где `spring-webmvc` — та самая основная мавен-зависимость Spring MVC, позволяющая создавать веб-приложения.

Особое внимание следует обратить на каталог src/main/webapp/WEB-INF, содержащий два файла:

- context.xml;
- web.xml.

Файл web.xml — стандартный дескриптор развертывания веб-приложения Jakarta EE (ранее — Java EE). Описание допустимых тегов и XSD-схему можно найти на сайте Jakarta EE[3] .

В нашем же случае этот файл начинается с добавления `ContextLoadListener` в список listener'ов:

```
<listener>
    <listener-class>
```

---

[1] См. **https://github.com/urvanov-ru/simple-spring-webmvc**.

[2] См. **https://urvanov.ru/книги/spring-book-2024/**.

[3] См. **https://jakarta.ee/**.

```
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

`ContextLoadListener` позволит нам инициализировать контекст `WebApplicationContext` с помощью XML-файлов.

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/context.xml
    </param-value>
</context-param>
```

Далее в файле идет описание сервлета, в котором указан `DispatcherServlet` из Spring Framework в качестве класса обработчика. Именно `DispatcherServlet` перенаправляет запросы к контроллеру, который их обрабатывает:

```
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>appServlet</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

В файле web.xml может быть описано несколько сервлетов, которые будут разделяться по обрабатываемым адресам с помощью servlet-mapping. В нашем простом примере задействован только один сервлет.

Обратите внимание, что файле web.xml упоминается файл context.xml как несущий XML-конфигурацию. В начале этого файла подключены пространства имен и указаны схемы XSD, по которым описан файл (листинг 4.1).

Листинг 4.1. Начало файла context.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans:beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans = "http://www.springframework.org/schema/beans"
    xmlns:context = "http://www.springframework.org/schema/context"
    xmlns:mvc = "http://www.springframework.org/schema/mvc"
    xsi:schemaLocation = "
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        ">
```

В коде этого листинга через `xmlns` указывается пространство имен для префиксов. так, для префикса `mvc` подключено пространство имен `http://www.springframework.org/schema/mvc`, а для префикса `context` — пространство имен `http://www.springframework.org/schema/context`.

Полезных строк в файле context.xml в нашем случае не так много (листинг 4.2).

Листинг 4.2. Продолжение файла context.xml

```
<!-- Поддержка Spring MVC @Controller -->
<mvc:annotation-driven />

<context:component-scan
    base-package =
    "ru.urvanov.springbook2024.simplespringwebmvc.controller" />
```

Здесь элемент `mvc:annotation-driven` позволяет использовать аннотации `@Controller` и дочерние для описания контроллеров Spring MVC.

В `component-scan` указан пакет, в котором будет осуществляться поиск бинов Spring. Такой поиск будет производиться также и в дочерних пакетах — т. е. согласно приведенному примеру в поиск попадут и пакеты `ru.urvanov.springbook2024.simplespringwebmvc.controller.domain`, и любые другие подпакеты, если они будут найдены.

Единственный контроллер, который у нас есть, — это `MainController` (листинг 4.3).

Листинг 4.3. MainController.java

```
package ru.urvanov.springbook2024.simplespringwebmvc.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class MainController {

    @GetMapping("/")
    public String home() {
        return "Hello, world!";
    }
}
```

Аннотация `@RestController` помечает здесь класс как контроллер Spring MVC. На самом деле она заменяет две аннотации:

- `@Controller` — помечает класс как контроллер Spring MVC;
- `@ResponseBody` — указывает, что методы контроллера возвращают тело ответа

4.2.3. Запуск

В контроллере `MainController` есть только один метод, и он помечен аннотацией `@GetMapping`, которая сопоставляет запросы GET с методом, на котором она проставлена. В `MainController` метод `home` будет обрабатывать HTTP-запросы GET на адрес **http://localhost:8080/simple-spring-webmvc/**, где:

- **localhost** — адрес сервера (при запуске локально — localhost),
- **8080** — порт сервера (по умолчанию для Apache Tomcat — 8080);
- **simple-spring-webmvc** — Apache Tomcat одновременно может хостить несколько приложений, разделяемых по разным интернет-адресам (URL). В нашем приложении по умолчанию это будет либо **simple-spring-webmvc** — при запуске из IDE, либо название war-файла.

Запустите приложение в вашей IDE так же, как это было описано в *разд. 3.1.3*, — по адресу **http://localhost:8080/simple-spring-webmvc** браузер должен отобразить текст **Hello, world!**

4.3. Простой сервис на Spring Boot

4.3.1. Скачайте исходные коды

Исходные коды приложения доступны на GitHub[1], но вы также можете скачать ZIP-архив из раздела моего сайта, посвященного этой книге[2]:

4.3.2. Spring Initializr

Сервис на Spring Boot создается гораздо проще, чем на Spring Framework. Вы можете сами воспользоваться онлайн-утилитой Spring Initializr[3] и создать его уже сейчас.

> ***Полезная инструкция***
>
> На сайте **https://urvanov.ru** можно найти инструкцию по использованию утилиты Spring Initializr для Eclipse, NetBeans и IntelliJ IDEA с цветными скриншотами.

Просто заполните все настройки так, как показано на рис. 4.1, а подробное описание настроек приведено в последующем тексте.

- Переключатель **Project** позволяет выбирать между менеджерами пакетов **Gradle – Groovy**, **Gradle – Kotlin** и **Maven**. Вы можете выбрать любой из них, но в этой книге все примеры сделаны именно с использованием Maven. Однако если вы будете разрабатывать приложения на Java, то вам, скорее всего, в той или иной мере придется изучить все эти три способа.

---

[1] См. **https://github.com/urvanov-ru/simple-spring-boot-webmvc**.

[2] См. **https://urvanov.ru/книги/spring-book-2024/**.

[3] См. **https://start.spring.io**.

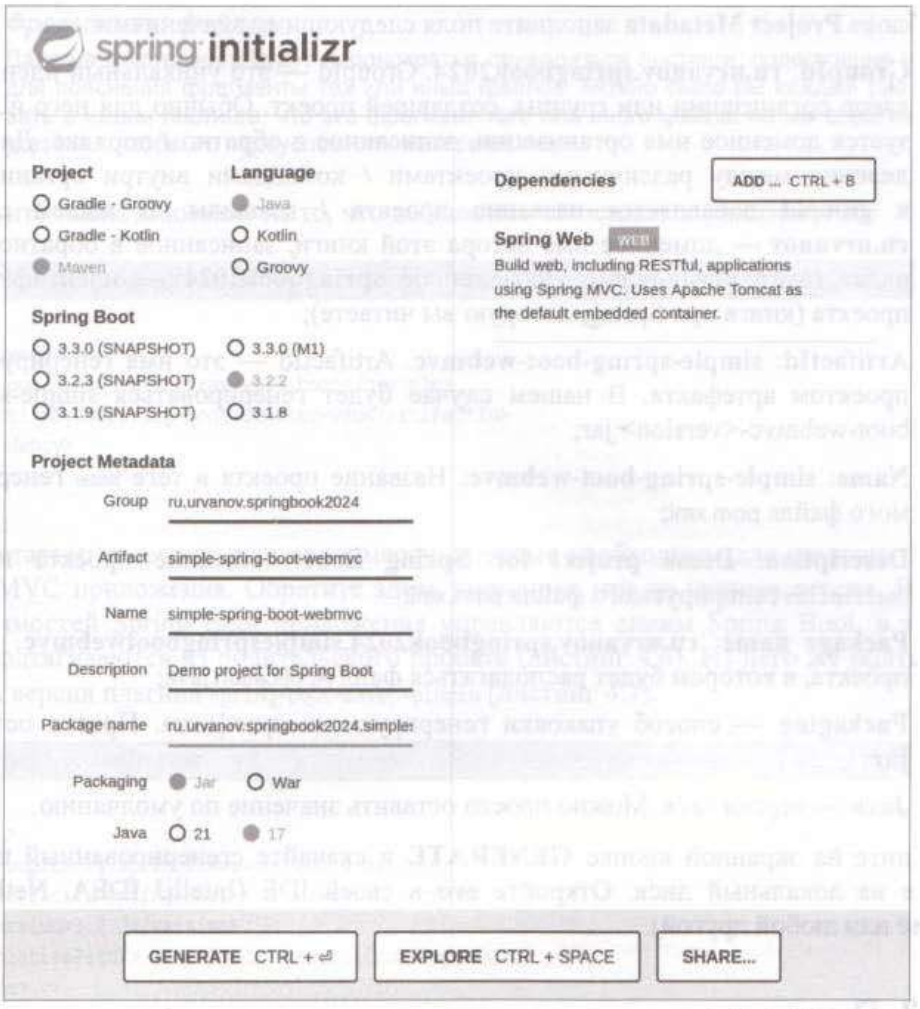


Рис. 4.1. Создание нового проекта с помощью онлайн-утилиты Spring Initializr

- Переключатель **Language** позволяет выбирать между тремя языками программирования: **Java**, **Kotlin** и **Groovy**. Просто оставьте переключатель в положении Java, как он и стоит по умолчанию.
- Переключатель версии **Spring Boot** по умолчанию ставится в самую последнюю доступную версию — здесь ничего менять не стоит.
- Особое внимание стоит уделить блоку **Dependencies**. В нем выбираются зависимости, которые будут подключены к генерируемому проекту. Сначала список зависимостей пуст. Пролистайте список зависимостей, доступных для добавления по кнопке **ADD...** Список довольно большой, и в нем можно найти зависимости для разработки слоя постоянства, для написания тестов, для различных интеграций и т. д. Для самого простого примера приложения Spring Web MVC на Spring Boot необходимо подключить только одну зависимость — **Spring Web**, чтобы добавить необходимый стартер в генерируемое приложение.

- В блоке **Project Metadata** заполните поля следующими значениями:
  - **GroupId**: **ru.urvanov.springbook2024**. GroupId — это уникальный идентификатор организации или группы, создавшей проект. Обычно для него используется доменное имя организации, записанное в обратном порядке. Для разделения между различными проектами / командами внутри организации к groupId добавляется название проекта / команды. В нашем случае **ru.urvanov** — доменное имя автора этой книги, записанное в обратном порядке (сайт **urvanov.ru**). Добавленное **springbook2024** — идентификатор проекта (книга про Spring, которую вы читаете);
  - **ArtifactId**: **simple-spring-boot-webmvc**. ArtifactId — это имя генерируемого проектом артефакта. В нашем случае будет генерироваться simple-spring-boot-webmvc-<version>.jar;
  - **Name**: **simple-spring-boot-webmvc**. Название проекта в теге `Name` генерируемого файла pom.xml;
  - **Description**: **Demo project for Spring Boot**. Описание проекта в теге `Description` генерируемого файла pom.xml;
  - **Package name**: **ru.urvanov.springbook2024.simplespringbootwebmvc**. Пакет проекта, в котором будет располагаться файл Application.java;
  - **Packaging** — способ упаковки генерируемого артефакта. Просто оставьте **jar**;
  - **Java** — версия Java. Можно просто оставить значение по умолчанию.

Щелкните на экранной кнопке **GENERATE** и скачайте сгенерированный проект к себе на локальный диск. Откройте его в своей IDE (IntelliJ IDEA, NetBeans, Eclipse или любой другой).

4.3.3. Пояснения к исходному коду

Обратите внимание на содержимое файла pom.xml — в нем указаны свойства из `Project Metadata` (листинг 4.4).

Листинг 4.4. Фрагмент файла pom.xml

```
...
<groupId>ru.urvanov.springbook2024</groupId>
<artifactId>simple-spring-boot-webmvc</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>simple-spring-boot-webmvc</name>
<description>Demo project for Spring Boot</description>
<properties>
    <java.version>17</java.version>
</properties>
...
```

Фрагменты файлов

Далее в тексте книги будут неоднократно приводиться листинги, содержащие нужные для пояснения фрагменты тех или иных файлов. Можно было бы каждый раз указывать в имени листинга, что это фрагмент того или иного файла, но мы опустим такие указания, чтобы не переусложнять материал книги.

Обратите также внимание на то, что добавилась зависимость (листинг 4.5).

Листинг 4.5. Файл pom.xml

```
...
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
...
```

Этот стартер содержит все зависимости, которые необходимы для создания Spring Web MVC приложения. Обратите здесь внимание, что не указана версия. Версии зависимостей Spring Boot приложения управляются самим Spring Boot, а точнее они подтягиваются из родительского проекта (листинг 4.6). Из него же подтягивается и версия плагина `spring-boot-maven-plugin` (листинг 4.7).

Листинг 4.6. Файл pom.xml

```
...
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.2.2</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
...
```

Листинг 4.7. Файл pom.xml

```
...
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
...
```

Из Java-кода утилитой **start.spring.io** создался всего один класс (листинг 4.8).

Листинг 4.8. SimpleSpringBootWebmvcApplication.java

```
package ru.urvanov.springbook2024.simplespringbootwebmvc;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SimpleSpringBootWebmvcApplication {

    public static void main(String[] args) {
        SpringApplication.run(
                SimpleSpringBootWebmvcApplication.class, args);
    }
}
```

Как видно из этого класса, приложение на Spring Boot — это обычное Java-приложение, в методе `main` которого инициализируется Spring Boot, который внутри себя и делает всю «магию» по настройке Embedded Tomcat.

Вы можете уже сейчас запустить проект так же, как запускали самые простые приложения на Java через **Run As | Java Application** в своей IDE. Оно запустится, но пока ничего не отобразит.

Добавьте самый простой контроллер, который будет обрабатывать запрос к **http://localhost:8080/** и возвращать строку **Hello, world!** (см. листинг 4.3).

4.3.4. Запуск

Контроллер для запуска приложения на Spring Boot абсолютно идентичен контроллеру из *разд. 4.2*. Обратите внимание, что теперь адрес, возвращающий **Hello, world!**, не содержит названия приложения, как это было для приложения на Spring Framework без Spring Boot, — корневая страница доступна сразу по адресу **http://localhost:8080**:

Hello, world!

Вы также можете собрать jar-файл и запустить его — использующийся в файле pom.xml плагин Spring Boot Maven Plugin сделает его запускаемым:

```
$ mvnw clean package
$ java -jar target/simple-spring-boot-webmvc-0.0.1-SNAPSHOT.jar
```

4.4. Различия между Spring Boot и Spring Framework

В предыдущих разделах этой главы мы рассмотрели простейшие примеры веб-приложения на Spring Framework и на Spring Boot, кроме того, в *главе 3* был показан запуск более сложного приложения в обоих вариантах. Сразу же должны возникнуть вопросы: если оба варианта приложения работают и в принципе делают

одно и то же, зачем нужен Spring Boot? Чем различаются Spring Boot и Spring Framework? Что такое Spring Framework и что такое Spring Boot?

Дело в том, что Spring Boot не является самостоятельным средством разработки — это, скорее, надстройка над Spring Framework, позволяющая создавать приложения, которые можно запускать без развертывания в сервисе приложения Apache Tomcat, GlassFish или аналогичном. Проще говоря, создаваемые с помощью Spring Boot jar-файлы приложений вы сможете запускать самостоятельно как обычное приложение Java, потому что он сам запустит и настроит встраиваемую версию сервиса приложений (по умолчанию Apache Tomcat), настроит его и развернет в нем разрабатываемое приложение.

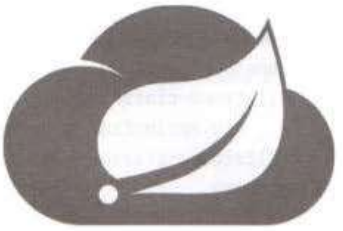
Spring Boot также облегчает управление зависимостями. Ведя разработку на Spring Framework без Spring Boot, программисту самому приходится следить за совместимостью и версиями достаточно большого количества библиотек и фреймворков, используемых приложением. А при разработке на основе Spring Boot приложение задает в качестве родительского артефакта `spring-boot-starter-parent` либо указывает в блоке `dependencyManagement` артефакт `spring-boot-dependencies`, который уже содержит названия версий допустимых зависимостей для большинства из возможных библиотек и фреймворков, поэтому вам не придется задавать версии каждой библиотеки вручную.

Кроме того, Spring Boot с помощью стартеров наподобие `spring-boot-starter-web`, который упоминался в *разд. 4.3.3*, а также автоконфигураций упрощает добавление в разрабатываемое приложение новых возможностей, для которых в Spring Framework без Spring Boot приходилось настраивать поведение и добавлять несколько зависимостей вместо одного стартера.

4.5. Резюме

Spring позволяет создавать консольные, десктопные и веб-приложения. Spring Boot облегчает настройку и первичную инициализацию приложения, использующего Spring Framework. Spring Initializr позволяет создать пустое приложение Spring Boot со всеми необходимыми зависимостями.

ГЛАВА 5



Модуль Spring Core

5.1. XML-конфигурация (для Spring Framework)

5.1.1. Листенер *ContextLoadListener*

> ***Java-конфигурация не обязательно требует Spring Boot***
>
> Spring Framework и Spring Boot позволяют использовать как XML, так и Java-конфигурацию, но для Spring Boot и новых приложений принято использовать именно вариант с Java-конфигурацией.

XML-конфигурацию удобнее изучать на примере из *разд. 3.1*. С чего начинается XML-конфигурация? Откуда она загружается? Рассмотрим содержимое файла src/main/webapp/WEB-INF/web.xml проекта virtualpets-server-springframework (листинг 5.1).

Листинг 5.1. Файл src/main/webapp/WEB-INF/web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring/root-context.xml
        /WEB-INF/spring/security.xml
        /WEB-INF/spring/appServlet/servlet-context.xml
    </param-value>
</context-param>
```

Файлы конфигураций, как можно здесь увидеть, указываются в параметре `contextConfigLocation`. Допускается запись нескольких файлов, при этом они разделяются любым количеством запятых, пробелов, табуляторов или переводов строк. В листинге 5.1 указаны три файла: `root-context.xml`, `security.xml`, `appServlet/servlet-context.xml`, которые находятся в каталоге src/main/webapp/WEB-INF/spring. Этот параметр считывается листенером `ContextLoadListener`, объявленным в том же файле (листинг 5.2).

Листинг 5.2. Файл src/main/webapp/WEB-INF/web.xml

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

5.1.2. Разделение по файлам и контекстам

Каждый файл конфигурации описывает бины определенной функциональности или слоя:

- root-context.xml — корневой контекст, в нем описаны все бины слоя сервисов, слоя постоянства, а также все остальные бины, необходимые для работы сервера;
- security.xml — бины, связанные со Spring Security, безопасностью, аутентификацией и авторизацией;
- servlet-context.xml — настройка Spring Web MVC и бинов, связанных с ним.

> ***Иметь несколько файлов конфигурации необязательно***
>
> Нет строгого требования по созданию нескольких файлов конфигурации и какому-либо их разделению. Вполне допускается использование только одного XML-файла, в котором настраиваются как основные бины приложения, так и Spring Security, Spring Web MVC и всё остальное.

В большинстве случаев, как и в примере из этой книги с сервером виртуальных питомцев, вполне достаточно одного контекста, поэтому все бины объявляются в одном контексте (пусть даже и в разных файлах, что также не обязательно), тогда в `DispatcherServlet` в качестве `contextConfigLocation` не передается ничего (листинг 5.3).

Листинг 5.3. Файл pom.xml

```
<servlet>
    <servlet-name>appServlet</servlet-name>
    <servlet-class>
        org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value></param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

В более сложных случаях есть возможность объявить корневой контекст, а в дополнение к нему объявить дополнительно дочерний контекст. Дочерний контекст объявляется в параметре `contextConfigLocation` сервлета `DispatcherServlet`. Сервлетов `DispatcherServlet` либо других сервлетов может быть несколько. Бины из корневого

контекста, объявленного в `context-param`, доступны в дочерних контекстах, объявленных в контекстах `DispatcherServlet`, они также могут быть переопределены в контекстах `DispatcherServlet`.

5.1.3. Пространства имен

Обычно в корневом контексте объявляются бины слоя постоянства и бины бизнес-логики, а бины, связанные с Spring Web MVC, объявляются в контексте `DispatcherServlet`.

В файле root-context.xml объявлены основные бины слоя доступа к данным и слоя бизнес-логики приложения. Начинается он с объявления префиксов для пространств имен (листинг 5.4).

Листинг 5.4. Файл root-context.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:context = "http://www.springframework.org/schema/context"
    xmlns:task = "http://www.springframework.org/schema/task"
    xmlns:aop = "http://www.springframework.org/schema/aop"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/task
    http://www.springframework.org/schema/task/spring-task.xsd">
```

Если вы уже знакомы с XML-схемами, то приведенный здесь код вам должен быть абсолютно понятен. В нем просто объявляются префиксы для пространств имен:

- пространство имен по умолчанию `http://www.springframework.org/schema/beans` содержит элементы для описания бинов;
- префикс `context` используется для пространства имен `http://www.springframework.org/schema/context`, которое обеспечивает поддержку конфигурирования ApplicationContext;
- префикс `task` ассоциирован с пространством имен `http://www.springframework.org/schema/task`, содержащим элементы для настройки запуска заданий по расписанию;
- префикс `aop` ассоциирован с пространством имен `http://www.springframework.org/schema/aop`, содержащим элементы для создания аспектов и аспектно-ориентированного программирования.

В следующих разделах в первую очередь мы рассмотрим пространство имен `http://www.springframework.org/schema/beans`, а также пространство имен `http://www.`

springframework.org/schema/context — как основные пространства для конфигурирования контекста Spring.

5.1.4. Объявление бинов

Обратите внимание на объявления бина в листинге 5.5.

Листинг 5.5. Файл root-context.xml

```
<bean id = "version" class = "java.lang.String">
    <constructor-arg value = "0.21" />
</bean>
```

Это объявление создаст бин с именем `version` и типом `String`. Подобное конфигурирование бинов рассмотрено в *разд. 4.1*.

Единственный элемент в этом описании, с которым мы еще не знакомы, это элемент `constructor-arg`. Элементы `constructor-arg` описывают внедрение зависимостей с помощью конструктора. В приведенном примере будет использован конструктор `new String("0.21")`, т. к. в атрибуте `value` указано строковое значение `0.21`.

Аналогично можно внедрить в бин зависимость от другого бина. Например, если мы создаем какой-нибудь бин, в который необходимо внедрить через конструктор зависимость от уже созданного бина `version`, то необходимо использовать элемент `constructor-arg` с атрибутом `ref`. Пример такого внедрения можно увидеть в файле security.xml (листинг 5.6).

Листинг 5.6. Файл security.xml

```
<bean id = "securityContextRepository"
        class =
           "org.springframework.security.web.context.DelegatingSecurityContextRepository">
           <constructor-arg ref = "securityContextRepositoryList" />
</bean>
```

Здесь создается бин класса `DelegatingSecurityContextRepository`, которому с помощью внедрения зависимостей в конструктор передается бин с именем `securityContextRepositoryList`.

В файле root-context.xml есть и другой способ внедрения зависимостей — через внедрение свойств (листинг 5.7).

Листинг 5.7. Файл root-context.xml

```
<bean id = "templateMessage"
        class = "org.springframework.mail.SimpleMailMessage">
    <property name = "from"
    value = "${virtualpets-server-springframework.mail.from}" />
    <property name = "subject" value = "Recover password" />
</bean>
```

```
        <context:property-placeholder
            location = "classpath:mail_prod.properties"
            file-encoding = "utf-8"
            ignore-unresolvable = "true" />
    </beans>
</beans>
```

Обратите внимание на разделы `beans profile` — с их помощью происходит разделение на профили:

- вся конфигурация внутри тега `<beans profile = "test">` будет применена только при активном профиле `test`;
- вся конфигурация внутри тега `<beans profile = "development">` будет применена только при активном профиле `development`;
- вся конфигурация внутри тега `<beans profile = "production">` будет применена только при активном профиле `production`.

Внутри тега `<beans profile=...>` может быть прописана любая конфигурация: бины, загрузка «пропертей» и т. д.

В примере сервера виртуальных питомцев virtualpets-server-springframework по профилям разделяется загрузка файлов «пропертей»:

- при активном профиле `test` «проперти» загружаются из файлов application_test.properties и mail_test.properties;
- при активном профиле `development` «проперти» загружаются из файлов application_dev.properties и mail_dev.properties;
- при активном профиле `production` «проперти» загружаются из файлов application_prod.properties и mail_prod.properties.

Активный профиль задается в контекстном параметре `spring.profiles.active` файла src/main/webapp/WEB-INF/web.xml (листинг 5.9).

Листинг 5.9. Файл src/main/webapp/WEB-INF/web.xml

```
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>development</param-value>
</context-param>
```

Контекстный параметр `spring.profiles.active` можно переопределить в файле context.xml экземпляра Apache Tomcat. Подробнее о переопределении контекстных параметров и о файлах «пропертей» рассказано в *разд. 8.1* и *8.2*.

Файл mail_dev.properties, из которого загружаются «проперти» при профиле `development`, выставленном по умолчанию, представляет собой обычный файл Java properties (листинг 5.10).

Листинг 5.10. Файл mail_dev.properties

```
virtualpets-server-springframework.mail.server=localhost
virtualpets-server-springframework.mail.port=8888
virtualpets-server-springframework.mail.username=dddd
virtualpets-server-springframework.mail.password=ddddd
```

Через внедрение полей добавляются не только значения, но и ссылки на другие бины — аналогично внедрению с помощью конструктора. Например, в файле security.xml есть подобный пример внедрения ссылки на бин `userService` через атрибут `ref` элемента `property` (листинг 5.11).

Листинг 5.11. Файл security.xml

```
<bean id = "authenticationProvider"
    class =
   "org.springframework.security.authentication.dao.DaoAuthenticationProvider">
      <property name = "userDetailsService" ref = "userDetailsService" />
      <property name = "passwordEncoder" ref = "passwordEncoder" />
</bean>
```

Здесь в поле `userDetailsService` бина `authenticationProvider` внедряется ссылка на бин `userDetailsService`, а в поле `passwordEncoder` внедряется ссылка на бин `passwordEncoder` соответственно.

5.1.6. Сканирование бинов

Описывать каждый бин в XML-файле конфигурации может быть довольно затруднительно. Для слоя сервисов с бизнес-логикой и слоя постоянства зачастую используют *сканирование компонентов* — как это сделано в файле root-context.xml (листинг 5.12).

Листинг 5.12. Файл root-context.xml

```
<context:component-scan
       base-package = "ru.urvanov.virtualpets.server.dao,
                    ru.urvanov.virtualpets.server.service
       " />
```

XML-элемент `context:component-scan` ищет бины в пакетах, указанных в `base-package`. Поиск бинов осуществляется по аннотации `@Component` (а также ее специализациям `@Repository`, `@Service`, `@Controller` и `@RestController`). В приведенном примере Spring будет искать бины, т. е. классы, помеченные соответствующими аннотациями, среди классов, находящихся в пакетах ru.urvanov.virtualpets.server.dao и ru.urvanov.virtualpets.server.service.

Например, бином Spring станет следующий класс (листинг 5.13).

Листинг 5.13. PetServiceImpl.java

```
package ru.urvanov.virtualpets.server.service;

@Service
public class PublicServiceImpl implements PublicApiService {
```

Аннотации имеют разное значение:

- `@Component` — обычный бин Spring;
- `@Repository` — ставится на слое постоянства, выполняет преобразование исключений различных API доступа к данным в одно из исключений `DataAccessException`, по которому можно понять причину ошибки вне зависимости от конкретного используемого API. Аналогично утилиты могут использовать эту аннотацию для определения классов из слоя постоянства;
- `@Service` — этой аннотацией помечаются бины из слоя бизнес-логики;
- `@Controller` — ставится на контроллерах Spring MVC, которые с помощью аннотации `@RequestMapping` определяют обрабатываемые их методами URL.

Бины могут использовать аннотацию `@Autowired` для внедрения своих зависимостей, которую обычно ставят на объявлении поля класса (листинг 5.14).

Листинг 5.14. PublicServiceImpl.java

```
@Autowired
private UserDao userDao;

@Autowired
private MailSender mailSender;

@Autowired
private SimpleMailMessage templateMessage;

@Autowired
private String version;
```

При инициализации бина в поле `userDao` будет внедрен бин из контейнера бинов Spring с типом `UserDao` или с дочерним от типа `UserDao` типом. В поля `mailSender`, `templateMessage` и `version` аналогично будут внедрены бины с типами `MailSender`, `SimpleMailMessage`, `String` или дочерними. Если в контейнере бинов не найдется бина с подходящим типом либо их окажется больше одного, то будет брошено исключение во время выполнения.

> ***`@Autowired` — не только для полей***
>
> Аннотацию `@Autowired` допускается использовать не только на объявлениях полей, но и на методах установки значений, а также конструкторах

В листинге 5.15 приведен пример `@Autowired` для метода установки значения (сеттера).

Листинг 5.15. JdbcReportDaoImpl.java

```
@Autowired
public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
}
```

С помощью аннотаций можно внедрять не только ссылки на бины, но и значения из property-файлов (листинг 5.16).

Листинг 5.16. PublicServiceImpl.java

```
@Value("${virtualpets-server-springframework.server.url}")
private String serverUrl;
```

Здесь значения считываются из файлов application_test.properties, application_dev.properties или application_prod.properties — в зависимости от профиля. Конфигурация загрузки property-файлов находится в файле properties.xml. В следующем примере (листинг 5.17) в поле `serverUrl` будет внедрено значение «проперти» с ключом `virtualpets-server-springframework.server.url`.

Листинг 5.17. Файл application_dev.properties

```
virtualpets-server-springframework.server.url =
                                    http://localhost:8080/virtualpets-server-springframework
virtualpets-server-springframework.play.url=http://localhost:8081/
```

5.1.7. Импортирование файлов конфигураций

Существует возможность разделять слишком большие файлы конфигураций Spring на несколько файлов, импортируя другие файлы в основной. Например, в сервере виртуальных питомцев файл root-context импортирует файлы servlet-jee.xml и servlet-tx.xml (листинг 5.18).

Листинг 5.18. Файл root-context.xml

```
<import resource = "properties.xml" />
<import resource = "servlet-jee.xml" />
<import resource = "servlet-tx.xml" />
```

Здесь файл properties.xml содержит загрузку property-файлов:

- файл servlet-jee.xml содержит конфигурацию JNDI-ресурсов. Название **-jee говорит о том, что в файле планируется хранить и другие бины, которые связаны с ресурсами, поставляемыми сервисом приложений Apache Tomcat, но в примере сервера виртуальных питомцев никакие другие ресурсы, кроме подключения к базе данных, пока не используются;
- файл servlet-tx.xml содержит описания бинов, связанных с управлением транзакциями, Hibernate, а также настраивает библиотеку Liquibase для управления миграциями данных.

Разделять файлы не обязательно — это просто пример

Нет никакого обязательного требования использовать подобное разделение на файлы servlet-jee.xml, servlet-tx.xml или какое-либо другое разделение. В вашем приложении может быть один конфигурационный файл, который настраивает всё необходимое.

5.1.8. Коллекции

С помощью XML-файлов конфигураций можно внедрять не только фиксированные значения, значения из файлов-property и ссылки на другие бины — Spring позволяет описывать коллекции и внедрять их в конструкторы и поля бинов (листинг 5.19).

Листинг 5.19. Файл root-context.xml

```
<bean id = "conversionService"
    class = "org.springframework.format.support.FormattingConversionServiceFactoryBean">
    <property name = "converters">
        <set>
            <bean
                class = "ru.urvanov.virtualpets.server.convserv.BookToApiConverter" />
            <bean
                class = "ru.urvanov.virtualpets.server.convserv.ClothToApiConverter" />
            <bean
                class = "ru.urvanov.virtualpets.server.convserv.DrinkToApiConverter" />
            <bean
                class = "ru.urvanov.virtualpets.server.convserv.FoodToApiConverter" />
            <bean
                class = "ru.urvanov.virtualpets.server.convserv.PetToApiConverter" />
        </set>
    </property>
</bean>
```

Здесь в поле `converters` бина `conversionService` внедряется реализация интерфейса java.util.Set с бинами экземпляров классов `BookToApiConverter`, `ClothToApiConverter`, `DrinkToApiConverter` и пр. (полный листинг находится в файле root-context.xml проекта virtualpets-server-springframework, рассматриваемого в книге в качестве примера).

Аналогичным образом можно внедрять реализации java.util.List, используя элемент `list`, и реализации java.util.Map, применяяя элемент `map`. Синтаксис для `List` выглядит аналогично `Set`, но для `map` в качестве элементов задействуются `entry`.

В сервере виртуальных питомцев не нашлось места для полноценных примеров внедрения java.util.Map и java.util.List (хотя в файле security.xml вы можете найти вариант создания бина, реализующего java.util.List, который выглядит почти так же, но представляет собой самостоятельный бин в контейнере Spring Framework), поэтому эти примеры мы рассмотрим на искусственно созданном для этих целей классе `InjectExample.java` (листинг 5.20).

Листинг 5.20. InjectExample.java

```
package ru.urvanov.virtualpets.server.example;
...
public class InjectExample {
...
    private List<String> names;
...
    public List<String> getNames() {
        return names;
    }

    public void setNames(List<String> names) {
        this.names = names;
    }

...
}
```

В классе `InjectExample` **поле** `names` **имеет тип java.util.List, в который внедряется реализация этого интерфейса в XML-конфигурации с помощью элемента** `list` **(листинг 5.21).**

Листинг 5.21. Файл root-context.xml

```
<bean class = "ru.urvanov.virtualpets.server.example.InjectExample">
    <property name = "names">
        <list>
            <value>Вася</value>
            <value>Шурик</value>
            <value>Оксана</value>
            <value>Семён</value>
            <ref bean = "version" />
            <bean class = "java.lang.String">
                <constructor-arg value = "Святослав-Бинов" />
            </bean>
        </list>
    </property>
...
</bean>
```

В этом примере в поле `names` **внедряется реализация java.util.List, заполненная значениями** `Вася`, `Шурик`, `Оксана`, `Семён`, **с помощью элемента** `ref` **в список добавляется бин** `version`, **а элемент** `Святослав-Бинов` **определяется с помощью** `bean`, **что делает его полноценным бином.**

Класс `InjectExample` **имеет также поле** `numberSumMap` **с типом java.util.Map (листинг 5.22).**

Листинг 5.22. InjectExample.java

```
package ru.urvanov.virtualpets.server.example;
...
public class InjectExample {
...
    private Map<String, BigDecimal> numberSumMap;
...

    public Map<String, BigDecimal> getNumberSumMap() {
        return numberSumMap;
    }

    public void setNumberSumMap(Map<String, BigDecimal> numberSumMap) {
        this.numberSumMap = numberSumMap;
    }
...
}
```

Для внедрения значения в `numberSumMap` в XML-конфигурации используется `map` (листинг 5.23).

Листинг 5.23. InjectExample.java

```
<bean class="ru.urvanov.virtualpets.server.example.InjectExample">
...
    <property name = "numberSumMap">
        <map>
            <entry key = "D-001" value = "100.00" />
            <entry key = "D-002" value = "55.01" />
            <entry key = "D-003" value = "34.23" />
            <entry>
                <key>
                    <ref bean = "version"/>
                </key>
                <value>1000.00</value>
            </entry>
            <entry key = "D-005">
                <bean class = "java.math.BigDecimal">
                    <constructor-arg value = "100000.00" />
                </bean>
            </entry>
        </map>
    </property>
...
</bean>
```

Приведенный пример внедряет в `numberSumMap` реализацию java.util.Map, содержащую:

- `D-001` ⟶ `BigDecimal` со значением `100,00`;
- `D-002` ⟶ `BigDecimal` со значением `55,01`;

- D-003 → BigDecimal со значением 34,23;
- в качестве ключа бин version → BigDecimal со значением 1000,00;
- D-005 → бин с типом BigDecimal и значением 100 000,00.

Spring также позволяет создавать бины, реализующие java.util.List, java.util.Set, java.util.Map, заполненные данными, для чего используется пространство имен http://www.springframework.org/schema/util, которое обычно связывается с префиксом util, как это сделано в файле security.xml (листинг 5.24).

Листинг 5.24. Файл security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
...
    xmlns:util="http://www.springframework.org/schema/util"
...
    xsi:schemaLocation="http://www.springframework.org/schema/beans
...
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util.xsd
...
     ">
...
    <util:list id = "securityContextRepositoryList">
        <bean class = "org.springframework.security.web.context.
                                        RequestAttributeSecurityContextRepository"></bean>
        <bean class = "org.springframework.security.web.context.
                                            HttpSessionSecurityContextRepository"></bean>
    </util:list>
```

Синтаксис создания бинов, реализующих java.util.Set и java.util.Map, с помощью префикса util аналогичен.

5.2. Java-конфигурация (для Spring Boot)

5.2.1. Аннотация @SpringBootApplication

В современном мире в основном используется не XML-конфигурация, а Java-конфигурация Spring совместно со Spring Boot.

Приложение на Spring Boot начинается с класса Application (название класса не обязательно должно быть именно таким, но чаще всего оно либо такое, либо *НазваниеПриложения*Application), помеченного аннотацией @SpringBootApplication. В примере виртуальных питомцев virtualpets-server-springboot основной класс Application выглядит так (листинг 5.25).

Листинг 5.25. Application.java

```
package ru.urvanov.virtualpets.server;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Аннотация `@SpringBootApplication`:

- включает механизм автоконфигурации;
- включает механизм сканирования бинов для текущего пакета и дочерних пакетов;
- использует помеченный класс как источник дополнительной конфигурации, помимо классов, помеченных аннотацией `@Configuration`, которую мы рассмотрим в *разд. 5.2.2*.

Механизм автоконфигурации автоматически настраивает разные возможности и бины Spring на основе зависимостей, найденных в `classpath`. Например, если будет найдена зависимость от HSQLDB, то Spring автоматически создаст бины с подключением к базе данных, использующей ОЗУ для хранения.

Механизм сканирования бинов ищет классы, помеченные аннотацией `@Component` и ее специализациями наподобие `@Service`, `@Repository` и другими, а также создает для них бины в инициализируемом контексте — аналогично тому, как это происходит при использовании `context:component-scan` при XML-конфигурации, описанной в *разд. 5.1.6*.

5.2.2. Аннотации @Configuration и @Bean

Если различным дополнительным проектам Spring, наподобие Spring Web MVC или Spring Security, нужна дополнительная конфигурация либо необходимо сконфигурировать дополнительные бины для использования, то обычно у пакета, в котором находится класс `Application`, создается подпакет `config`, а в нем размещаются классы с конфигурациями, помеченные аннотациями `@Configuration` (листинг 5.26).

Листинг 5.26. ClockConfig.java

```
package ru.urvanov.virtualpets.server.config;

import java.time.Clock;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
/**
 * конфигурация {@link Clock}
 */
@Configuration
public class ClockConfig {

    /**
     * Классы пакета {@link java.time} используют этот экземпляр Clock
     * при создании экземпляров с текущей датой и временем. Подобная
     * практика позволяет подставить в тесты другой Clock,
     * возвращающий всегда фиксированное значение.
     * @return Экземпляр {@link Clock}
     */
    @Bean
    public Clock clock() {
        return Clock.systemDefaultZone();
    }
}
```

Аннотацией `@Bean` помечаются методы, возвращающие экземпляры классов, из которых необходимо создать бины. В качестве аргументов методов, помеченных этой аннотацией, принимаются другие бины, а также значения из property-файлов с помощью аннотации `@Value`.

5.2.3. Профили

Любой класс, помеченный явно или неявно аннотацией `@Component` (в том числе и `@Configuration`), может быть дополнительно помечен аннотацией `@Profile` с указанием профилей, для которых бин необходимо создать. Вот, например, как может выглядеть класс `ClockConfig` для тестов (листинг 5.27).

Листинг 5.27. ClockConfig.java

```
package ru.urvanov.virtualpets.server.config;

import java.time.Clock;
import java.time.Instant;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.temporal.TemporalAccessor;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Profile;

@Profile({"test-dao", "test-mock-mvc"})
public class ClockConfig {
    /**
     * Настраивает экземпляр Clock, возвращающий всегда
     * одну и ту же дату и одно и то же время для предсказуемости тестов.
     * @return Экземпляр Clock с фиксированной датой и временем.
     */
```

```
    @Bean
    public Clock clock() {
        ZoneId zoneId = ZoneId.of("Europe/Moscow");
        TemporalAccessor offsetDateTime = ZonedDateTime.of(
                2024, 3, 15, 18, 52, 0, 0, zoneId);
        Instant instant = Instant.from(offsetDateTime );
        return Clock.fixed(instant, zoneId);
    }
}
```

Аналогично аннотация `@Profile` может быть на методе `@Bean` — например, как показано в листинге 5.28.

Листинг 5.28. MailConfig.java

```
@Profile({"test-dao", "test-mock-mvc"})
@Bean
public Clock clock() {
    ZoneId zoneId = ZoneId.of("Europe/Moscow");
    TemporalAccessor offsetDateTime = ZonedDateTime.of(
            2024, 3, 15, 18, 52, 0, 0, zoneId);
    Instant instant = Instant.from(offsetDateTime );
    return Clock.fixed(instant, zoneId);
}
```

Внутри аннотации `@Profile` допускаются не только имена профилей — в ней разрешаются логические операции:

- логическое НЕ — например: `@Profile("!production");`
- логическое И — например: `@Profile("development & feature1");`
- логическое ИЛИ — пример: `@Profile("development | test").`

Не забудьте только при комбинировании И и ИЛИ использовать скобки.

В приложении сервера виртуальных питомцев есть также классы: `WebConfig` — для настройки Spring Web MVC и `SecurityConfig` — для настройки Spring Security. Содержимое этих классов конфигураций будет рассмотрено в соответствующих главах книги.

Spring Boot считывает настройки из файлов application.yaml или application.properties, которые обычно находятся в каталоге src/main/resources. Большая часть конфигурации производится путем внесения настроек в эти файлы.

В проекте virtualpets-server-springboot имеется файл src/main/resources/application.yaml, в котором содержатся настройки подключения к базе данных, JPA, Liquibase и настройки самого проекта virtualpets-server-springboot.

В листинге 5.29 приведено начало файла application.yaml — чтобы сформировать у вас представление о том, что он собой представляет. Значения каждой из настроек будут рассмотрены далее в соответствующих разделах.

Листинг 5.29. Файл application.yaml

```
spring:
  application:
    name: virtualpets-server-springboot
  config:
    import: optional:configserver:http://localhost:8888
  datasource:
    url: jdbc:postgresql://localhost:5432/postgres
    driver-class-name: org.postgresql.Driver
    username: postgres
    password: postgres
    hikari:
      schema: virtualpets_server_springboot
  messages:
    fallback-to-system-locale: false
  jpa:
    properties:
      hibernate:
        globally_quoted_identifiers: 'true'
```

Описание YAML

Формат YAML имеет множество возможностей для представления словарей и списков. Однострочный вариант представления фактически делает JSON подмножеством YAML. Подробное описание формата вы найдете по адресу: **https://yaml.org/**. В рамках примера virtualpets-server-springboot используется лишь простейшее блочное представление.

Например, код, приведенный в листинге 5.29, настраивает:

- `spring.datasource.url = jdbc:postgresql://localhost:5432/postgres`;
- `spring.datasource.driver-class-name = org.postgresql.Driver` и т. д.

Дополнительно Spring Boot считывает настройки из файлов application-*<активный_профиль>*.yaml и application-*<активный_профиль>*.properties, что позволяет задать дополнительные настройки локального запуска в файле application-local.yaml или application-local.properties, а потом запускать проект на компьютере разработчика с профилем `local`.

Параметры из файла конфигурации профиля переопределяют значения из основного файла конфигурации. Если какая-нибудь настройка указана только в основном файле конфигурации, то используется значение из основного файла. Если же для настройки существует аналогичная настройка из файла конфигурации, специфичная для профиля, то используется настройка уже из него.

Для запуска приложения с нужным профилем необходимо его указать в аргументе командной строки: `--spring.profiles.active`.

В Eclipse аргументы командной строки задаются в горизонтальном меню **Run | Run Configurations** (рис. 5.1). В дополнение к `--spring.profiles.active` в этом поле указываются необходимые параметры запуска приложения. В нашем случае достаточно параметра `spring.profiles.active` со значением `local`.

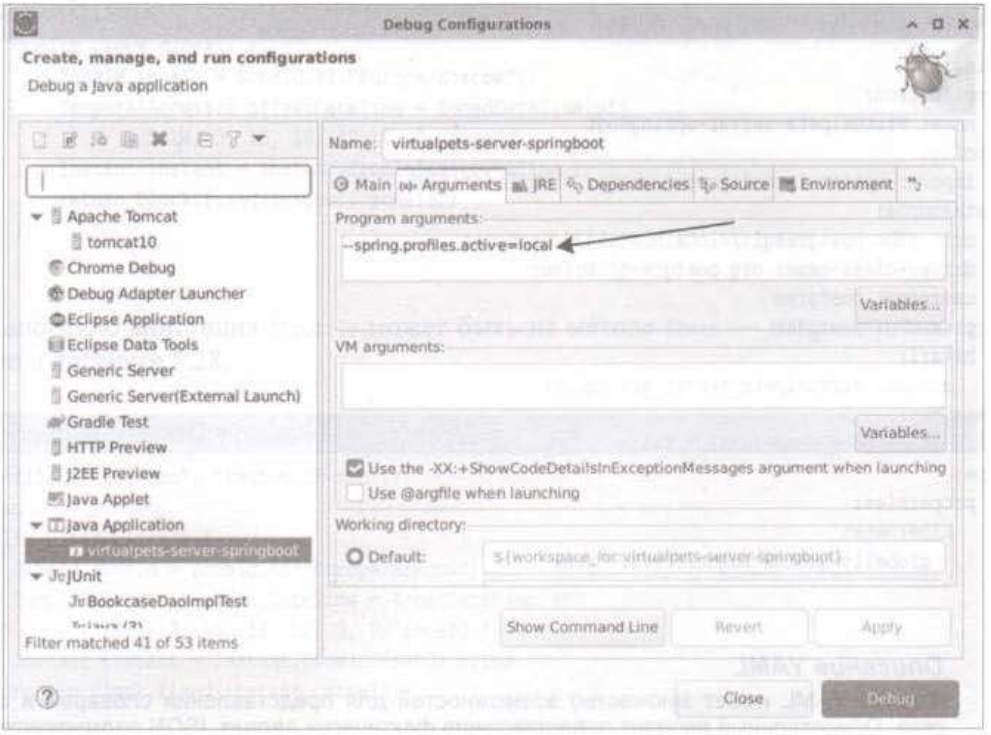


Рис. 5.1. Профиль **local** в конфигурации запуска Eclipse

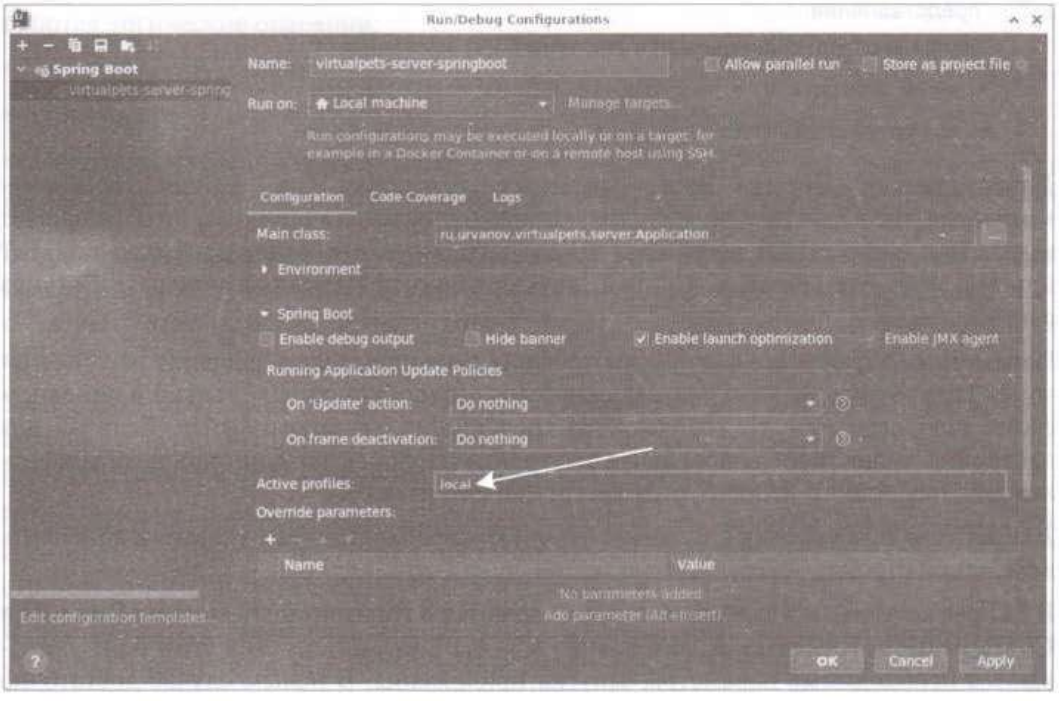


Рис. 5.2. Профиль **local** в конфигурации запуска Spring Boot приложения в IntelliJ IDEA

В IntelliJ IDEA приложения Spring Boot имеют свой тип конфигурации запуска с параметрами, специфичными для Spring Boot. Профили запуска здесь указываются в специальном поле конфигурации запуска **Active profiles** (рис. 5.2).

В логах запуска активные профили выводятся строкой вида:

```
04.06.2024 14:59:56.646 [main] INFO  r.u.v.s.Application - The following 1 profile is active:
"local"
```

5.3. Бины Spring

5.3.1. Объявление

Бины Spring Framework являются основным механизмом, за счет которого осуществляется вся «подкапотная магия» Spring. Как уже рассказывалось в предыдущих разделах, бины Spring описываются либо с помощью XML-конфигурации (листинг 5.30)...

Листинг 5.30. Файл security.xml

```
<bean id = "authenticationProvider"
    class =
"org.springframework.security.authentication.dao.DaoAuthenticationProvider">
    <property name = "userDetailsService" ref = "userDetailsService" />
    <property name = "passwordEncoder" ref = "passwordEncoder" />
</bean>
```

...либо с помощью Java-конфигурации (листинг 5.31)...

Листинг 5.31. Файл SecurityConfig.java

```
@Bean
public DaoAuthenticationProvider authenticationProvider(
        UserDetailsService userService,
        PasswordEncoder passwordEncoder) {
    DaoAuthenticationProvider daoAuthenticationProvider
            = new DaoAuthenticationProvider();
    daoAuthenticationProvider.setUserDetailsService(userService);
    daoAuthenticationProvider.setPasswordEncoder(passwordEncoder);
    return daoAuthenticationProvider;
}
```

...либо с помощью аннотаций и механизма сканирования бинов (листинг 5.32).

Листинг 5.32. Файл RoomServiceImpl.java

```
@Service
public class RoomServiceImpl implements RoomApiService {

    private static final Logger logger = LoggerFactory
            .getLogger(RoomServiceImpl.class);
```

```
    @Autowired
    private RoomDao roomDao;

    @Autowired
    private PetDao petDao;
```

В современном приложении на Spring Boot используются способы через аннотацию `@Bean` (см. листинг 5.31) и со сканированием бинов (см. листинг 5.32).

5.3.2. Жизненный цикл

Бины Spring имеют свой жизненный цикл. Они создаются контейнером и им же уничтожаются. Бины могут подписываться на события своего жизненного цикла тремя способами:

- с помощью аннотаций `@PostConstruct` и `@PreDestroy` из пакета `jakarta.annotation`;
- с помощью интерфейсов `InitializingBean` и `DisposableBean`;
- с помощью атрибутов `init-method` и `destroy-method` из XML-конфигурации бина.

Класс `InitDestroyExample` в примере virtualpets-server-springframework содержит примеры всех этих трех способов.

Аннотации `@PostConstruct` и `@PreDestroy` из пакета `jakarta.annotation` находятся в зависимости, показанной в листинге 5.33.

Листинг 5.33. Файл pom.xml

```
<dependency>
    <groupId>jakarta.annotation</groupId>
    <artifactId>jakarta.annotation-api</artifactId>
    <version>2.1.1</version>
    <scope>provided</scope>
</dependency>
```

Здесь `Scope` указан как `provided`, поскольку эта зависимость предоставляется контейнером Apache Tomcat. Версия зависимости, которую предоставляет конкретная версия Apache Tomcat, указана в документации к этой версии.

При использовании Spring Boot зависимость `jakarta.annotation-api` подтягивается автоматически из spring-boot-starter.

Методы бина, помеченные аннотацией `@PostConstruct`, выполняются после создания экземпляра бина и внедрения всех зависимостей. Этот экземпляр должен содержать дополнительный код инициализации бина, который необходимо выполнить перед его использованием (листинг 5.34).

Листинг 5.34. InitDestroyExample.java

```
import jakarta.annotation.PostConstruct;
...
```

```
public class InitDestroyExample ...

    @PostConstruct
    public void jakartaAnnotationPostConstruct() {
        logger.info("(1) @PostConstruct from jakarta.annotation");
    }
...
```

Методы бина, помеченные аннотацией `@PreDestroy`, выполняются перед уничтожением бина — например, когда приложение получит сигнал остановки (листинг 5.35).

Листинг 5.35. InitDestroyExample.java

```
import jakarta.annotation.PostConstruct;
...
public class InitDestroyExample ...

    @PreDestroy
    public void jakartaAnnotationPreDestroy() {
        logger.info("(4) @PreDestroy from jakarta.annotation");
    }
...
```

Конфигурация бина для класса `InitDestroyExample` находится в файле root-context.xml (листинг 5.36).

Листинг 5.36. Файл root-context.xml

```
<bean class = "ru.urvanov.virtualpets.server.example.InitDestroyExample"
        init-method = "initMethodFromXmlConfiguration"
        destroy-method = "destroyMethodFromXmlConfiguration" />
```

Атрибуты `init-method` и `destroy-method` определяют метод инициализации бина и метод уничтожения бина соответственно — т. е. метод `initMethodFromXmlConfiguration` будет выполняться при инициализации бина после внедрения всех зависимостей, а метод `destroyMethodFromXmlConfiguration` — при уничтожении бина (листинг 5.37).

Листинг 5.37. InitDestroyExample.java

```
public class InitDestroyExample ...

    public void initMethodFromXmlConfiguration() {
        logger.info("(3) init-method from XML-configuration");
    }

    public void destroyMethodFromXmlConfiguration() {
        logger.info("(6) destroy-method from XML-configuration");
    }
```

Интерфейсы `InitializingBean` и `DisposableBean` из пакета `org.springframework.beans.factory` предоставляют еще один способ обработки событий инициализации и уничтожения

бина, но добавляют зависимость от Spring Framework в ваш код, т. к. класс должен явно указать, что он реализует эти интерфейсы.

Интерфейс `InitializingBean` содержит только один метод — `afterPropertiesSet`, реализация которого будет выполняться при инициализации бина после внедрения всех зависимостей (листинг 5.38).

Листинг 5.38. InitDestroyExample.java

```
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
...
public class InitDestroyExample
        implements InitializingBean ... {
...
    @Override
    public void afterPropertiesSet() throws Exception {
        logger.error(
                """
                (2) afterPropertiesSet from \
                InitializingBean interface""");
    }
...
```

Интерфейс `DisposableBean` содержит только один метод `destroy`, реализация которого будет вызываться перед уничтожением бина (листинг 5.39).

Листинг 5.39. InitDestroyExample.java

```
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;
...
public class InitDestroyExample
        implements ... DisposableBean {
...
    @Override
    public void destroy() throws Exception {
        logger.info("(5) destroy from DisposableBean interface");
    }
...
```

В реальных приложениях нет смысла смешивать несколько методов обработки инициализации и уничтожения бина, как это сделано в `InitDestroyExample`, — правильнее будет выбрать один из них и придерживаться его по всему проекту, чтобы упростить чтение и анализ кода в дальнейшем.

Если же в бине смешано несколько способов обработки инициализации и уничтожения, то при создании бина порядок выполнения следующий:

1. Сначала выполняются методы, помеченные `@PostConstruct`.
2. Затем метод `afterPropertiesSet` из интерфейса `InitializingBean`.
3. Потом метод атрибута `init-method` из XML-конфигурации.

При уничтожении бина порядок выполнения следующий:

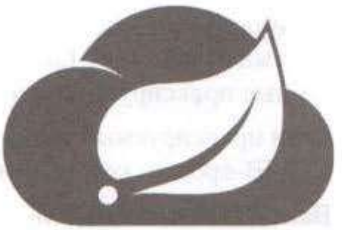
1. Сначала выполняются методы, помеченные `@PreDestroy`.
2. Затем метод `destroy` из интерфейса `DisposableBean`.
3. Потом метод атрибута `destroy-method` из XML-конфигурации.

5.4. Резюме

Spring Framework поддерживает два способа конфигурирования: XML-конфигурацию и Java-конфигурацию. Современные приложения пишутся с использованием Spring Boot и Java-конфигурации. XML-конфигурацию можно увидеть в старых проектах.

Бины Spring позволяют разработчику добавлять свои действия в процессы инициализации и уничтожения бина.

ГЛАВА 6



Аспектно-ориентированное программирование

6.1. Прокси JDK и CGLIB

В предыдущих главах книги рассматривались бины Spring. В этом разделе мы кратко рассмотрим два вида прокси: JDK и CGLIB. Понимание их особенностей и ограничений пригодится при рассмотрении Spring AOP в последующих разделах, т. к. оно реализуется с помощью этих прокси.

В качестве примера представим, что объект `roomService` вызывает метод `addExperience` у объекта `petService`. При обычном вызове это выглядит так, как показано на рис. 6.1.

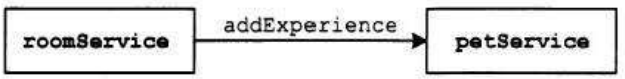


Рис. 6.1. Обычный вызов метода

При использовании Spring AOP объект оборачивается в прокси, и вызывающий объект `roomService` вызывает метод прокси, который, в свою очередь, после выполнения своих действий вызывает реальный метод `petService` (рис. 6.2).

Рис. 6.2. Вызов метода прокси-объекта при Spring AOP

Для создания прокси Spring AOP использует один из двух способов:

- динамические прокси JDK;
- библиотеку генерации байт-кода CGLIB (перепакованную внутри Spring).

> ***Spring Boot по умолчанию использует CGLIB***
>
> В отличие от Spring Framework, автоконфигурация Spring Boot по умолчанию настраивает Spring AOP на использование прокси CGLIB. Для использования прокси JDK необходимо выставить настройку `spring.aop.proxy-target-class` в `false`.

Если проксируемый класс реализует хотя бы один интерфейс, то Spring использует динамический JDK-прокси, который проксирует все методы интерфейсов, реализуемые проксируемым классом.

Если проксируемый класс не реализует ни одного интерфейса, то Spring использует CGLIB-прокси, который основан на генерации байт-кода класса.

Важно понимать, что любой из указанных видов прокси — это не магия. Каждый из них имеет определенные ограничения:

- JDK-прокси несколько медленнее, чем CGLIB-прокси;
- при использовании CGLIB final-методы не могут быть переопределены в генерируемых дочерних классах, поэтому на них не работают аспекты.

6.2. Аспекты Spring

6.2.1. Аннотация @Transactional

Практически каждое Spring-приложение использует аспектно-ориентированное программирование. Например, аннотация `@Transactional`, задействуемая для управления транзакциями, работает на основе Spring AOP (листинг 6.1).

Листинг 6.1. ClothDaoImpl.java

```
...
public class ClothDaoImpl implements ClothDao {
...
    @Override
    @Transactional(readOnly = true)
    public Optional<Cloth> findById(String id) {
        Cloth cloth = em.find(Cloth.class, id);
        return Optional.ofNullable(cloth);
    }
```

Полученных вами знаний вполне достаточно...

Теоретического материала про аспектно-ориентированное программирование из *разд. 1.7* и приведенной здесь информации о `@Transactional` вам будет достаточно в большинстве ситуаций. Редко случается так, что действительно приходится разрабатывать аспекты самостоятельно. Да и в приложении сервера виртуальных питомцев нет полноценного использования аспектов, т. к. им там просто не нашлось места. Аналогично и в коммерческих приложениях — в большинстве случаев самостоятельно описывать срезы и советы не требуется.

6.2.2. Подключение зависимостей

Для использования аспектно-ориентированного программирования сначала необходимо подключить зависимости.

Зависимости для приложения на Spring Framework приведены в листинге 6.2.

Листинг 6.2. Файл pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>${org.springframework-version}</version>
</dependency>
```

Для создания аспектов с помощью аннотации `@AspectJ` также потребуется зависимость (листинг 6.2), где `${org.springframework-version}` и `${org.aspectj-version}` — это версии Spring Framework и AspectJ, объявленные в секции `properties` (листинг 6.3).

Листинг 6.2. Файл pom.xml

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>${org.aspectj-version}</version>
</dependency>
```

Листинг 6.3. Файл pom.xml

```
<properties>
    ...
    <org.springframework-version>6.0.11</org.springframework-version>
    <org.aspectj-version>1.9.21</org.aspectj-version>
    ...
</properties>
```

Зависимости для приложения на Spring Boot приведены в листинге 6.4.

Листинг 6.4. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

Spring предоставляет два способа описания аспектов, которые будут рассмотрены далее:

- на основе XML-конфигурации;
- на основе аннотаций `@AspectJ`.

6.2.3. XML-конфигурация АОП

С помощью XML-конфигурации аспекты в приложении описываются с использованием пространства имен `http://www.springframework.org/schema/aop`, которое обычно связывается с префиксом `aop` (листинг 6.5).

Листинг 6.5. Файл root-context.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
...
    xmlns:aop = "http://www.springframework.org/schema/aop"
...
    xsi:schemaLocation = "
...
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/task/spring-task.xsd">
...
```

Сами аспекты определяются внутри тега `aop:config` (листинг 6.6).

Листинг 6.6. Файл root-context.xml

```
<bean id = "schemaBasedAdvice"
    class = "ru.urvanov.virtualpets.server.example.SchemaBasedAdvice" />

<aop:config>
     <aop:aspect id = "beforeAspectExample" ref = "schemaBasedAdvice">
        ... Советы и срезы
    </aop:aspect>
</aop:config>
```

В приведенном примере объявлен бин `schemaBasedAdvice`, который содержит методы с логикой советов.

Внутри тега `aop:aspect` описываются советы и срезы. Вот, например, описание среза `beforeDaoPointcut` для всех методов в классах, находящихся внутри пакета слоя постоянства (листинг 6.7).

Листинг 6.7. Файл root-context.xml

```
...
<aop:config>
    <aop:aspect id = "beforeAspectExample" ref = "schemaBasedAdvice">
        <aop:pointcut id = "beforeDaoPointcut"
    expression = "execution(* ru.urvanov.virtualpets.server.dao.*.*(..))" />
...
    </aop:aspect>
</aop:config>
...
```

А вот объявленный для него совет, который будет выполняться перед методами этого среза (листинг 6.8).

Листинг 6.8. Файл root-context.xml

```
...
<aop:before pointcut-ref = "beforeDaoPointcut"
    method = "beforeDaoAdvice" />
...
```

Атрибут `pointcut-ref` здесь указывает на срез, а в атрибуте `method` указывается наименование метода в классе с советами. В нашем примере метод `beforeDaoAdvice` выглядит так, как показано в листинге 6.9.

Листинг 6.9. SchemaBasedAdvice.java

```
package ru.urvanov.virtualpets.server.example;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import ru.urvanov.virtualpets.server.controller.api.domain.HiddenObjectsGame;

public class SchemaBasedAdvice {
...
    public void beforeDaoAdvice(JoinPoint joinPoint) {
        logger.info("Accessing DAO layer {}.{}, arguments count {}.",
                joinPoint.getTarget().getClass().getName(),
                joinPoint.getSignature().getName(),
                joinPoint.getArgs().length);
    }
...
```

Метод `beforeDaoAdvice` будет выполняться перед выполнением методов бинов из слоя доступа к данным. В качестве параметра он принимает `joinPoint`, содержащий информацию об аргументах, сигнатуре вызываемого метода, классе и т. д. и выводит эту информацию в лог.

> ***Совет из примера искусственный...***
>
> Он не несет действительно полезных приложению действий. Это не просто так. В большинстве случаев использование аспектно-ориентированного программирования, кроме `@Transactional` и подобных стандартных механизмов, приложению не нужно. В вашей работе вам, скорее всего, тоже редко придется создавать самостоятельные аспекты.

В рассмотренном примере при описании аспекта, который выполняется перед методами слоя постоянства, описание среза было вынесено в отдельный блок, а у среза имелся свой `id = beforeDaoPointcut` (см. листинг 6.7). Но срезы не обязательно описывать отдельно, а потом на них ссылаться через `pointcut-ref`, — можно сразу описать выражение среза внутри атрибута `pointcut` в теге `aspect`, как это сделано в следующем примере (листинг 6.10).

Листинг 6.10. Файл root-context.xml

```
...
<aop:config>
    <aop:aspect id = "beforeAspectExample" ref = "schemaBasedAdvice">
...
    <aop:after-returning
        pointcut = "execution(*
              ru.urvanov.virtualpets.server.service.HiddenObjectsServiceImpl.joinGame(..))"
        method = "afterJoinGameReturningAdvice"
        returning = "hiddenObjectsGame" />
...
    </aop:aspect>
</aop:config>
...
```

Здесь использован срез `aop:after-returning` для описания совета, который будет выполняться после успешного завершения метода `joinGame` бина типа `HiddenObjectsServiceImpl`. Обратите внимание, что в атрибуте `returning` указывается наименование параметра совета, в который будет передано возвращаемое методом значение.

Реализация метода `afterJoinGameReturningAdvice` в классе `SchemaBasedAdvice` приведена в листинге 6.11.

Листинг 6.11. SchemaBasedAdvice.java

```
...
    public void afterJoinGameReturningAdvice(
            JoinPoint joinPoint,
            HiddenObjectsGame hiddenObjectsGame) {
              logger.info("""
                      A player joined hidden objects game. \
                      Arguments count {}. \
                      Returning {}. \
                      """,
                      joinPoint.getArgs().length,
                      hiddenObjectsGame);
    }
...
```

Наш совет `after-returning` просто выводит в лог информацию, что игрок подключился к мини-игре с поиском скрытых предметов, а также количество переданных аргументов метода.

Совет `after-throwing` выполняется, когда метод из среза бросает исключение (листинг 6.12).

Листинг 6.12. Файл root-context.xml

```
...
<aop:config>
    <aop:aspect id = "beforeAspectExample" ref = "schemaBasedAdvice">
...
    <aop:after-throwing
        pointcut = "execution(*
ru.urvanov.virtualpets.server.service.RoomServiceImpl.build*(..))"
        throwing = "throwableParameter"
        method = "afterBuildThrowingAdvice" />
...
    </aop:aspect>
</aop:config>
...
```

`RoomServiceImpl.build*` **в паттерне** `execution` **у среза означает, что если методы** `buildRefrigerator`, `buildMachineWithDrinks`, `buildBookcase` **бросят исключение, то будет выполняться совет** `afterBuildThrowingAdvice`. **В атрибуте** `throwing` **указывается наименование параметра, в который передается бросаемое исключение (листинг 6.13).**

Листинг 6.13. SchemaBasedAdvice.java

```
...
public void afterBuildThrowingAdvice(
        JoinPoint joinPoint,
        Throwable throwableParameter) {
    logger.info("""
                A build method thrown an exception. \
                Throwing {} \
                Arguments count {}. \
                """,
            throwableParameter,
            joinPoint.getArgs().length);
}
...
```

Совет `after` **отрабатывается после завершения метода вне зависимости от успешности его выполнения (листинг 6.14).**

Листинг 6.14. Файл root-context.xml

```
<aop:config>
    <aop:aspect id = "beforeAspectExample" ref = "schemaBasedAdvice">
...
    <aop:after
        pointcut = "execution(* ru.urvanov.virtualpets.server.service.RoomServiceImpl.build*(..))"
        method = "afterBuildAdvice" />
...
    </aop:aspect>
</aop:config>
```

Соответствующий ему метод `afterBuildAdvice` класса `SchemaBasedAdvice` приведен в листинге 6.15.

Листинг 6.15. SchemaBasedAdvice.java

```
...
public void afterBuildAdvice(JoinPoint joinPoint) {
    logger.info("Build finished. Arguments count {}.", joinPoint.getArgs().length);
}
...
```

Совет `around` выполняется вместо вызываемого метода. Внутри себя он может вызвать вызываемый изначально метод либо после выполнения своих действий, либо перед их выполнением, либо не вызывать его вообще (листинг 6.16).

Листинг 6.16. Файл root-context.xml

```
...
<aop:config>
    <aop:aspect id = "beforeAspectExample" ref = "schemaBasedAdvice">
...
    <aop:around
        pointcut = "execution(*
                    ru.urvanov.virtualpets.server.service.RoomServiceImpl.openBoxNewbie(..))"
        method = "aroundOpenBoxNewbieAdvice" />
...
    </aop:aspect>
</aop:config>
...
```

Метод `aroundOpenBoxNewbieAdvice` принимает параметр `ProceedingJoinPoint`, метод `proceed()` которого вызывает целевой метод бина (листинг 6.17).

Листинг 6.17. SchemaBasedAdvice.java

```
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
...
public Object aroundOpenBoxNewbieAdvice(
        ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
    logger.info("We are in openBoxNewbiew around advice.");
    return proceedingJoinPoint.proceed();
}
...
```

Обратите внимание, что возвращаемое методом `aroundOpenBoxNewbieAdvice` значение станет возвращаемым значением исходного метода.

Аспекты описываются не только в XML-конфигурации, но и с помощью аннотаций. Эти два способа взаимозаменяемы, их даже можно использовать совместно —

например, описать срез в XML-конфигурации, а использовать его на совете, описанном с помощью аннотаций.

Для включения конфигурации с помощью аннотации `@AspectJ` в Spring AOP необходимо добавить ее в XML-файл конфигурации (листинг 6.18).

Листинг 6.18. Файл root-context.xml

```
<aop:aspectj-autoproxy />
```

6.2.4. Java-конфигурация АОП

Для случая Java-конфигурации необходимо указать аннотацию `@EnableAspectJAutoProxy` на файле с аннотацией `@Configuration`.

При использовании Spring Boot никакой дополнительной настройки не нужно — достаточно добавления `spring-boot-starter-aop` в зависимости от проекта (листинг 6.19).

Листинг 6.19. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

Для объявления аспектов в Spring Boot нам достаточно над бином добавить аннотацию `@Aspect` из пакета `org.aspectj.lang.annotation.Aspect` (листинг 6.20).

Листинг 6.20. AnnotationBasedAspect.java

```
package ru.urvanov.virtualpets.server.example;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import ru.urvanov.virtualpets.server.controller.api.domain.HiddenObjectsGame;

@Aspect
@Component
public class AnnotationBasedAspect {
    private static final Logger logger = LoggerFactory.getLogger(
        AnnotationBasedAspect.class);
    // ... объявления советов
}
```

Сами советы объявляются с помощью аннотаций над методами бина, который помечен аннотацией `@Aspect`. Например, совет, который будет выполняться перед всеми контроллерами встроенного мини-сайта из соответствующего пакета, создается с помощью аннотации `@Before`, в параметре `pointcut` которого задается срез (листинг 6.21).

Листинг 6.21. AnnotationBasedAspect.java

```
...
@Before("""
        execution(* \
        ru.urvanov.virtualpets.server.controller.site.*.*\
        (..))""")
public void beforeSitePage(JoinPoint joinPoint) {
    logger.info("Requesting controller {}.{}, arguments count {}.",
            joinPoint.getTarget().getClass().getName(),
            joinPoint.getSignature().getName(),
            joinPoint.getArgs().length);
}
...
```

Параметр с типом `JoinPoint` уже должен быть знаком вам из советов, созданных на основе XML-конфигурации. Он содержит информацию о вызываемом классе, методе и переданных аргументах.

Вот совет, выполняющийся после успешного возвращения из метода `getGameInfo` сервиса `HiddenObjectServiceImpl` (листинг 6.22).

Листинг 6.22. AnnotationBasedAspect.java

```
...
@AfterReturning(
        value = """
                execution(* \
                ru.urvanov.virtualpets.server.service\
                .HiddenObjectsServiceImpl.getGameInfo\
                (..))""",
        returning = "hiddenObjectsGame")
public void afterReturningHiddenObjectGameInfo(
        JoinPoint joinPoint,
        HiddenObjectsGame hiddenObjectsGame) {
    logger.info("""
            HiddenObjectsGame finished. \
            Arguments count {}. \
            Result {}. \
            """,
            joinPoint.getArgs().length,
            hiddenObjectsGame);
}
```

В аннотации `@AfterReturning` в параметре `returning` указывается название параметра совета, в который будет передаваться результат работы вызванного метода. В нашем случае это объект типа `HiddenObjectsGame`, возвращенный из метода `getGameInfo`.

Вот совет, выполняющийся в случаях, когда метод открытия коробки с лутбоксами бросает исключение (листинг 6.23).

Листинг 6.23. AnnotationBasedAspect.java

```
...
@AfterThrowing(
        value = """
                execution(* \
                ru.urvanov.virtualpets.server.service\
                .RoomServiceImpl.openBoxNewbie\
                (..))""",
        throwing = "throwableParameter"
        )
public void openBoxNewbieException(
        JoinPoint joinPoint,
        Throwable throwableParameter) {
    logger.info("""
            OpenBoxNewbie thrown an exception. \
            Arguments count {}. \
            Throwing \
            """,
        joinPoint.getArgs().length,
        throwableParameter);
}
...
```

В аннотации `@AfterThrowing` в параметре `throwing` указывается параметр метода-совета, в который будет передаваться исключение, брошенное на вызванном методе.

С помощью аннотации `@After` создается совет, выполняющийся после завершения метода независимо от результата. Совет будет выполняться как для случая, когда выполнение вызываемого метода бросило исключение, так и при успешном выполнении метода. Этим он напоминает логику `finally` (листинг 6.24).

Листинг 6.24. AnnotationBasedAspect.java

```
...
@After("""
        execution(* \
        ru.urvanov.virtualpets.server.service\
        .RoomServiceImpl.move*\
        (..))""")
public void afterBuildingMoved(JoinPoint joinPoint) {
    logger.info("""
            After building moved. \
            Arguments count {}. \
            """,
```

```
            joinPoint.getArgs().length);
}
...
```

Совет, выполняющийся вместо вызываемого метода, создается с помощью аннотации `@Around` (листинг 6.25).

Листинг 6.25. AnnotationBasedAspect.java

```
...
@Around("""
        execution(* \
        ru.urvanov.virtualpets.server.service.\
        RoomServiceImpl.getBuildMenuCosts\
        (..))""")
public Object aroundGetBuildMenuCosts(
        ProceedingJoinPoint proceedingJoinPoint)
        throws Throwable {
    logger.info("around getBuildMenuCosts advice");
    return proceedingJoinPoint.proceed();
}
...
```

Аналогично варианту с XML-конфигурацией совет принимает параметр с типом `ProceedingJoinPoint`, с помощью метода `proceed()` которого мы вызываем исходный метод.

Обратите внимание, что метод `aroundGetBuildMenuCosts` возвращает результат метода `proceed`. Допускается изменять возвращаемое значение, а также возвращать другое значение вместо исходного, что приведет к изменению возвращаемого значения исходного метода.

6.3. Библиотека AspectJ

Spring AOP имеет определенные ограничения:

- позволяет использовать в качестве точек соединения только методы;
- основан на использовании прокси-объектов и может быть применен только к бинам;
- поддерживает только связывание в момент выполнения.

В большинстве случаев возможностей Spring AOP вполне достаточно — вам вряд ли когда-либо придется использовать библиотеку AspectJ в реальных приложениях, но если же по какой-либо причине вам понадобится что-либо более мощное, чем аспектно-ориентированное программирование, предоставляемое Spring, то вы можете подключить перепакованную библиотеку AspectJ:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
```

```
    <version>${org.springframework-version}</version>
</dependency>
```

где `${org.springframework-version}` — версия Spring Framework, объявленная в блоке `properties`.

С помощью AspectJ вы сможете:

- внедрять зависимости в классы, созданные вне контейнера бинов Spring, — например, в классы предметной области, которые обычно создаются через `new`;
- использовать способ связывания, отличный от связывания во время выполнения. Например, AspectJ поддерживает связывание во время загрузки классов;
- управлять транзакциями Spring для классов вне контейнера бинов.

> ***Примечание***
>
> В целом вам вряд ли когда-либо всё это пригодится, но на случай, если эти возможности будут необходимы, полезно знать об их существовании.

6.4. Резюме

Аспектно-ориентированное программирование позволяет добавлять в приложение сквозную логику, не затрагивая основной код. Spring поддерживает АОП с помощью прокси-объектов.

Используются два вида прокси:

- JDK;
- CGLIB.

Поддерживается конфигурация аспектов как с помощью XML-, так и с помощью Java-конфигурации. Существует возможность использовать библиотеку AspectJ, если возможностей Spring AOP недостаточно.

В большинстве коммерческих приложений нет необходимости самостоятельно создавать аспекты — чаще всего использования стандартных механизмов наподобие `@Transactional` достаточно.

ГЛАВА 7

Работа с базами данных

7.1. Слой постоянства

Сервису виртуальных питомцев необходимо где-то сохранять информацию об игроках, о созданных ими питомцах, полученных достижениях, прогрессе игры, собранных ресурсах, записях в дневнике. В большинстве случаев в современном мире для этих целей используются реляционные базы данных — чаще всего это PostgreSQL, либо Oracle, либо Microsoft SQL Server.

В связи с последними мировыми событиями наиболее разумным выбором для новых приложений будет PostgreSQL.

Сервер приложения использует СУБД PostgreSQL, в которой хранит данные: в схеме virtualpets_server_springframework — для базы данных варианта тестового приложения на Spring Framework, и в схеме virtualpets_server_springboot — для варианта тестового приложения на Spring Boot.

В игре присутствуют две самые основные сущности: пользователь и питомец. Один пользователь может создать несколько питомцев. Почти все остальные данные привязаны к питомцу: дневник, достижения, собранные ресурсы, опыт, уровень, прогресс в игре. К самому пользователю, кроме питомцев, привязан только чат.

ER-диаграмма базы данных в виде рисунка со всеми сущностями, связями между ними и полями расположена на странице книги[1].

Сам сервер обращается к базе данных с помощью JDBC, при этом он либо формирует SQL-команды и отправляет их СУБД через абстракцию `JdbcTemplate`, либо использует JPA, Hibernate и Spring Data, рассмотренные подробнее в следующих разделах главы.

Для работы с PostgreSQL серверу игры виртуальных питомцев необходимо добавить зависимость от JDBC-драйвера.

Зависимость от JDBC-драйвера PostgreSQL для варианта сервера виртуальных питомцев virtualpets-server-springframework представлена в листинге 7.1.

---

[1] См. **https://urvanov.ru/книги/spring-book-2024/virtualpets-database-erd-2024-10-12/**.

Листинг 7.1. Файл pom.xml

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.6.0</version>
    <scope>provided</scope>
</dependency>
```

Обратите внимание, что `scope` указан как `provided` — это значит, что зависимость будет предоставлена окружением, а значит, эту зависимость не нужно добавлять при упаковке в jar-файл. Сам файл драйвера JDBC необходимо скачать с **https://jdbc.postgresql.org/** и расположить в каталоге lib установленного экземпляра Apache Tomcat.

Почему нельзя указать `scope compile`? Дело в том, что JDBC-драйверы регистрируют себя в общем для всех приложений синглетоне DriverManager. Если два приложения зарегистрируют два раза один и тот же JDBC-драйвер, то это может привести к проблеме. Сканирование JDBC-драйверов выполняется только один раз при старте Apache Tomcat, при этом драйверы JDBC ищутся только в каталогах $CATALINA_HOME/lib и $CATALINA_BASE/lib. Подробнее эта проблема описана в разделе JNDI Datasource How-To документации на Apache Tomcat.

Подключение к базе данных оформлено как JNDI-ресурс — в самом приложении сервера виртуальных питомцев бин `dataSource`, описывающий источник данных, представлен в файле servlet-jee.xml (листинг 7.2).

Листинг 7.2. Файл servlet-jee.xml

```
<bean id = "dataSource"
    class = "org.springframework.jndi.JndiObjectFactoryBean">
    <property name = "jndiName"
        value = "java:/comp/env/jdbc/virtualpetsDB" />
</bean>
```

Кроме бина `dataSource`, необходимо обязательно настроить управление транзакциями с помощью `tx:annotation-driven`, как это сделано в файле servlet-tx.xml (листинг 7.3).

Листинг 7.3. Файл servlet-tx.xml

```
<bean id = "transactionManager"
    class = "org.springframework.orm.jpa.JpaTransactionManager">
    <property name = "entityManagerFactory" ref = "emf" />
    <property name = "dataSource" ref = "dataSource" />
</bean>

<tx:annotation-driven transaction-manager = "transactionManager" />
```

Здесь `JpaTransactionManager` — это реализация интерфейса `PlatformTransactionManager`, специализированная на управлении транзакциями JPA.

Существуют и другие реализации `PlatformTransactionManager`, в том числе:

- `JdbcTransactionManager` — специализированный на управлении транзакциями для `JdbcTemplate`;
- `HibernateTransactionManager` — специализированный на управлении транзакциями для `SessionFactory` из Hibernate.

После настройки транзакций `tx:annotation-driven` управление транзакциями осуществляется добавлением аннотации `@Transactional` к методам и классам — например, как показано в листинге 7.4.

Листинг 7.4. ClothDaoImpl.java

```
@Override
@Transactional(readOnly = true)
public Optional<Cloth> findById(String id) {
    Cloth cloth = em.find(Cloth.class, id);
    return Optional.ofNullable(cloth);
}
```

В варианте сервера виртуальных питомцев для Spring Boot зависимость от JDBC-драйвера PostgreSQL подключается со `scope compile`, но без версии, т. к. версия JDBC-драйвера PostgreSQL задается в `spring-boot-parent` (листинг 7.5).

Листинг 7.5. Файл pom.xml

```
...
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
</dependency>
...
```

Подключение к базе данных настраивается в файле application.yaml (листинг 7.5).

Листинг 7.6. Файл application.yaml

```
spring:
...
  datasource:
    url: jdbc:postgresql://localhost:5432/postgres
    driver-class-name: org.postgresql.Driver
    username: postgres
    password: postgres
    hikari:
      schema: virtualpets_server_springboot
```

В качестве пула по умолчанию Spring Boot использует HikariCP, но поддерживаются и другие пулы соединений. Зависимость от HikariCP согласно документации в virtualpets-server-springboot появляется как транзитивная зависимость от `spring-`

boot-starter-jdbc и spring-boot-starter-jpa, поэтому добавлять ее дополнительно в файл pom.xml не нужно, — Spring Boot автоматические настраивает управление транзакциями при наличии spring-boot-starter-jpa или spring-boot-starter-jdbc в зависимостях проекта.

7.2. Библиотека Liquibase

7.2.1. Подключение зависимостей

Как пример virtualpets-server-springframework, так и пример virtualpets-server-springboot задействуют библиотеку Liquibase для создания структуры таблиц, используемой слоем постоянства.

> Liquibase — это библиотека для отслеживания и управления изменениями схемы базы данных.

Для подключения зависимости к Maven-проекту в файл pom.xml эту библиотеку необходимо добавить (листинг 7.7).

Листинг 7.7. Файл pom.xml

```
<dependency>
    <groupId>org.liquibase</groupId>
    <artifactId>liquibase-core</artifactId>
    <version>4.25.1</version>
</dependency>
```

При использовании Spring Boot версию Liquibase указывать не нужно (листинг 7.8).

Листинг 7.8. Файл pom.xml

```
<dependency>
    <groupId>org.liquibase</groupId>
    <artifactId>liquibase-core</artifactId>
</dependency>
```

7.2.2. Настройка для Spring Framework

Настройка Liquibase для Spring Framework осуществляется конфигурированием бина SpringLiquibase. В проекте виртуальных питомцев virtualpets-server-springframework пример создания подобного бина приведен в файле servlet-tx.xml (листинг 7.9).

Листинг 7.9. Файл servlet-tx.xml

```
<bean id = "myLiquibase"
    class = "liquibase.integration.spring.SpringLiquibase">
    <property name = "dataSource" ref = "dataSource" />
    <property name = "changeLog"
        value = "classpath:liquibase/db-changelog.xml" />
```

```
    <property name = "defaultSchema"
        value = "virtualpets_server_springframework" />
</bean>
```

Основные поля класса `SpringLiquibase`:

- `dataSource` — источник данных;
- `changeLog` — файл в `classpath` со списком изменений схемы базы данных;
- `defaultSchema` — схема базы данных по умолчанию.

Класс `SpringLiquibase` осуществляет интеграцию Liquibase со Spring Framework и запуск миграций схемы базы данных после инициализации контекста Spring.

Файл src/main/resources/liquibase/db-changelog.xml содержит описание всех миграций, включая каждую из них с помощью `include` из отдельного файла, содержащегося в соответствующем подкаталоге. В примере сервера виртуальных питомцев содержится одна миграция, которая указывает на файл с SQL-командами первичной инициализации схемы базы данных (листинг 7.10).

Листинг 7.10. Файл db-changelog.xml

```
<databaseChangeLog xmlns = "http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://www.liquibase.org/xml/ns/dbchangelog
        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.4.xsd">

    <include file = "classpath:liquibase/20240102/schema.xml"
        relativeToChangelogFile="false" />

</databaseChangeLog>
```

Файл миграции liquibase/20240102/schema.xml показан в листинге 7.11.

Листинг 7.11. Файл schema.xml

```
<?xml version = "1.1" encoding = "UTF-8" standalone = "no"?>
<databaseChangeLog xmlns = "http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://www.liquibase.org/xml/ns/dbchangelog
        http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-3.5.xsd">

    <include relativeToChangelogFile = "true" file = "schema.sql"/>

</databaseChangeLog>
```

В файле schema.sql находятся DML-команды, создающие первоначальное состояние схемы данных, используемой сервером виртуальных питомцев.

Структура каталогов и файлов с миграциями Liquibase, используемая virtualpets-server-springframework, показана на рис. 7.1.

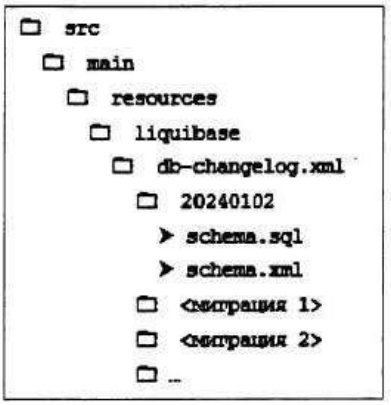


Рис. 7.1. Дерево каталогов с миграциями Liquibase

Под файлы каждой миграции предполагается создавать отдельный каталог. На текущий момент миграция только одна, поэтому и каталог здесь только один — с датой в названии **20240102**.

> ***Структура миграций Liquibase не обязательно должна быть такой***
> В вашем проекте может быть принята другая структура каталогов и файлов для миграций Liquibase, более подходящая под требования вашего проекта.

7.2.3. Настройка для Spring Boot

Для Spring Boot, использующегося в virtualpets-server-springboot, создавать бин `SpringLiquibase` не нужно — Spring Boot при обнаружении Liquibase в `classpath` сам произведет необходимые настройки и создаст этот бин с помощью автоконфигурации. Если какие-либо настройки Liquibase по умолчанию не подходят вашему приложению, то их значения меняются с помощью ключей `spring.liquibase` в файле application.yaml (листинг 7.12).

Листинг 7.12. Файл application.yaml

```
spring:
...
  liquibase:
    default-schema: virtualpets_server_springboot
```

Ключ `spring.liquibase.default-schema` задает схему, которую по умолчанию будет использовать Liquibase при выполнении миграций.

В качестве основного файла миграций по умолчанию используется db/changelog/db.changelog-master.yaml, но это можно изменить с помощью ключа `spring.liquibase.change-log`. В примере сервера виртуальных питомцев используется расположение по умолчанию (листинг 7.13).

Листинг 7.13. Файл db.changelog-master.yaml

```
databaseChangeLog:
  - include:
      file: 20240102/schema.yaml
      relativeToChangelogFile: true
```

Само содержимое файла аналогично содержимому XML-файла из примера для Spring Framework, но переделано в формат YAML. Структура используемых каталогов также аналогична.

Файл миграции schema.yaml аналогичен файлу schema.xml из примера для Spring Framework, но в формате YAML (листинг 7.14).

Листинг 7.14. Файл schema.yaml

```
databaseChangeLog:
  - include:
      file: schema.sql
      relativeToChangelogFile: true
```

Файлы с DDL-командами в примерах для Spring Framework и для Spring Boot идентичны практически полностью, за исключением разных используемых схем: virtualpets_server_springframework — для Spring Framework, и virtualpets_server_springboot — для Spring Boot.

7.3. Spring JDBC

7.3.1. Подключение зависимостей

Spring Framework предоставляет полную поддержку создания слоя постоянства на JDBC и самостоятельного написания разработчиком SQL-запросов, но в современных приложениях это используется редко. В реальных приложениях вам, скорее всего, придется работать с JPA, Spring Data, Hibernate, MyBatis, Query DSL, Criteria API и другими подобными им технологиями.

Если же в вашем проекте используется именно JDBC напрямую, то Spring предоставляет абстракции `JdbcTemplate`, `NamedParameterJdbcTemplate` и `JdbcClient`, появившиеся в Spring 6.1. Для работы со всеми этими абстракциями в проект нужно добавить дополнительную зависимость.

Зависимость для версии приложения на Spring Framework, пример которого находится в проекте virtualpets-server-springframework, где `${org.springframework-version}` — версия Spring Framework, объявленная в разделе `properties` pom-файла (листинг 7.15).

Листинг 7.15. Файл pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
```

```
    <artifactId>spring-jdbc</artifactId>
    <version>${org.springframework-version}</version>
</dependency>
```

Зависимость для приложения на Spring Boot, пример которого находится в virtualpets-server-springboot, представлена в листинге 7.16.

Листинг 7.16. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

7.3.2. Абстракция *JdbcTemplate*

Абстракция `JdbcTemplate` обычно создается в методе `setDataSource`, который через аннотацию `@Autowired` получает источник данных из контейнера Spring, как это сделано в `JdbcReportDaoImpl` в проекте virtualpets-server-springframework (листинг 7.17).

Листинг 7.17. JdbcReportDaoImpl.java

```
package ru.urvanov.virtualpets.server.dao;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;
...
@Repository
public class JdbcReportDaoImpl implements JdbcReportDao {

    private JdbcTemplate jdbcTemplate;

    private LastRegisteredUserMapper lastRegisteredUsersMapper
            = new LastRegisteredUserMapper();

    @Autowired
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }
...
```

Абстракция `JdbcTemplate` потокобезопасна, поэтому один ее созданный экземпляр можно сохранить в поле экземпляра класса, как это сделано в `JdbcReportDaoImpl`, и использовать ее во всех методах, не создавая новую.

Выполнение SQL-запросов осуществляется методом `query` при выборке данных либо методом `update` при изменении данных (SQL insert, SQL update, SQL delete), *как показано в листинге 7.18.*

Листинг 7.18. JdbcReportDaoImpl.java

```
...
private LastRegisteredUserMapper lastRegisteredUsersMapper
        = new LastRegisteredUserMapper();
...
@Override
@Transactional(readOnly = true)
public List<LastRegisteredUser> findLastRegisteredUsers(int start, int limit) {
    return jdbcTemplate.query("""
            select
              u.id as id,
              u.registration_date as registration_date,
              u.name as name,
              count(p.id) as pets_count
            from "user" u
              left join pet p on p.user_id = u.id
            group by
              u.id,
              u.registration_date,
              u.name
            order by registration_date desc offset ? limit ?
            """,
            lastRegisteredUsersMapper,
            start,
            limit);
}
...
```

В приведенном коде в метод `query` класса `JdbcTemplate` передается SQL-запрос на выборку данных, маппер `LastRegisteredUserMapper` и параметры запроса.

Маппер `LastRegisteredUserMapper` — это класс, реализующий интерфейс `RowMapper` и осуществляющий отображение данных из строки результатов запроса на класс Java из предметной области (листинг 7.19).

Листинг 7.19. LastRegisteredUserMapper

```
package ru.urvanov.virtualpets.server.dao.mapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import ru.urvanov.virtualpets.server.dao.domain.LastRegisteredUser;
```

```
public class LastRegisteredUserMapper
        implements RowMapper<LastRegisteredUser> {

    @Override
    public LastRegisteredUser mapRow(ResultSet rs, int rowNum)
            throws SQLException {
        LastRegisteredUser lastRegisteredUser = new LastRegisteredUser();
        lastRegisteredUser.setId(rs.getInt("id"));
        lastRegisteredUser.setRegistrationDate(
                rs.getTimestamp("registration_date"));
        lastRegisteredUser.setName(rs.getString("name"));
        lastRegisteredUser.setPetsCount(rs.getLong("pets_count"));
        return lastRegisteredUser;
    }
}
```

Для проверки работоспособности метода выполните следующие шаги:

1. Поставьте точку остановки на одной из инструкций метода `JdbcReportDaoImpl#findLastRegisteredUsers`.
2. Запустите virtualpets-server-springframework по инструкции из *разд 3.1*.
3. Откройте в браузере адрес: **http://localhost:8080/virtualpets-server-springframework/site/home**.
4. Перейдите на открывшейся странице в раздел **Информация | Статистика**, выберите из выпадающего списка **Последние зарегистрированные** и нажмите кнопку **Показать статистику**.

Ваша IDE должна остановиться на строке, выбранной в пункте 1.

Если вы продолжите выполнение программы с точки остановки, то в браузере должна отобразиться страница, пример которой показан на рис. 7.2.

> ***Примечание***
>
> В этом разделе рассматривается только часть получения данных — отображение HTML-страниц с результатами будет рассмотрено позднее в соответствующих разделах.

Spring Boot инициализирует экземпляр класса `JdbcTemplate`, поэтому в `JdbcReportDaoImpl` достаточно просто использовать аннотацию `@Autowired` для внедрения уже созданного бина (листинг 7.20).

Листинг 7.20. JdbcReportDaoImpl.java

```
@Repository
public class JdbcReportDaoImpl implements JdbcReportDao {

    @Autowired
    private JdbcTemplate jdbcTemplate;
```

В проекте virtualpets-server-springboot абстракция `JdbcTemplate` не используется — вместо нее задействуется `JdbcClient`, описанная в *разд. 7.3.3*.

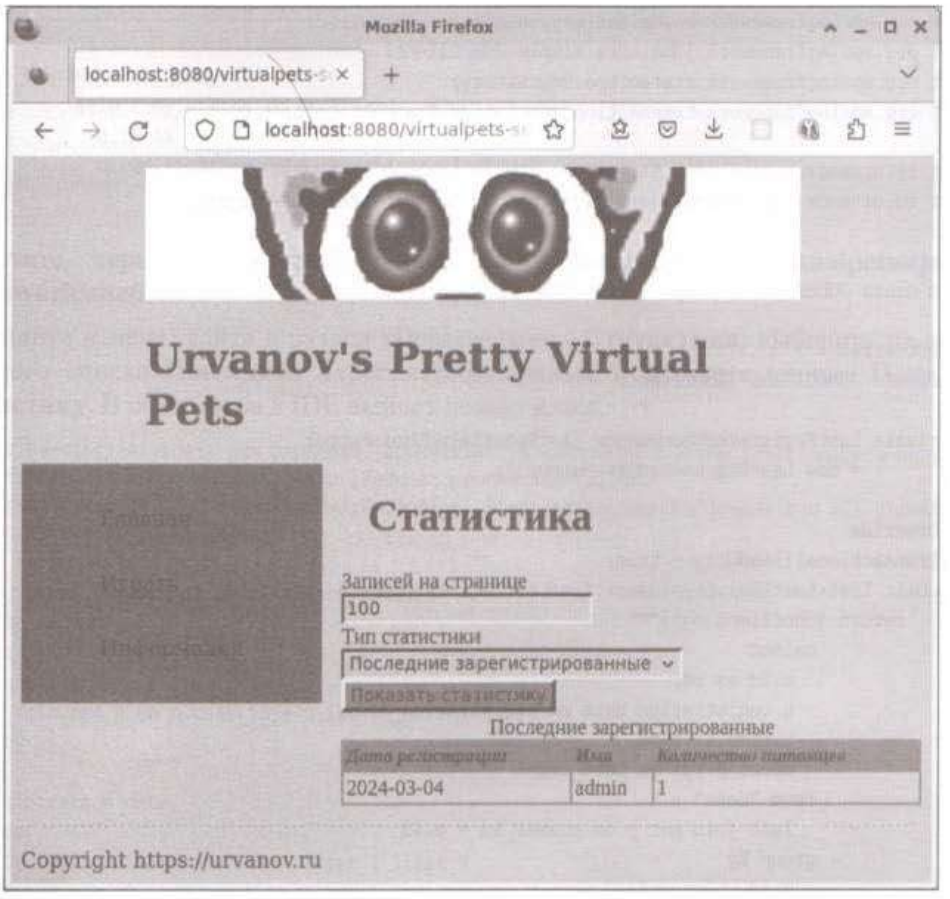


Рис. 7.2. Таблица с результатом работы метода `JdbcReportDaoImpl#findLastRegisteredUsers`

7.3.3. JdbcClient

В Spring Framework 6.1 в дополнение к `JdbcTemplate` и `NamedParameterJdbcTemplate` была добавлена `JdbcClient`.

> ***JdbcClient***
>
> `JdbcClient` доступна и для приложения на Spring Framework без Spring Boot, но без Spring Boot экземпляр `JdbcClient` нужно создавать самостоятельно.

Spring Boot автоматически инициализирует не только бин `JdbcTemplate`, но и бин `JdbcClient` на основе `NamedParameterJdbcTemplate`, поэтому вместо внедрения `JdbcTemplate` можно внедрить экземпляр `JdbcClient` (листинг 7.21).

Листинг 7.21. JdbcReportDaoImpl.java

```
package ru.urvanov.virtualpets.server.dao;

import java.util.List;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.simple.JdbcClient;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import ru.urvanov.virtualpets.server.dao.domain.LastRegisteredUser;
import ru.urvanov.virtualpets.server.dao.mapper.LastRegisteredUserMapper;

@Repository
public class JdbcReportDaoImpl implements JdbcReportDao {

    @Autowired
    private JdbcClient jdbcClient;

    private LastRegisteredUserMapper lastRegisteredUsersMapper
            = new LastRegisteredUserMapper();

    @Override
    @Transactional(readOnly = true)
    public List<LastRegisteredUser> findLastRegisteredUsers(int start, int limit) {
        return jdbcClient.sql("""
                select
                  u.id as id,
                  u.registration_date as registration_date,
                  u.name as name,
                  count(p.id) as pets_count
                from "user" u
                  left join pet p on p.user_id = u.id
                group by
                  u.id,
                  u.registration_date,
                  u.name
                order by registration_date desc offset ? limit ?
                """)
                .param(1, start)
                .param(2, limit)
                .query(lastRegisteredUsersMapper)
                .list();
    }
}
```

7.3.4. Обработка исключений

Все ошибки SQLException, возникающие в `JdbcTemplate`, `NamedParameterJdbcTemplate` и `JdbcClient`, преобразуются в стандартное исключение времени выполнения `org.springframework.dao.DataAccessException`.

Попробуйте для эксперимента «сломать» SQL-запрос внутри `jdbc.query` в примере virtualpets-server-springframework, например, добавив туда случайный набор символов (листинг 7.22).

Листинг 7.22. JdbcReportDaoImpl.java

```
...
return jdbcTemplate.query("""
            select my_string_to_broke_sql
              u.id as id,
              u.registration_date as registration_date,
```

Запустите сервер и откройте адрес **http://localhost:8080/virtualpets-spring-framework/site/home**.

Перейдите в меню сайта в раздел **Информация | Статистика**, выберите из выпадающего списка **Последние зарегистрированные** и нажмите кнопку **Показать статистику**. В окне логов в IDE выйдет исключение:

```
SEVERE: Servlet.service() для сервлета [appServlet] в контексте с путем [/virtualpets-server-
springframework] выбросил исключение [Request processing failed:
org.springframework.jdbc.BadSqlGrammarException: PreparedStatementCallback; bad SQL grammar
[select my_string_to_broke_sql
  u.id as id,
  u.registration_date as registration_date,
  u.name as name,
  count(p.id) as pets_count
from "user" u
  left join pet p on p.user_id = u.id
group by
  u.id,
  u.registration_date,
  u.name
order by registration_date desc offset ? limit ?
]] с первопричиной
org.postgresql.util.PSQLException: ERROR: syntax error at or near "."
  Позиция: 34
      at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2713)
      at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:2401)
      at org.postgresql.core.v3.QueryExecutorImpl.execute(QueryExecutorImpl.java:368)
      at org.postgresql.jdbc.PgStatement.executeInternal(PgStatement.java:498)
      at org.postgresql.jdbc.PgStatement.execute(PgStatement.java:415)
      at org.postgresql.jdbc.PgPreparedStatement.executeWithFlags(PgPreparedStatement.java:190)
```

Основное исключение `org.springframework.jdbc.BadSqlGrammarException` — это дочернее исключение от `org.springframework.dao.DataAccessException`. Иерархия исключений от `DataAccessException` до `BadSqlGrammarException` показана на на рис. 7.3.

При этом исходное исключение `PSQLException` все еще доступно и выводится в лог как первопричина. Исходное исключение, если оно есть, может быть получено с помощью метода `getCause`.

Иерархия `DataAccessException` довольно обширна, и при разработке приложений для определения типа ошибки нужно опираться именно на исключения из этой иерархии.

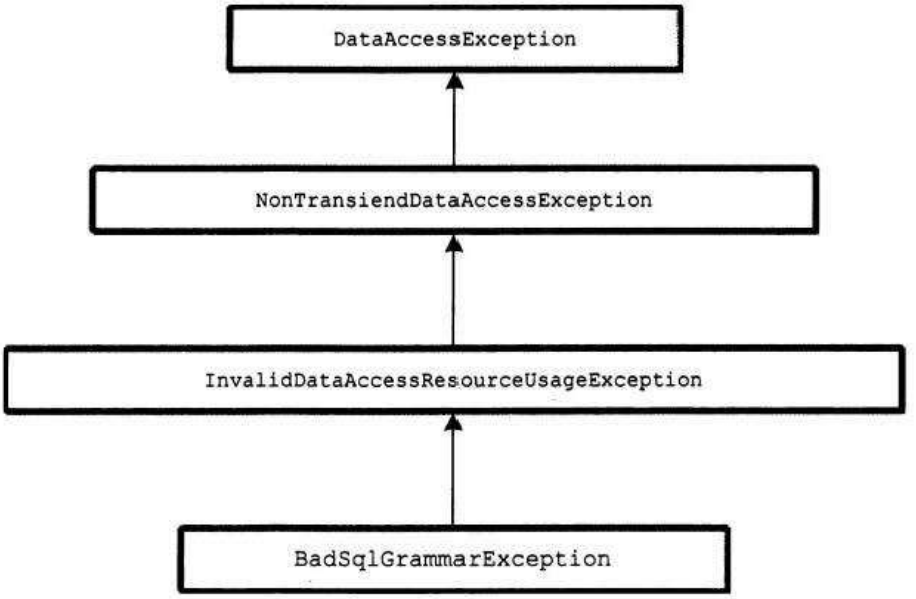


Рис. 7.3. Исключение `BadSqlGrammerException` **и его отношение к исключению** `DataAccessException`

7.4. Спецификация JPA

7.4.1. Введение

JPA — это Jakarta Persistence, ранее называвшаяся Java Persistence API, — спецификация интерфейсов для реализации слоя постоянства, отображения сущностей реляционных баз данных на экземпляры объектов Java (например, таблицы `pet` на класс `Pet`).

Примеры реализации JPA:

- EclipseLink;
- Hibernate;
- OpenJPA;
- DataNucleus.

Hibernate — самая популярная на текущий момент реализация JPA. При работе над коммерческими проектами вы с большой долей вероятности будете работать именно с Hibernate.

До переименования основным пакетом JPA был javax.persistence. В старом коде, использующем Jakarta EE 8 и ниже (Jakarta Persistence 2.2 и ниже), все еще можно встретить применение этого пакета.

После переименования, начиная с Jakarta Persistence 3.0, в Jakarta Persistence основным пакетом стал jakarta.persistence.

Jakarta Persistence — это только спецификация (пример реализации: Hibernate и Spring Data JPA).

В примере сервера виртуальных питомцев virtualpets-server-springframework для создания слоя постоянства используются JPA, Hibernate и Criteria. Проект virtualpets-server-springboot ориентирован на Spring Data JPA.

В этом разделе мы рассмотрим аннотации JPA и их использование совместно с реализацией Hibernate.

7.4.2. Подключение зависимостей

Пример подключения зависимостей для проекта virtualpets-server-springframework на Spring Framework приведен в листинге 7.23.

Листинг 7.23. Файл pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${org.springframework-version}</version>
</dependency>
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${hibernate.version}</version>
</dependency>
```

Здесь `${org.springframework-version}` и `${hibernate.version}` — это версии Spring Framework и Hibernate (листинг 7.24).

Листинг 7.24. Файл pom.xml

```
<properties>
    <org.springframework-version>6.1.10</org.springframework-version>
    <hibernate.version>6.5.2.Final</hibernate.version>
</properties>
```

Пример подключения зависимостей для проекта virtualpets-server-springboot на Spring Boot приведен в листинге 7.25.

Листинг 7.25. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

7.4.3. Настройка для Spring Framework

Конфигурация необходимых бинов virtualpets-server-springframework находится в файле servlet-tx.xml (листинг 7.26).

Листинг 7.26. Файл servlet-tx.xml

```
<bean id = "transactionManager"
    class = "org.springframework.orm.jpa.JpaTransactionManager">
    <property name = "entityManagerFactory" ref = "emf" />
    <property name = "dataSource" ref = "dataSource" />
</bean>

<tx:annotation-driven transaction-manager = "transactionManager" />

<bean id = "emf"
    class = "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name = "dataSource" ref = "dataSource" />
    <property name = "jpaVendorAdapter">
        <bean class = "org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
    </property>
    <property name = "packagesToScan"
        value = "ru.urvanov.virtualpets.server.dao.domain" />
    <property name = "jpaProperties">
        <props>
            <prop key = "hibernate.dialect">
                org.hibernate.dialect.PostgreSQLDialect
            </prop>
            <prop key = "hibernate.globally_quoted_identifiers">
                true
            </prop>
            <prop key = "hibernate.default_schema">
                virtualpets_server_springframework
            </prop>
            <prop key = "hibernate.show_sql">true</prop>
            <prop key = "hibernate.max_fetch_depth">3</prop>
            <prop key = "hibernate.jdbc.fetch_size">50</prop>
            <prop key = "hibernate.jdbc.batch_size">10</prop>
            <prop key = "hibernate.use_nationalized_character_data">
                true
            </prop>
            <prop key = "hibernate.physical_naming_strategy">
                org.hibernate.boot.model.naming.CamelCaseToUnderscoresNamingStrategy
            </prop>
        </props>
    </property>
</bean>
```

Бин `transactionManager` **типа** `JpaTransactionManager` **уже частично описан в** *разд. 7.1* **(см. листинг 7.3).**

Бин `emf` **имеет тип** `LocalContainerEntityManagerFactoryBean` **— это основной класс для настройки JPA. Его главные свойства:**

- `dataSource` **— источник данных. Сервер виртуальных питомцев virtualpets-server-springframework использует источник данных, описанный как ресурс JNDI в контейнере Apache Tomcat;**

- `jpaVendorAdapter` — реализация JPA, сервер виртуальных питомцев virtualpets-server-springframework использует Hibernate, поэтому в этом свойстве указан бин класса `HibernateJpaVendorAdapter`;
- `packagesToScan` — базовые пакеты для сканирования сущностей по аналогии со сканированием бинов Spring. В сервере виртуальных питомцев классы сущностей предметной области находятся в пакете ru.urvanov.virtualpets.server.dao.domain;
- `jpaProperties` — настройки JPA, заполненный экземпляр класса `java.util.Properties`.

  Свойство `jpaProperties` содержит настройки Hibernate, связанные с используемой базой данных, генерацией SQL, размером пакета команд, а также настройки пула соединений и т. д. Пример virtualpets-server-springframework включает следующие настройки:

  - `hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect`

    Hibernate может работать с разными базами данных с различными особенностями и отличающимися синтаксисами SQL, которые необходимо учитывать. С помощью `hibernate.dialect` указывается тип базы данных. Начиная с Hibernate 6.1, тип базы данных определяется из строки подключения JDBC, поэтому эта настройка не обязательна, но вы всегда будете ее видеть в старых проектах;

  - `hibernate.globally_quoted_identifiers = true`

    Значение по умолчанию: `false`. При установке в `true` все идентификаторы обрамляются кавычками — например: `virtualpets-springframework"."pet"` вместо `virtualpets-springframework.pet`;

  - `hibernate.default_schema = virtualpets_server_springframework`

    Схема базы данных, используемая по умолчанию;

  - `hibernate.show_sql = true`

    Значение по умолчанию: `false`. Включает логирование генерируемых SQL-команд в консоль;

  - `hibernate.max_fetch_depth = 3`

    Значение по умолчанию: `0`. Максимальная глубина вложенных `outer join` при выборке подчиненных сущностей;

  - `hibernate.jdbc.fetch_size = 50`

    Значение по умолчанию: `0`. Подсказка JDBC-драйверу, какое количество записей нужно выбирать, когда требуются дополнительные записи;

  - `hibernate.jdbc.batch_size = 10`

    Значение по умолчанию: `0`. SQL-команды будут объединяться в пакеты, каждый из которых будет содержать это количество команд;

- `hibernate.use_nationalized_character_data = true`

  Значение по умолчанию: `false`. Hibernate по умолчанию для типов String и Clob использует типы JDBC Types.VARCHAR и Types.CLOB. При `use_nationalized_character_data = true` вместо них Hibernate будет использовать Types.NVARCHAR и Types.NCLOB;

- `hibernate.physical_naming_strategy = org.hibernate.boot.model.naming.CamelCaseToUnderscoresNamingStrategy`

  Определяет `PhysicalNamingStrategy`, осуществляющий преобразование из логических имен в физические имена таблиц и колонок. Например, преобразование логического имени `machineWithDrinksId` в имя колонки machine_with_drinks_id. Стратегии именования подробно рассматриваются в *разд. 7.4.8*.

7.4.4. Настройка для Spring Boot

Настройки Hibernate для варианта приложения virtualpets-server-springboot содержатся в файле application.yaml (листинг 7.27).

Листинг 7.27. Файл application.yaml

```
spring:
...
  jpa:
    properties:
      hibernate:
        globally_quoted_identifiers: 'true'
        default_schema: virtualpets_server_springboot
        show_sql: 'true'
        max_fetch_depth: 3
        jdbc:
          fetch_size: 50
          batch_size: 10
        use_nationalized_character_data: 'true'
        connection:
          CharSet: utf8
          characterEncoding: utf8
          useUnicode: 'true'
        implicit_naming_strategy: default
```

Свойство `jpaProperties` здесь имеет те же значения, какие приведены в *разд. 7.4.3*.

7.4.5. Сущность JPA

Классы сущностей проекта virtualpets-server-springframework находятся в пакете `ru.urvanov.virtualpets.server.dao.domain`, как это указано в свойстве `packagesToScan`.

> ***Сущность JPA***
>
> Сущность JPA — это любой класс Java, аннотированный в соответствии с правилами Jakarta Persistence, либо специально описанный в XML-файле. Сущности обычно сохраняются в таблице реляционной базы данных и могут иметь связи друг с другом.

Начнем разбор с самой простой сущности `Level` (листинг 7.28).

Листинг 7.28. Level.java

```
package ru.urvanov.virtualpets.server.dao.domain;

import java.io.Serializable;
import java.util.Objects;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

/**
 * Запись из справочника уровней питомца.
 */
@Entity
@Table(name = "level")
public class Level implements Serializable {

    private static final long serialVersionUID = 1477585564717835763L;

    /**
     * Первичный ключ. Новые записи в справочник уровней питомца
     * добавляются только из скриптов liquibase, поэтому первичный ключ
     * не генерируется ни в БД, ни в Java-коде.
     */
    @Id
    private int id;

    /**
     * Количество опыта, которое необходимо набрать питомцу для достижения уровня.
     */
    private int experience;

    /**
     * Конструктор по умолчанию требуется JPA для создания объекта.
     */
    public Level() {

    }

    /**
     * Конструктор для создания экземпляров в самом приложении
     * виртуальных питомцев, например в тестах.
     * @param id {@link #id Первичный ключ}
     * @param experience {@link #experience}.
     */
    public Level(int id, int experience) {
        this.id = id;
        this.experience = experience;
    }
```

```
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getExperience() {
        return experience;
    }

    public void setExperience(int experience) {
        this.experience = experience;
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Level other = (Level) obj;
        return Objects.equals(id, other.id);
    }

    @Override
    public String toString() {
        return "Level [id=" + id + ", experience=" + experience + "]";
    }
}
```

Что находится в этом файле?

Начинается он с объявления пакета `ru.urvanov.virtualpets.server.dao.domain`. **Этот пакет указан в** `packagesToScan` **бина типа** `LocalContainerEntityManagerFactoryBean`, **а значит, класс** `Level` **попадет в сканирование сущностей JPA.**

Затем идут импорты. Обратите внимание на импорт аннотаций из пакета `jakarta.persistence`:

- `Entity` — **классы, помеченные этой аннотацией, становятся сущностями JPA;**
- `Id` — **этой аннотацией помечается поле первичного ключа сущности;**
- `Table` — **служит для обозначения физической таблицы сущности в базе данных.**

Основной и единственный класс файла Level.java — это класс `Level`, который представляет собой объект POJO.

> ***Объект POJO***
>
> POJO, Plain Old Java Object — простой старый Java-объект, т. е. класс, состоящий только из полей, методов установки значений и методов получения значений, не унаследованный от какого-либо специфичного класса и не реализующий какой-либо специфичный интерфейс.

Класс `Level` помечен двумя аннотациями: `@Entity` и `@Table`:

- аннотация `@Entity` указывает, что класс `Level` — это класс слоя постоянства JPA;
- аннотация `@Table` с помощью своего атрибута `name` определяет наименование физической таблицы в базе данных, в которой хранятся строки, отображающиеся на класс `Level`. В нашем случае это таблица `level`, DML-команда создания которой выглядит так:

```
create table level(
                    id INT NOT NULL,
                    experience INT NOT NULL,
                    PRIMARY KEY(id)
);
```

Ее ER-диаграмма представлена на рис. 7.4.

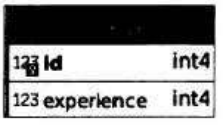


Рис. 7.4. ER-диаграмма таблицы level

В классе `Level` определены два поля, отображающиеся на поля таблицы `level`: `id` и `experience`.

Поле `id` помечено аннотацией `@Id`, поэтому оно будет идентификатором сущности для запросов Hibernate.

Обратите внимание, что аннотация `@Id` указана именно у поля `id`. Если аннотация `@Id` указана для поля класса, как в нашем случае, то Hibernate применяет ее к полям для установки и получения значений.

Всего существуют два способа доступа: доступ к полям и доступ к свойствам:

- если аннотация `@Id` указана на поле класса, то Hibernate использует доступ к полям;
- если аннотация `@Id` указана на методе получения значения, то Hibernate использует доступ к свойствам через методы установки значения и методы получения значения.

Поскольку для сущности `Level` используется доступ к полям, а сама сущность является справочной и заполняется только скриптами DML из Liquibase, то классу `Level` не нужны методы установки значений — Hibernate присвоит нужные значения без

них. Код сервера виртуальных питомцев использует только методы получения значений. Несмотря на это, класс `Level` содержит методы установки значений, т. к. без них создание экземпляров `Level` из кода тестов становится проблематичным (тесты подробно рассмотрены в *главе 14*).

Наименования колонок в таблице `level` совпадают с именами полей класса `Level`.

Особое внимание стоит уделить методам `hashCode` и `equals`. В классе `Level` они переопределены и используют только поле `id`. Конкретно в нашем случае поле `id` однозначно идентифицирует объект. Из программного кода Java новые записи `Level` не создаются, поэтому не будет ситуации, когда `id == null`, а поскольку поле `id` в базе данных является первичным ключом таблицы `level`, то не будет и ситуации, когда эти поля повторяются для разных экземпляров `Level`. Сами методы `hashCode` и `equals` сгенерированы с помощью IDE, и их содержимое вполне стандартно.

7.4.6. Выборка сущности JPA

Выборка данных из базы данных и отображение на описанный в *разд. 7.4.5* класс `Level` происходит в слое доступа к данным DAO (Data Access Layer). Пример выборки класса `Level` находится в файле **LevelDaoImpl.java** проекта **virtualpets-server-springframework** (листинг 7.29).

Листинг 7.29. Файл LevelDaoImpl.java

```
package ru.urvanov.virtualpets.server.dao;

import java.util.List;
import java.util.Optional;

import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.TypedQuery;
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import ru.urvanov.virtualpets.server.dao.domain.Level;

@Repository("levelDao")
public class LevelDaoImpl implements LevelDao {

    @PersistenceContext
    private EntityManager em;

    @Transactional(readOnly = true)
    @Override
    public Optional<Level> findById(Integer id) {
        Level level = em.find(Level.class, id);
        return Optional.ofNullable(level);
    }
```

```
    @Transactional(readOnly = true)
    @Override
    public List<Level> findAll() {
        CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
        CriteriaQuery<Level> criteriaQuery = criteriaBuilder.createQuery(Level.class);
        criteriaQuery.from(Level.class);
        TypedQuery<Level> typedQuery = em.createQuery(criteriaQuery);
        return typedQuery.getResultList();
    }
}
```

Обратите здесь внимание на строку в методе `findById`:

```
Level level = em.find(Level.class, id);
```

Метод `find` интерфейса `EntityManager` позволяет выбрать сущность по идентификатору. Первым параметром метода `find` передается класс сущности, которую необходимо выбрать, вторым параметром — идентификатор сущности.

Интерфейс `EntityManager` описывает основные методы работы с сущностями. Помимо метода `find` он содержит методы сохранения и удаления сущностей, а также методы выборки сущностей по различным критериям (работа с интерфейсом `EntityManager` описана в *разд. 7.5*).

Hibernate как поставщик JPA сгенерирует необходимые SQL-скрипты выборки и преобразует результат в экземпляр класса `Level`.

7.4.7. Более сложная сущность JPA

Рассмотрим более сложный вариант сущности JPA (листинг 7.30).

Листинг 7.30. Food.java

```
package ru.urvanov.virtualpets.server.dao.domain;

import java.io.Serializable;
import java.util.Objects;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

/**
 * Запись из справочника еды.
 */
@Entity
@Table(name = "food")
public class Food implements Serializable {

    private static final long serialVersionUID = 8791181701061581183L;
```

```
    /**
     * Первичный ключ. Новые записи в справочник еды добавляются
     * только скриптами liquibase, поэтому первичный ключ
     * не генерируется ни в БД, ни в Java-коде.
     */
    @Id
    @Enumerated(EnumType.STRING)
    private FoodId id;

    @Column(name = "refrigeratorId")
    private int refrigeratorLevel;

    private int refrigeratorOrder;

    private float hiddenObjectsGameDropRate;

    /**
     * Конструктор по умолчанию требуется JPA для создания объекта.
     */
    public Food() {
        super();
    }

    /**
     * Конструктор для создания экземпляров в самом приложении
     * виртуальных питомцев, например в тестах.
     * @param id {@link #id}
     * @param refrigeratorLevel {@link #refrigeratorLevel}
     * @param refrigeratorOrder {@link #refrigeratorOrder}
     * @param hiddenObjectsGameDropRate {@link
     * #hiddenObjectsGameDropRate}
     */
    public Food(FoodId id, int refrigeratorLevel, int refrigeratorOrder,
            float hiddenObjectsGameDropRate) {
        super();
        this.id = id;
        this.refrigeratorLevel = refrigeratorLevel;
        this.refrigeratorOrder = refrigeratorOrder;
        this.hiddenObjectsGameDropRate = hiddenObjectsGameDropRate;
    }

    public FoodId getId() {
        return id;
    }

    public int getRefrigeratorLevel() {
        return refrigeratorLevel;
    }

    public int getRefrigeratorOrder() {
        return refrigeratorOrder;
    }
```

```
    public float getHiddenObjectsGameDropRate() {
        return hiddenObjectsGameDropRate;
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Food other = (Food) obj;
        return id == other.id;
    }

    @Override
    public String toString() {
        return "Food [id=" + id
                + ", refrigeratorLevel=" + refrigeratorLevel
                + ", refrigeratorOrder=" + refrigeratorOrder
                + ", hiddenObjectsGameDropRate="
                + hiddenObjectsGameDropRate
                + "]";
    }
}
```

DML-команда создания таблицы food:

```
create table food(
                    id varchar(50) NOT NULL,
                    refrigerator_id INT NOT NULL,
                    refrigerator_order INT NOT NULL,
                    hidden_objects_game_drop_rate REAL NOT NULL,
                    PRIMARY KEY(id)
);
```

ER-диаграмма таблицы food представлена на рис. 7.5.

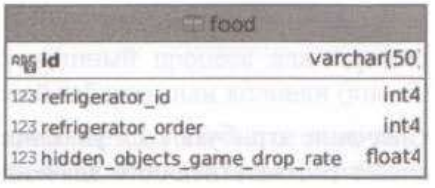


Рис. 7.5. ER-диаграмма таблицы food

Класс Food по сложности лишь немного превосходит класс Level. В Food используются те же самые аннотации @Entity, @Table и @Id, что и в Level.

Но есть и два действительно существенных различия:

- первичный ключ в классе Food отображается не на строку, как это было в Level, а на перечисление FoodId, о чем свидетельствует аннотация @Enumerated;
- для поля refrigeratorLevel используется аннотация @Column, в атрибуте name которой указано логическое имя refrigeratorId, т. к. имя поля не совпадает с именем колонки в таблице.

Аннотация @Enumerated применяется для отображения на перечисление не только первичных ключей — ее можно задействовать для любых полей класса, связанных с колонками в таблице.

В единственном атрибуте value аннотации @Enumerated указывается, каким образом перечисление отображается на значение в колонке таблицы:

- EnumType.ORDINAL — поле сохраняется в базе данных в колонку с числовым типом;
- EnumType.STRING — поле сохраняется в базе данных в колонку со строковым типом, содержащим текстовое значение перечисления.

Перечисление FoodId содержит возможные значения ключей напитков (листинг 7.31).

Листинг 7.31. FoodId.java

```
package ru.urvanov.virtualpets.server.dao.domain;

/**
 * Код еды.
 */
public enum FoodId {
    CARROT,
    DRY_FOOD,
    FISH,
    ICE_CREAM,
    APPLE,
    CABBAGE,
    CHOCOLATE,
    FRENCH_FRIES,
    JAPANESE_ROLLS,
    PIE,
    POTATOES,
    SANDWICH,
    BANANA,
    WATERMELON
}
```

Для аннотации @Enumerated значение атрибута value указано как EnumType.STRING, поэтому, например, для FoodId.CARROT соответствующим значением колонки id в таблице food базы данных будет строка FOOD.

Если вместо `EnumType.STRING` в значение атрибута `value` аннотации `@Enumerated` передавать значение `EnumType.ORDINAL`, то значения перечисления будут отображаться в базе данных на числа 0, 1, 2 ... и т. д. в соответствии со своим порядком в перечислении, возвращаемом методом `ordinal()`.

Элементы этого перечисления облегчат работу со справочником еды из Java-кода сервера виртуальных питомцев. В реальных приложениях систем корпоративного учета вряд ли встретятся ситуации, когда перечисление можно использовать в качестве первичного ключа. Скорее всего, первичный ключ в большинстве случаев станет генерироваться последовательностью в базе данных. Либо первичный ключ будет числовым или строковым, но с достаточно большим количеством значений, который в коде на Java окажется проще отобразить на строку, чем на перечисление.

Остальные поля сущности `Food` отображаются на примитивные типы `int` и `float`. В нашем случае классы-обертки `Integer` и `Float` не используются, т. к. соответствующие поля в базе данных не могут принимать NULL-значения.

Почему выбран именно тип `float`? Согласно рекомендациям по разработке на Java при работе с вещественными числами в первую очередь стоит рассматривать тип `double`. В нашем случае тоже можно было задействовать `double`, но в базе данных выбран тип колонки `real`. Тип данных `real` в PostgreSQL занимает четыре байта, как и тип `float` в Java, поэтому разумнее его отобразить на `float`, чем на `double`. Для поля `hiddenObjectsGameDropRate` точность значения не сильно важна — в нем будут значения наподобие 0,1 или 1,3 или любые другие вещественные числа для вычисления вероятности выпадения при игре в поиск скрытых предметов.

> ***Типы данных float и double — не для хранения денежных сумм***
>
> Имейте в виду, что типы `float` и `double` в Java, так же как и типы `real` и `double precision` в PostgreSQL, не подходят для хранения и обработки суммы денег, т. к. не обеспечивают точных вычислений. Для хранения и обработки денежных сумм необходимо использовать `java.math.BigDecimal` в Java и тип `numeric` в PostgreSQL.

Методы `hashCode` и `equals` класса `Food` используют только поле `id`, т. к. это поле всегда однозначно идентифицирует сущность `Food`, при этом оно не может быть null, т. к. экземпляры сущности не создаются из Java-кода, а таблица `food` используется только как справочник, заполняемый DML-скриптами из Liquibase.

7.4.8. Стратегия именования

В *разд. 7.4.7* при описании сущности `Food` использовалась аннотация `@Column`. Как уже было сказано ранее, в атрибуте `name` этой аннотации указывается логическое имя сущности, если оно не совпадает с именем поля.

Hibernate использует двухэтапный процесс для определения логического имени объекта и имени физической таблицы или колонки (рис. 7.6):

1. На начальном этапе определяется логическое имя, которое может быть явно указано в атрибуте name аннотаций `@Column` и `@Table` либо определено самим Hibernate с помощью стратегии `ImplicitNamingStrategy`.

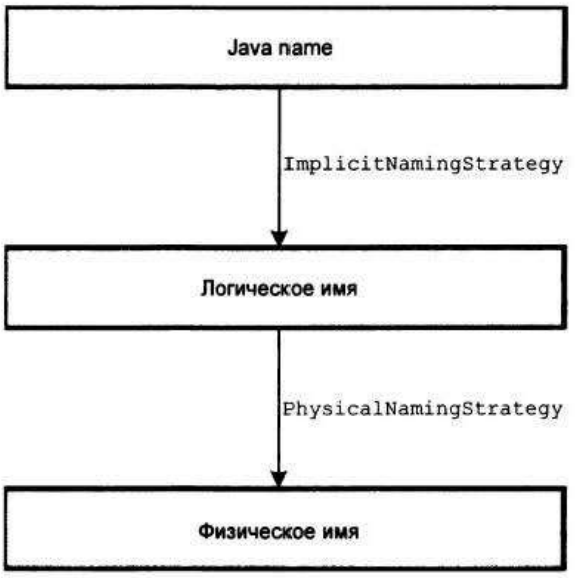


Рис. 7.6. Преобразование имени от Java объекта/поля до физического имени таблицы/колонки

2. На втором этапе определяется физическое имя в базе данных на основе стратегии `PhysicalNamingStrategy`.

> ***Разделение имен на логическое и физическое — только для Hibernate***
>
> Спецификация Jakarta Persistence не разделяет имена на логическое и физическое. Согласно этой спецификации логическое имя — это и есть физическое имя, стало быть, для Jakarta Persistence имя, указанное в `@Column`, будет окончательным именем таблицы или колонки в базе данных.

Стратегия `ImplicitNamingStrategy` задействуется, если явно не определено логическое имя сущности через `@Table` или логическое имя атрибута через `@Column`. В этом случае необходимо неявно вычислить это имя на основе имени класса либо имени поля. По умолчанию Hibernate использует стратегию `ImplicitNamingStrategyJpaCompliantImpl`, которая формирует логическое имя, идентичное имени класса Java или поля.

Стратегия `PhysicalNamingStrategy` преобразует логическое имя в соответствующее физическое имя таблицы или поля.

Spring Boot для `PhysicalNamingStrategy` определяет свою стратегию — `CamelCaseToUnderscoresNamingStrategy`, в которой все точки заменяются на подчеркивания, а camelCase («верблюжья нотация», принятая в Java) заменяется на нотацию с подчеркиванием между слов. В дополнение все имена таблиц генерируются в нижнем регистре.

Например, по `CamelCaseToUnderscoresNamingStrategy` логическое имя сущности `MachineWithDrinks` будет заменено на `machine_with_drinks`.

Для Spring Framework без Spring Boot по умолчанию в качестве физического имени используется логическое имя, что в случае сервера виртуальных питомцев не то,

что нужно, поэтому проект virtualpets-server-springframework настраивает CamelCaseToUnderscoresNamingStrategy самостоятельно, устанавливая значение настройки hibernate.physical_naming_strategy свойства jpaProperties для бина emf в org.hibernate.boot.model.naming.CamelCaseToUnderscoresNamingStrategy (листинг 7.32).

Листинг 7.32. Файл servlet-tx.xml

```
<property name="jpaProperties">
    <props>

        <prop key="hibernate.physical_naming_strategy">
org.hibernate.boot.model.naming.CamelCaseToUnderscoresNamingStrategy
        </prop>
...
    </props>
</property>
```

Для сущности Food после преобразования логические имена большинства свойств сущности будут совпадать с именами полей класса Food, т. к. ImplicitNamingStrategyJpaCompliantImpl формирует имя, идентичное имени поля класса.

Единственное отличающееся от этого алгоритма поле — refrigeratorLevel, для которого с помощью аннотации @Column указано логическое имя refrigeratorId.

Физические имена колонок в базе данных, формируемые CamelCaseToUnderscoresNamingStrategy, приведены в табл. 7.1.

***Таблица 7.1.** Преобразования имен полей класса Food на имена колонок физической таблицы food*

Имя поля Java	Логическое имя	Физическое имя
id	id	id
refrigeratorLevel	refrigeratorId	refrigerator_id
refrigeratorOrder	refrigeratorOrder	refrigerator_order
hiddenObjectsGameDropRate	hiddenObjectsGameDropRate	hidden_objects_game_drop_rate

В атрибуте name аннотации @Column поля refrigeratorLevel можно было указать сразу refrigerator_id — тогда логическое и физическое имена для поля refrigeratorLevel совпадали бы.

7.4.9. Связи 1:M и M:1

К настоящему времени все рассматриваемые сущности не имели связей друг с другом. Поскольку основные моменты создания одиночных сущностей уже рассмотрены, перейдем теперь к более сложному варианту (листинг 7.33).

Листинг 7.33. Refrigerator.java

```
package ru.urvanov.virtualpets.server.dao.domain;

import java.io.Serializable;
import java.util.Map;
import java.util.Objects;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Id;
import jakarta.persistence.MapKeyColumn;
import jakarta.persistence.MapKeyEnumerated;
import jakarta.persistence.OneToMany;
import jakarta.persistence.Table;

/**
 * Запись справочника холодильников.
 */
@Entity
@Table(name = "refrigerator")
public class Refrigerator implements Serializable {

    private static final long serialVersionUID = -6335332962524762996L;

    /**
     * Первичный ключ. Новые записи в справочник холодильников
     * добавляются только скриптами liqubiase, поэтому первичный ключ
     * не генерируется ни в БД, ни в Java-коде.
     */
    @Id
    private int id;

    /**
     * Количество получаемого при постройке/улучшении опыта
     */
    private int experience;

    /**
     * Количество ресурсов, необходимое для строительства/улучшения холодильника.
     */
    @OneToMany(mappedBy = "refrigerator",
            cascade = CascadeType.ALL, orphanRemoval = true)
    @MapKeyEnumerated(EnumType.STRING)
    @MapKeyColumn(name = "building_material_id")
    private Map<BuildingMaterialId, RefrigeratorCost> refrigeratorCosts;

    public int getId() {
        return id;
    }
```

```
    public void setId(int id) {
        this.id = id;
    }

    public int getExperience() {
        return experience;
    }

    public void setExperience(int experience) {
        this.experience = experience;
    }

    public Map<BuildingMaterialId, RefrigeratorCost>
            getRefrigeratorCosts() {
        return refrigeratorCosts;
    }

    public void setRefrigeratorCosts(Map<BuildingMaterialId,
            RefrigeratorCost> refrigeratorCosts) {
        this.refrigeratorCosts = refrigeratorCosts;
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Refrigerator other = (Refrigerator) obj;
        return Objects.equals(id, other.id);
    }

    @Override
    public String toString() {
        return "Refrigerator [id=" + id + "]";
    }
}
```

Класс `Refrigerator`, отображающийся на таблицу `refrigerator` в базе данных, похож на классы `Level` и `Food`, рассмотренные до этого. Единственное различие — в поле `refrigeratorCost` типа `java.util.Map`, на которое отображается связь 1:M с таблицей `refrigerator_cost`.

ER-диаграмма (она же диаграмма «сущность-связь») этих таблиц показана на рис. 7.7.

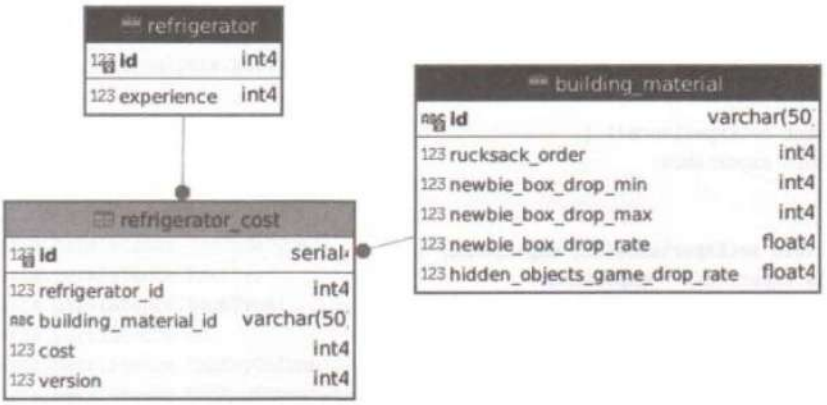


Рис. 7.7. ER-диаграмма таблицы `refrigerator_cost`

Перед разбором аннотаций над полем `refrigeratorCost` **сначала необходимо разобрать сущность** `RefrigeratorCost`**, отображающуюся на таблицу** `refrigerator_cost` **соответственно.**

В качестве значений поля `refrigeratorCost` **в** `java.util.Map` **используется сущность** `RefrigeratorCost` **(листинг 7.34).**

Листинг 7.34. RefrigeratorCost.java

```
package ru.urvanov.virtualpets.server.dao.domain;

import java.util.Objects;

import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;

/**
 * Запись о количестве строительного материала, необходимого
 * для постройки/улучшения холодильника.
 */
@Entity
@Table(name="refrigerator_cost")
public class RefrigeratorCost {

    /**
     * Первичный ключ.
     */
    @Id
    private int id;

    @ManyToOne(fetch = FetchType.LAZY)
    private Refrigerator refrigerator;
```

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "building_material_id")
private BuildingMaterial buildingMaterial;

private int cost;

/**
 * Конструктор без параметров, необходимый JPA.
 */
public RefrigeratorCost() {
    super();
}

/**
 * Конструктор с параметрами, используемый приложением, например  тестами.
 * @param id {@link #id Первичный ключ}
 * @param refrigerator {@link #refrigerator}
 * @param buildingMaterial {@link #buildingMaterial}
 * @param cost {@link #cost Необходимое количество материала}
 */
public RefrigeratorCost(int id, Refrigerator refrigerator,
        BuildingMaterial buildingMaterial, int cost) {
    super();
    this.id = id;
    this.refrigerator = refrigerator;
    this.buildingMaterial = buildingMaterial;
    this.cost = cost;
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public Refrigerator getRefrigerator() {
    return refrigerator;
}

public void setRefrigerator(Refrigerator refrigerator) {
    this.refrigerator = refrigerator;
}

public BuildingMaterial getBuildingMaterial() {
    return buildingMaterial;
}

public void setBuildingMaterial(BuildingMaterial buildingMaterial) {
    this.buildingMaterial = buildingMaterial;
}
```

```
    public int getCost() {
        return cost;
    }

    public void setCost(int cost) {
        this.cost = cost;
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        RefrigeratorCost other = (RefrigeratorCost) obj;
        return id == other.id;
    }

    @Override
    public String toString() {
        return "RefrigeratorCost [id=" + id
                + ", refrigerator=" + refrigerator
                + ", buildingMaterial=" + buildingMaterial
                + ", cost=" + cost
                + "]";
    }
}
```

Методы получения значений, `hashCode`, `equals`, первичного ключа — всё это в `RefrigeratorCost` выглядит аналогично примерам, рассмотренным ранее.

Новые аннотации в `RefrigeratorCost`:

- `@ManyToOne` — используется для описания связи M:1;
- `@JoinColumn` — необязательная аннотация, в которой указывается физическое имя колонки в таблице, содержащей вторичные ключи связи.

Аннотация `@JoinColumn`, как уже сказано, не обязательна, и если ее не использовать, то имя колонки генерируется на основе конкатенации имени свойства, символа нижнего подчеркивания и имени колонки первичного ключа. Например, для поля `refrigerator` из приведенного примера имя колонки будет `refrigerator_id`, где `refrigerator` — имя поля с аннотацией `@ManyToOne`, а `id` — имя поля с первичным ключом.

Если имя колонки с вторичным ключом отличается от генерируемого по умолчанию — например, при наличии нескольких разных колонок, связанных связью M:1 с одинаковой таблицей, то имя колонки вторичного ключа указывается с помощью аннотации `@JoinColumn`, как это сделано для поля `buildingMaterial` (листинг 7.35).

Листинг 7.35. Пример использования @JoinColumn

```
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "building_material_id")
private BuildingMaterial buildingMaterial;
```

В нашем случае аннотация `@JoinColumn` не нужна — имя колонки с вторичным ключом, сгенерированное по умолчанию, было бы точно таким же (`building_material_id`), а пример приведен только для того, чтобы показать такую возможность.

Аналогичным образом можно было бы указать имя колонки вторичного ключа и для `refrigerator`, но, как и в случае с `buildingMaterial`, в этом нет необходимости (листинг 7.36).

Листинг 7.36. Пример использования @JoinColumn

```
@ManyToOne(fetch = FetchType.LAZY)
private Refrigerator refrigerator;
```

Обратите внимание на атрибут `fetch = FetchType.LAZY` аннотации `@ManyToOne`. Атрибут `fetch` управляет стратегией выборки данных связанной сущности. Он может принимать два значения:

- `FetchType.LAZY` — ленивое заполнение данными. Связанная сущность заполняется только по требованию;
- `FetchType.EAGER` — связанная сущность всегда заполняется данными.

По умолчанию атрибут `fetch` аннотации `@ManyToOne` принимает значение `FetchType.EAGER`, что может привести к лишним SQL-командам выборки данных либо к дополнительным объединениям таблиц и лишним заполняемым данным в SQL-запросах, генерируемых Hibernate.

При проектировании слоя предметной области важно правильно выбирать стратегию выборки данных:

- с точки зрения легкости использования классов лучше во всех случаях использовать `FetchType.EAGER`, т. к. при этой стратегии выборки данные в классах всегда будут доступны для использования;
- с точки зрения производительности во всех случаях необходимо применять `FetchType.LAZY`, а необходимые связанные сущности и коллекции определять при выборке данных с помощью указания `fetch join`. В противном случае связанные сущности и коллекции будут заполняться сами при первом обращении к их свойствам, генерируя при этом дополнительные SQL-запросы.

В `RefrigeratorCost` для всех аннотаций `@ManyToOne` выбрана стратегия выборки `FetchType.LAZY`.

Следующая сущность, использующаяся в классе `RefrigeratorCost` и в поле `refrigeratorCost` класса `Refrigerator`, — `BuildingMaterial` (листинг 7.37).

Листинг 7.37. BuildingMaterial.java

```
package ru.urvanov.virtualpets.server.dao.domain;

import java.io.Serializable;
import java.util.Objects;

import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

/**
 * Запись из справочника материалов для строительства.
 */
@Entity
@Table(name = "building_material")
public class BuildingMaterial implements Serializable {

    private static final long serialVersionUID = -6611026384958159106L;

    /**
     * Первичный ключ. Новые записи в справочник материалов
     * для строительства добавляются скриптами liquibase,
     * первичный ключ не генерируется ни в БД, ни в Java-коде.
     */
    @Id
    @Enumerated(EnumType.STRING)
    private BuildingMaterialId id;

    private int rucksackOrder;

    private int newbieBoxDropMin;

    private int newbieBoxDropMax;

    private float newbieBoxDropRate;

    private float hiddenObjectsGameDropRate;

    /**
     * Конструктор по умолчанию, необходимый JPA
     */
    public BuildingMaterial() {
        super();
    }
```

```
/**
 * Конструктор, используемый приложением, например тестами.
 * @param id {@link #id}
 */
public BuildingMaterial(BuildingMaterialId id) {
    super();
    this.id = id;
}

public BuildingMaterialId getId() {
    return id;
}

public int getRucksackOrder() {
    return rucksackOrder;
}

public int getNewbieBoxDropMin() {
    return newbieBoxDropMin;
}

public int getNewbieBoxDropMax() {
    return newbieBoxDropMax;
}

public float getNewbieBoxDropRate() {
    return newbieBoxDropRate;
}

public float getHiddenObjectsGameDropRate() {
    return hiddenObjectsGameDropRate;
}

@Override
public int hashCode() {
    return Objects.hash(id);
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    BuildingMaterial other = (BuildingMaterial) obj;
    return id == other.id;
}
```

```
    @Override
    public String toString() {
        return "BuildingMaterial [id=" + id
                + ", rucksackOrder=" + rucksackOrder
                + ", newbieBoxDropMin=" + newbieBoxDropMin
                + ", newbieBoxDropMax=" + newbieBoxDropMax
                + ", newbieBoxDropRate=" + newbieBoxDropRate
                + ", hiddenObjectsGameDropRate=" + hiddenObjectsGameDropRate
                + "]";
    }
}
```

Вот DML-команда создания таблицы `building_material`, соответствующей сущности `BuildingMaterial`:

```
create table building_material(
    id varchar(50) NOT NULL,
    rucksack_order INT NOT NULL,
    newbie_box_drop_min INT NOT NULL,
    newbie_box_drop_max INT NOT NULL,
    newbie_box_drop_rate REAL NOT NULL,
    hidden_objects_game_drop_rate REAL NOT NULL,
    PRIMARY KEY (id)
);
```

ER-диаграмма этой таблицы приведена на рис. 7.8.

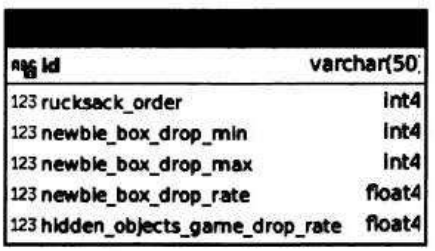


Рис. 7.8. ER-диаграмма таблицы `building_material`

Таким образом, все аннотации, используемые в классе `BuildingMaterial`, уже рассмотрены.

Перечисление `BuildingMaterialId` содержит возможные значения ключей строительных материалов (листинг 7.38).

Листинг 7.38. BuildingMaterialId.java

```
package ru.urvanov.virtualpets.server.dao.domain;

/**
 * Коды материалов для строительства.
 */
public enum BuildingMaterialId {
    TIMBER,
```

```
    BOARD,
    STONE,
    CHIP,
    WIRE,
    IRON,
    OIL,
    BLUE_CRYSTAL,
    RUBBER
```

Обратимся еще раз к сущности `Refrigerator` — в частности, к ее свойству `refrigeratorCost` (листинг 7.39).

Листинг 7.39. Refrigerator.java

```
...
/**
 * Количество ресурсов, необходимое для строительства/улучшения холодильника.
 */
@OneToMany(mappedBy = "refrigerator", cascade = CascadeType.ALL, orphanRemoval = true)
@MapKeyEnumerated(EnumType.STRING)
@MapKeyColumn(name = "building_material_id")
private Map<BuildingMaterialId, RefrigeratorCost> refrigeratorCost;
...
```

Аннотация `@OneToMany` используется для связи 1:M. В ней указано неколько атрибутов:

- в атрибуте `mappedBy` определяется имя поля из связанной сущности, для которого указана аннотация `@ManyToOne`. В нашем случае это поле `refrigerator`, ссылающееся на класс `Refrigerator`. Атрибут `mappedBy` используется для двусторонних связей, как в случае с `refrigeratorCost`, когда `RefrigeratorCost` ссылается на `Refrigerator`, а сам `Refrigerator` содержит коллекцию связанных `RefrigeratorCost`, при этом атрибут `mappedBy` указывается для главной сущности, которая владеет связью между этими сущностями;
- в атрибуте `cascade` определяется, какие операции необходимо распространять на связанные сущности из коллекции. Значение `CascadeType.ALL` указывает, что на сущности из коллекции будут распространяться все операции. В нашем случае коллекция представляет собой экземпляр Map, и операции будут распространяться на значения из пар «ключ-значение» `refrigeratorCost`. По умолчанию же на связанные сущности из коллекции не распространяются никакие операции;
- атрибут `orphanRemoval`, выставленный в `true`, указывает, что элементы, удаленные из `refrigeratorCost`, следует удалить из базы данных.

В аннотации `@OneToMany` присутствует также атрибут `fetch`, как и в `@ManyToOne`. В отличие от `@ManyToOne`, для `@OneToMany` значение атрибута `fetch` по умолчанию равно `FetchType.LAZY`. Указывать `fetch` для `@OneToMany` имеет смысл, только если вы хотите вместо ленивой выборки данных использовать `FetchType.EAGER`.

Атрибут `name` в аннотации `@MapKeyColumn` указывает на имя колонки, в которой находятся ключи для экземпляра Map. Для `refrigeratorCost` ключи экземпляра Map берутся из колонки `building_material_id` таблицы `refrigerator_cost`, на которую отображается сущность `RefrigeratorCost`.

Аннотация `@MapKeyEnumerated(EnumType.STRING)` работает аналогично аннотации `@Enumerated(EnumType.STRING)`, но для ключей экземпляра Map. Для `refrigeratorCost` ключи типа `BuildingMaterialId` будут отображаться на строковые значения в базе данных.

Используемый в приложении сервера виртуальных питомцев способ отображения таблицы `refrigerator_cost` на классы Java — не единственный. Вместо него можно придумать другие способы, например:

- не описывать в классе `Refrigerator` связь с `RefrigeratorCost` — в этом случае связь между `RefrigeratorCost` и `Refrigerator` была бы односторонней;
- отобразить связь класса `Refrigerator` с `RefrigeratorCost` с помощью `Set` или `List`;
- в классе `RefrigeratorCost` не описывать связь `Refrigerator`, а вместо этого отобразить колонку `refrigerator_id` таблицы `refrigerator_cost` на обычное поле типа `int`.

В любом случае при описании предметной области стоит подумать над тем, как будет использоваться сущность, какой способ отображения будет удобнее, какой способ отображения позволит повысить производительность и т. п.

7.4.10. Сущность *Pet*

Самый перегруженный аннотациями и связями класс из предметной области в сервисе виртуальных питомцев — это класс `Pet` (питомец).

Класс `Pet` имеет определенные различия в virtualpets-server-springboot и virtualpets-server-springframework. Вариант из virtualpets-server-springframework показан в листинге 7.40.

Листинг 7.40. Pet.java

```
package ru.urvanov.virtualpets.server.dao.domain;

import java.io.Serializable;
import java.time.OffsetDateTime;
import java.util.Map;
import java.util.Objects;
import java.util.Set;

import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
```

```
import jakarta.persistence.JoinColumn;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.MapKeyColumn;
import jakarta.persistence.MapKeyEnumerated;
import jakarta.persistence.NamedAttributeNode;
import jakarta.persistence.NamedEntityGraph;
import jakarta.persistence.NamedQuery;
import jakarta.persistence.NamedSubgraph;
import jakarta.persistence.OneToMany;
import jakarta.persistence.SequenceGenerator;
import jakarta.persistence.Table;
import jakarta.persistence.Version;
import jakarta.validation.constraints.Size;

/**
 * Питомец.
 */
@Entity
@Table(name = "pet")
... именованные запросы и графы сущностей
public class Pet implements Serializable {

    private static final long serialVersionUID = 2699175148933987413L;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "pet_seq")
    @SequenceGenerator(name = "pet_seq", sequenceName = "pet_id_seq", allocationSize = 1)
    private Integer id;

    @Size(max = 50)
    private String name;

    private String sessionKey;

    private OffsetDateTime createdDate;

    private OffsetDateTime loginDate;

    private int satiety;

    private int mood;

    private int education;

    private int drink;

    @Size(max = 50)
    private String comment;
```

```
@ManyToOne(fetch = FetchType.LAZY)
private User user;

@Enumerated
private PetType petType;

@ManyToOne(fetch = FetchType.LAZY)
private Cloth hat;

@ManyToOne(fetch = FetchType.LAZY)
private Cloth cloth;

@ManyToOne(fetch = FetchType.LAZY)
private Cloth bow;

@ManyToOne(fetch = FetchType.LAZY)
private Level level;

private int experience = 0;

private int eatCount = 0;

private int drinkCount = 0;

private int teachCount = 0;

private int buildCount = 0;

private int hiddenObjectsGameCount;

private OffsetDateTime everyDayLoginLast;

private int everyDayLoginCount;

@Version
private int version;

@OneToMany(mappedBy = "pet", cascade = CascadeType.ALL, orphanRemoval = true)
@MapKeyEnumerated(EnumType.STRING)
@MapKeyColumn(name = "food_id")
private Map<FoodId, PetFood> foods;

@ManyToMany
@JoinTable(name = "pet_cloth",
        joinColumns = @JoinColumn(name = "pet_id"),
        inverseJoinColumns = @JoinColumn(name = "cloth_id"))
private Set<Cloth> cloths;

@OneToMany(mappedBy = "pet", cascade = CascadeType.ALL,  orphanRemoval = true)
@MapKeyEnumerated(EnumType.STRING)
@MapKeyColumn(name = "building_material_id")
private Map<BuildingMaterialId, PetBuildingMaterial> buildingMaterials;
```

```
    @ManyToMany
    @JoinTable(
            inverseJoinColumns = @JoinColumn(name = "book_id"))
    private Set<Book> books;

    @OneToMany(mappedBy= "pet", cascade = CascadeType.ALL,  orphanRemoval = true)
    @MapKeyEnumerated(EnumType.STRING)
    @MapKeyColumn(name = "drink_id")
    private Map<DrinkId, PetDrink> drinks;

    @OneToMany(mappedBy = "pet", cascade = CascadeType.ALL, orphanRemoval = true)
    @MapKeyEnumerated(EnumType.STRING)
    @MapKeyColumn(name = "journal_entry_id")
    private Map<JournalEntryId, PetJournalEntry> journalEntries;

    @OneToMany(mappedBy = "pet", cascade = CascadeType.ALL, orphanRemoval = true)
    @MapKeyEnumerated(EnumType.STRING)
    @MapKeyColumn(name = "achievement_id")
    private Map<AchievementId, PetAchievement> achievements;

//... методы установки и получения значения

    @Override
    public int hashCode() {
        return Objects.hash(createdDate, name, petType, user.getId());
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Pet other = (Pet) obj;
        return Objects.equals(createdDate, other.createdDate)
                && Objects.equals(name, other.name) && petType == other.petType
                && Objects.equals(user.getId(), other.user.getId());
    }

    @Override
    public String toString() {
        return "Pet [id=" + id + ", name=" + name + ", createdDate="
                + createdDate + ", comment=" + comment
                + ", user.id=" + user.getId()
                + ", petType=" + petType + "]";
    }
}
```

Аннотации `@Entity` и `@Table`, упомянутые в этом коде, уже были описаны ранее.

7.4.11. Генерация первичного ключа

В отличие от рассмотренных до этого сущностей, первичный ключ сущности `Pet` генерируется последовательностью PostgreSQL (листинг 7.41).

Листинг 7.41. Pet.java

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "pet_seq")
@SequenceGenerator(name = "pet_seq", sequenceName = "pet_id_seq", allocationSize = 1)
private Integer id;
```

В качестве типа поля `id` класс-обертка `Integer` выбран не случайно. Питомцы создаются самим сервером виртуальных питомцев. Первичный ключ в таблице `pet`, в которой хранятся созданные питомцы, генерируется последовательностью `pet_id_seq`. При создании нового питомца и заполнении полей данными у сущности питомца нет первичного ключа до момента сохранения питомца в базу данных, поэтому в поле `id` всё это время хранится значение `null`.

Аннотация `@GeneratedValue` указывает, что значения для поля генерируются. В атрибуте `strategy` аннотации `@GeneratedValue` указывается стратегия генерации первичного ключа:

- `GenerationType.AUTO` — поставщик JPA определяет способ генерации значений на основе выбранной базы данных;
- `GenerationType.IDENTITY` — значения генерируются в базе данных с автоинкрементной колонкой;
- `GenerationType.SEQUENCE` — значения генерируются последовательностью;
- `GenerationType.TABLE` — значения генерируются с помощью вспомогательной таблицы, которая эмулирует последовательности.

Сервер виртуальных питомцев для генерируемых ключей во всех случаях использует последовательности с именем *имя_таблицы_имя_колонки*_seq.

- В атрибуте `generator` указывается название генератора из атрибута `name` аннотации `@SequenceGenerator`.
- Аннотация `@SequenceGenerator` указывает на последовательность, которая используется для генерации значений.
- Атрибут `name` аннотации `@SequenceGenerator` — это имя генератора, указываемое в атрибуте `generator` аннотации `@GeneratedValue`.
- В атрибуте `sequenceName` аннотации `@SequenceGenerator` указывается имя последовательности в базе данных. В таблице `pet` для генерации значений колонки `id` используется последовательноть `pet_id_seq`.
- В атрибуте `allocationSize` указывается количество, на которое увеличивается счетчик в последовательности. Число в `allocationSize` определяет количество идентификаторов, которое блокируется для использования при запросе следующих значений из последовательности. Значение по умолчанию: 50.

Значение `allocationSize` обязательно должно совпадать с `INCREMENT` для последовательности в базе данных — в противном случае корректная работа с последовательностью будет невозможна из-за конфликтов.

> ***Необходимо обязательно указывать allocationSize***
>
> При использовании `@SequenceGenerator` атрибут `allocationSize` не обязателен, но в реальности его всегда нужно указывать либо указывать значение 50 в DDL-скриптах создания последовательности в качестве инкремента.

7.4.12. Оптимистичная блокировка

Содержимое таблицы `pet` в базе данных постоянно обновляется, при этом изменения могут происходить в разных потоках. Для обеспечения консистентности изменений используется *механизм оптимистичной блокировки* с помощью аннотации `@Version`, описанной спецификацией Jakarta Persistence (листинг 7.42).

Листинг 7.42. Pet.java

```
@Version
private int version;
```

В сервере виртуальных питомцев для поля, определяемого механизмом оптимистичной блокировки, используется примитивный тип `int`.

Согласно Jakarta Persistence для поля с аннотацией `@Version` разрешены следующие типы данных:

- `int`;
- `short`;
- `long`;
- `java.lang.Integer`;
- `java.lang.Short`;
- `java.lang.Long`;
- `java.sql.Timestamp`;
- `java.time.LocalDateTime` и `java.time.Instant` (поддерживаются, начиная с Jakarta Persistence 3.2, вышедшей 10 апреля 2024 года).

Hibernate позволяет использовать типы из Java 8 Date-Time, а также любые другие типы, если вы объявите свой `UserVersionType`.

Поле с аннотацией `@Version` увеличивается каждый раз при изменении атрибута сущности. При фиксации транзакции значение из поля `@Version` используется менеджером сущностей для обнаружения конфликта одновременных изменений, т. е. ситуаций, когда сущность уже была изменена в другой транзакции с момента чтения сущности в этой транзакции.

Значение поля @Version не должно изменяться кодом вашего приложения

Лучший способ избежать этого — не создавать методов установки и получения значения для поля с аннотацией `@Version`. Hibernate может изменять значения поля без метода установки значения и считывать без метода получения значения.

7.4.13. Связь M:M

Аннотация `@ManyToMany`, используемая в классе `Pet` для полей `books` и `cloths`, применяется для отображения связей M:M (многие-ко-многим) на коллекции (листинг 7.43).

Листинг 7.43. Pet.java

```
@ManyToMany
@JoinTable(name = "pet_cloth",
        joinColumns = @JoinColumn(name = "pet_id"),
        inverseJoinColumns = @JoinColumn(name = "cloth_id"))
private Set<Cloth> cloths;
...
@ManyToMany
@JoinTable(
        inverseJoinColumns = @JoinColumn(name = "book_id"))
private Set<Book> books;
```

- Промежуточная таблица и названия полей для связи M:M определяются с помощью аннотации `@JoinTable` — в атрибуте `name` этой аннотации указывается название промежуточной таблицы. Название по умолчанию составляется из конкатенации названия таблицы сущности, владеющей связью (в которой используется аннотация `@ManyToMany` на поле с коллекцией), и связанной сущности, разделенных символом подчеркивания.
- В атрибуте `joinColumns` указывается название колонки из промежуточной таблицы с ключами таблицы сущности, владеющей связью. Название колонки по умолчанию составляется из названия таблицы и колонки с первичным ключом, разделенных символом подчеркивания.
- В атрибуте `inverseJoinColumns` указывается название колонки из промежуточной таблицы с ключами из связанной таблицы. Название колонки по умолчанию составляется из названия таблицы связанной сущности и колонки с первичным ключом связанной сущности, разделенных символом подчеркивания.

Обратите внимание, что в случае с `books` только для атрибута `inverseJoinColumns` явно указана колонка `book_id`, потому что в противном случае значение по умолчанию было бы `books_id`, а это не то имя колонки, которое нам нужно.

Если в качестве имени поля выбрать `book` вместо `books`, тогда можно было бы обойтись без аннотации `@JoinTable` в принципе, но логически имя поля `books` больше похоже на имя колонки с коллекцией книг, чем имя `book`.

В случае с именем поля `book` аннотации выглядели бы так (листинг 7.44).

Листинг 7.44. Pet.java

```
@ManyToMany
private Set<Book> book;
```

Во многих проектах, в том числе и коммерческих, используется интерфейс `List`, но в большинстве случаев интерфейс `Set` для обработки Hibernate покажет результаты лучше с точки зрения производительности, чем `List`, по следующим причинам:

- при использовании коллекций типа `Set` все коллекции могут быть заполнены с помощью одного `SELECT`, в котором через `LEFT OUTER JOIN` соединяются все необходимые для конечного результата таблицы. При этом вернется больше строк, т. к. некоторые строки могут дублироваться, если аннотаций `@ManyToOne` несколько, а уникальные результаты для коллекций определятся в момент вставки в `Set` с помощью методов `hashCode` и `equals` элементов коллекций;
- контракт `Set`, не допускающий дублирования элементов, позволит избежать повторного добавления элемента в коллекцию, а значит, и повторного добавления строки в таблицу;
- контракт `Set` предполагает отсутствие порядка элементов, тогда как контракт `List` предполагает, что должен быть определенный строгий порядок, что не так в случае использования `List` в Hibernate. При использовании `List` никакой строгий порядок не поддерживается без использования дополнительной сортировки — например, с аннотацией `@OrderColumn`.

7.4.14. Методы *hashCode* и *equals*

Сущности `Book` и `Cloth` переопределяют методы `hashCode` и `equals`, используя `id`. Пример для `Book` приведен в листинге 7.45.

Листинг 7.45. Book.java

```
@Override
public int hashCode() {
    return Objects.hash(id);
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Book other = (Book) obj;
    return Objects.equals(id, other.id);
}
```

В сервере виртуальных питомцев справочники наполняются только скриптами Liquibase. Первичный ключ `id` для них всегда будет заполнен — не встретится случая, когда `id == null`, поэтому для поля `id` выбран примитивный тип `int`, а оно само идеально подходит для `hashCode` и `equals`.

Питомцы создаются каждым игроком самостоятельно. В момент создания до сохранения в базу данных первичный ключ питомца будет равен null, поэтому использовать только `id` для методов `hashCode` и `equals` класса `Pet` нельзя.

Класс `Pet` не содержит поля, которое однозначно позволило бы идентифицировать питомца. Имена питомцев могут повторяться, номера паспорта у них нет, уникальных логинов или e-mail'ов тоже нет. В таких случаях для создания методов `hashCode` и `equals` используются следующие способы:

- не переопределять `hashCode` и `equals`, унаследованные от класса `Object`. Не самый лучший вариант, но тоже рабочий. В этом случае использовать Set или Map в качестве коллекций полноценно уже не получится;
- найти часть полей класса, которые позволили бы его однозначно идентифицировать. В случае с классом `Pet` использован именно этот подход — методы `hashCode` и `equals` используют только поля `createdDate`, `name`, `petType`, `user.id`, которые однозначно позволяют определить питомца. У одного пользователя не может быть двух питомцев, созданных в одно и то же время с точностью до миллисекунды, с одним и тем же именем и с одним и тем же типом;
- сгенерировать методы `hashCode` и `equals` так, чтобы они использовали все или почти все поля класса. Вполне работоспособный вариант, но нужно понимать, что после добавления экземпляра класса в Set или при использовании его в качестве ключа Map у него не должны меняться поля, которые задействуются методами `hashCode` и `equals`, т. к. это может привести к изменению их результата работы, а значит, к нарушению контракта Set и Map соответственно. Особенно внимательным нужно быть с первичными ключами, заполняющимися с помощью `@GeneratedValue`, т. к. их значение равно null до сохранения в базу данных.

7.5. Шаблон Data Access Object

7.5.1. DAO на аннотациях JPA

Data Access Object (DAO) — шаблон проектирования, предоставляющий интерфейс для сохранения в базе данных или другой механизм постоянства. Слой бизнес-логики приложения работает только со слоем DAO и изолирован от деталей конкретной реализации (разделение приложения на слои описывается в *главе 2*).

Приложение виртуальных питомцев virtualpet-server-springframework использует имя пакета ru.urvanov.virtualpets.server.dao для хранения интерфейсов и реализаций слоя DAO.

> ***Примечание***
>
> Все примеры *разд. 7.5* и его подразделов взяты из проекта virtualpets-server-springframework. В случае использования Spring Boot эти примеры выглядели бы точ-

но так же (на примере проекта virtualpets-server-springboot далее мы рассмотрим модуль Spring Data JPA).

Jakarta Persistence API, рассмотренный в *разд. 7.4* при описании слоя предметной области, предоставляет интерфейс `EntityManager` и его фабрику `EntityManagerFactory`.

Классы слоя Dao получают экземпляр `EntityManager` с помощью аннотации `@PersistenceContext`. Пример класса `LevelDaoImpl` проекта virtualpets-server-springframework приведен в листинге 7.46.

Листинг 7.46. LevelDaoImpl.java

```
...
@Repository("levelDao")
public class LevelDaoImpl implements LevelDao {

    @PersistenceContext
    private EntityManager em;
...
```

Аннотация `@PersistenceContext` может находиться на методе присваивания значения — если нужна дополнительная логика, которая будет выполняться внутри метода после получения ссылки на `EntityManager` (листинг 7.47).

Листинг 7.47. FoodDaoImpl.java

```
...
@Repository(value = "foodDao")
public class FoodDaoImpl implements FoodDao {

    private EntityManager em;

    @PersistenceContext
    public void setEntityManager(EntityManager em) {
        this.em = em;
    }
...
```

Hibernate как поставщик JPA реализует интерфейсы `EntityManager` и `EntityManagerFactory` и в дополнение к ним имеет свои специфичные интерфейсы `Session` и `SessionFactory`. Интерфейс `Session` можно рассматривать как специфичный для Hibernate аналог `EntityManager`, а интерфейс `SessionFactory` — как специфичный для Hibernate интерфейс `EntityManagerFactory`.

`EntityManager` содержит достаточно большое количество методов для получения сущностей из базы данных, сохранения сущностей, удаления сущностей и т. п.

7.5.2. Выборка сущности

Для получения сущностей из базы данных чаще всего применяется метод `find`. Метод перегружен и имеет различные варианты с разным количеством и типом

параметров. В листинге 7.48 приведен пример использования метода `find` для класса `LevelDaoImpl` (подобный пример мы уже видели в *разд. 7.4.6*).

Листинг 7.48. LevelDaoImpl.java

```
@Transactional(readOnly = true)
@Override
public Optional<Level> findById(Integer id) {
    Level level = em.find(Level.class, id);
    return Optional.ofNullable(level);
}
```

Обратите внимание на аннотацию `@Transactional` с атрибутом `readOnly = true`. Аннотация `@Transactional` указывает, что метод `findById` должен выполняться внутри транзакции. Атрибут `readOnly = true` определяет, что в методе осуществляется только считывание данных. Драйверу JDBC в этом случае будет передана подсказка, что в транзакции не станет осуществляться изменение данных, чтобы он оптимизировал свою работу. В дополнение к этому Spring Framework оптимизирует работу провайдера JPA.

> ***Аннотация @Transactional в большинстве случаев используется с readOnly = true***
>
> Для большинства методов с аннотацией `@Transactional` вам необходимо указывать `readOnly = true`, т. к. чаще всего создаются методы считывания данных. Аннотация `@Transactional` без атрибута `readOnly = true` указывается для методов, изменяющих состояние базы данных, — обычно это только метод сохранения сущности в классе DAO.

В сервере виртуальных питомцев включено логирование SQL-методов, генерируемых Hibernate, поэтому, если вы поставите точку остановки на методе `findById`, то при пошаговом выполнении в консоли IDE отобразится выполненный SQL-запрос для получения данных (листинг 7.49).

Листинг 7.49. Фрагмент лога, демонстрирующий генерируемый Hibernate запрос

```
Hibernate: select l1_0."id",l1_0."experience" from
"virtualpets_server_springframework"."level" l1_0 where l1_0."id"=?
```

7.5.3. Получение ссылки на сущность без обращения к БД

Помимо метода `find`, интерфейс `EntityManager` содержит метод `getReference` (листинг 7.50).

Листинг 7.50. ClothDaoImpl.java

```
@Override
public Cloth getReference(String id) {
    return em.getReference(Cloth.class, id);
}
```

Метод `EntityManager#getReference` возвращает экземпляр прокси, в котором заполнено только поле первичного ключа. Обращение к любому другому полю или методу сущности вызовет загрузку экземпляра сущности из базы данных либо исключение `org.hibernate.LazyInitializationException`, если обращение произошло вне сессии Hibernate. Сам метод `EntityManager#getReference` не обращается к базе данных.

Подобный экземпляр прокси, полученный с помощью `EntityManager#getReference`, используется для заполнения связей `@ManyToOne` без лишнего обращения к базе данных — например, полей `hat`, `cloth` и `bow` в `Pet` (листинг 7.51).

Листинг 7.51. Pet.java

```
@ManyToOne(fetch = FetchType.LAZY)
private Cloth hat;

@ManyToOne(fetch = FetchType.LAZY)
private Cloth cloth;

@ManyToOne(fetch = FetchType.LAZY)
private Cloth bow;
```

Код метода `savePetCloths` в классе `PetServiceImpl` заполняет поля `hat`, `cloth` и `bow`, не нагружая базу данных дополнительными запросами на загрузку экземпляров `Cloth` из базы данных (листинг 7.52).

Листинг 7.52. PetServiceImpl.java

```
Cloth hat = null;
if (saveClothArg.hatId() != null) {
    // Получаем прокси-сущности Cloth
    // без лишнего обращения к базе данных.
    hat = clothDao.getReference(saveClothArg.hatId());
}
Cloth cloth = null;
if (saveClothArg.clothId() != null) {
    // Получаем прокси-сущности Cloth
    // без лишнего обращения к базе данных.
    cloth = clothDao.getReference(saveClothArg.clothId());
}
Cloth bow = null;
if (saveClothArg.bowId() != null) {
    // Получаем прокси-сущности Cloth
    // без лишнего обращения к базе данных.
    bow = clothDao.getReference(saveClothArg.bowId());
}
pet.setHat(hat);
pet.setCloth(cloth);
pet.setBow(bow);
```

7.5.4. Сохранение сущности

Сохранение сущностей осуществляется методами `persist` и `merge`. В простейшем случае самое короткое объяснение смысла этих методов выглядит следующим образом:

- метод `persist` применяется для сохранения новой сущности;
- метод `merge` служит для изменения существующей сущности.

Объяснение поверхностное и не совсем правильно отражает происходящее (более подробно эти методы описываются в *разд. 7.5.5*).

Пример использования методов `persist` и `merge` из класса `PetDaoImpl` приведен в листинге 7.53.

Листинг 7.53. PetDaoImpl.java

```
@Override
@Transactional
public void save(Pet pet) {
    if (pet.getId() == null) {
        em.persist(pet);
    } else {
        em.merge(pet);
    }
}
```

> ***Важно***
>
> **Здесь аннотация `@Transactional` приведена *без* атрибута `readOnly = true` — потому что в методе `save` выполняется изменение данных.**

7.5.5. Состояния сущностей Hibernate

Почему используются два метода: `persist` и `merge`? Чем они отличаются? Для понимания смысла этих методов необходимо иметь представление о состояниях сущностей и переходах между этими состояниями.

Интерфейсы `org.hibernate.Session` и `jakarta.persistence.EntityManager` представляют контекст для работы с постоянно хранимыми данными. Этот контекст носит название `persistence context`. Сохраняемые данные имеют свое состояние как относительно `persistence context`, так и относительно базы данных. Сущность Hibernate может находиться при этом в одном из четырех состояний:

- transient — экземпляр сущности был создан, но не связан с `persistence context`. Он не сохранен в базе данных и обычно не имеет идентификатора;
- managed или persistent — сущность связана с `persistence context` и имеет связанный с ней идентификатор. Она может как существовать в базе данных, так и пока отсутствовать в ней;
- detached — с сущностью связан идентификатор, но она больше не связана с `persistence context` (обычно из-за того, что `persistence context` был закрыт либо экземпляр был удален из него);

- removed — сущность имеет идентификатор и связана с `persistence context`, но она запланирована к удалению из базы данных.

Сущности меняют свои состояния (рис. 7.9):

1. После создания с помощью `new` сущность находится в состоянии transient.
2. Вызов `em.persist()` сущности, находящейся в состоянии transient, включает ее в `persistence context` и переводит в состояние managed (persistent).
3. Закрытие контекста постоянства, фиксация или откат транзакции переводит все сущности из `persistence context` в состояние detached.
4. Метод `em.merge()` переводит сущность из состояния detached назад в состояние managed (persistent).
5. Метод `em.remove()` переводит сущность в состояние removed.

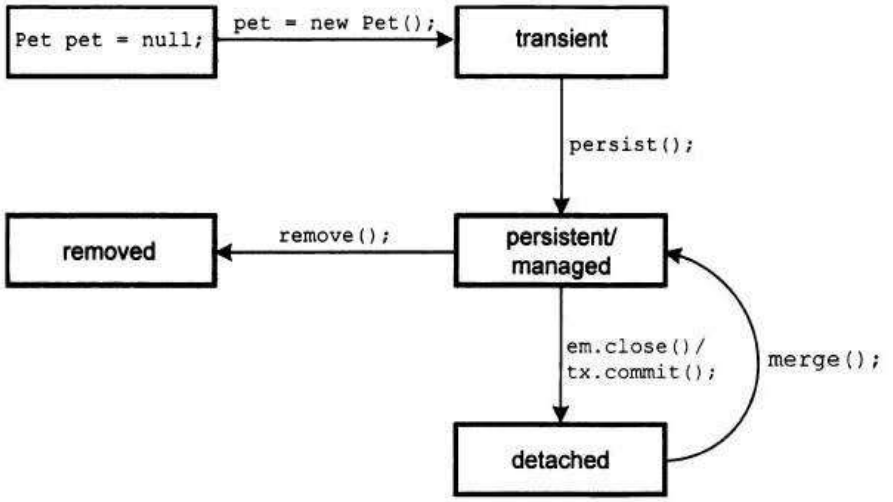


Рис. 7.9. Переход сущности между состояниями

Изменения сущности в состоянии persistent (managed) отражаются в базе данных

Для сущностей, находящихся в состоянии persistent (managed), не нужно вызывать методы `EntityManager` `persist` или `merge` — их состояния автоматически отразятся в базе без вызова `persist`, `merge` или `remove`. Метод `save` из `PetDaoImpl`, упомянутый в *разд 7.5.4*, имеет смысл только для сущностей, находящихся в состоянии transient или detached.

Для метода `PetDaoImpl#save` имеют смысл два варианта использования:

- пользователь создал нового питомца, которого нужно сохранить в базе данных. В этом случае `pet.id == null`, поэтому исполнение кода пойдет по ветке `em.persist(pet)`, что вызовет добавление строки в таблицу `pet` и таблицы из связанных коллекций еды, строительных материалов, одежды и т. д.;
- транзакция, в которой сущность питомца получена, была завершена (например исполнение кода вышло из метода, аннотированного `@Transactional`), но необхо-

димо отразить последние изменения в сущности, которые были произведены уже после завершения транзакции. В этом случае поле pet.id уже заполнено, и исполнение кода пойдет по ветке em.merge(pet), что, в свою очередь, вызовет обновление строки из таблицы pet и добавление/обновление строк из таблиц сущностей связанных коллекций еды, строительных материалов, одежды и т. д.

7.5.6. Удаление сущности

Пример удаления сущности из persistence context и из базы данных соответственно приведен в листинге 7.54.

Листинг 7.54. PetDaoImpl.java

```
@Override
@Transactional
public void delete(Pet fullPet) {
    em.remove(fullPet);
}
```

Метод em.remove переведет сущность pet в состояние removed, что запланирует ее удаление из базы данных. Обратите внимание, что метод принимает экземпляр сущности, в которой должны быть загружены все поля и все связанные коллекции. Для этого следует использовать метод PetDaoImpl#findFullById, рассмотренный далее в *разд. 7.5.7*. Выборка сущности Pet и всех связанных сущностей необходима для каскадного удаления связанных записей.

Hibernate сгенерирует DML-команды для удаления связанных сущностей, для которых настроено каскадное удаление записей. В случае с сущностью Pet удаляются все элементы связанных коллекций (листинг 7.55).

Листинг 7.55. Сгенерированные DML-команды удаления

```
Hibernate: delete from "virtualpets_server_springframework"."pet_book" where "pet_id"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_cloth" where "pet_id"=?
Hibernate: delete from "virtualpets_server_springframework"."room"
                                              where "pet_id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_achievement"
                                              where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_achievement"
                                              where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_achievement"
                                              where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_achievement"
                                              where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_achievement"
                                              where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_achievement"
                                              where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_achievement"
                                              where "id"=? and "version"=?
```

```
Hibernate: delete from "virtualpets_server_springframework"."pet_building_material"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_building_material"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_drink"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_food"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_journal_entry"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_journal_entry"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_journal_entry"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_journal_entry"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_journal_entry"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_journal_entry"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_journal_entry"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_journal_entry"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_journal_entry"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet_journal_entry"
                                           where "id"=? and "version"=?
Hibernate: delete from "virtualpets_server_springframework"."pet"
                                           where "id"=? and "version"=?
```

7.5.7. Именованные запросы

Метод `PetDaoImpl#findFullById` использует именованный запрос (листинг 7.56).

Листинг 7.56. PetDaoImpl.java

```
@Override
@Transactional(readOnly = true)
public Optional<Pet> findFullById(Integer id) {
    TypedQuery<Pet> query = em.createNamedQuery(
            "Pet.findFullById", Pet.class);
    query.setParameter("id", id);
    List<Pet> pets = query.getResultList();
    return DataAccessUtils.optionalResult(pets);
}
```

Здесь метод `createNamedQuery` интерфейса `EntityManager` создает экземпляр `TypedQuery`, использующийся для выполнения именованного запроса `Pet.findFullById`, переданного в качестве параметра метода `createNamedQuery`. Именованный запрос — это не объект базы данных. Именованные запросы описываются в аннотациях к сущности. Например, именованный запрос `Pet.findFullById` имеет следующий вид (листинг 7.57).

Листинг 7.57. Pet.java

```
...
@Entity
@Table(name = "pet")
@NamedQuery(name = "Pet.findByUserId",
        query = "from Pet p where p.user.id = :userId")
@NamedQuery(name = "Pet.findFullById", query = """
        from Pet p
        left outer join fetch p.level l
        left outer join fetch p.hat hl
        left outer join fetch p.cloth cl
        left outer join fetch p.bow bl
        left outer join fetch p.user u
        left outer join fetch p.cloths c
        left outer join fetch p.books b
        left outer join fetch p.foods f
        left outer join fetch p.buildingMaterials bm
        left outer join fetch bm.buildingMaterial
        left outer join fetch p.drinks d
        left outer join fetch p.journalEntries je
        left outer join fetch p.achievements ach
        where p.id = :id""")
...
public class Pet...
```

Для сущности `Pet` здесь описаны два именованных запроса:

- `Pet.findByUserId` — выборка питомца по идентификатору пользователя;
- `Pet.findFullById` — выборка питомца по идентификатору с выборкой всех связанных сущностей. Выбираемые сущности указаны через `left outer join fetch`.

> ***В атрибуте query аннотации @NamedQuery используется JPQL***
>
> Запросы выборки данных для именованных запросов описаны с помощью языка JPQL (Jakarta Persistence Query Language), а не SQL. Язык JPQL имеет определенное сходство с SQL, но это совершенно другой язык, в котором вы манипулируете сущностями JPA, а не объектами базы данных.

Язык JPQL позволяет накладывать условия на выбираемые сущности, принудительно выбирать данные для связанных сущностей, помеченных `FetchType.LAZY`, сортировать выбираемые данные и т. п.

Генерируемая Hibernate SQL-команда именованного запроса `Pet.findFullById` выглядит как объединение связанных таблиц через `left join` (листинг 7.58).

Листинг 7.58. Генерируемая Hibernate SQL-команда именованного запроса Pet.findFullById

```
Hibernate: select
p1_0."id",a1_0."pet_id",a1_0."achievement_id",a1_0."id",a1_0."achievement_id",a1_0."version",
a1_0."was_shown",b2_0."pet_id",b2_1."id",b2_1."bookcase_id",b2_1."bookcase_order",
b2_1."hidden_objects_game_drop_rate",b1_0."id",b1_0."cloth_type",b1_0."hidden_objects_game_
drop_rate",b1_0."wardrobe_order",p1_0."build_count",b3_0."pet_id",b3_0."building_material_
```

```
id",b3_0."id",b4_0."id",b4_0."hidden_objects_game_drop_rate",b4_0."newbie_box_drop_max",b4_0."
newbie_box_drop_min",b4_0."newbie_box_drop_rate",b4_0."rucksack_order",b3_0."building_
material_count",b3_0."version",c1_0."id",c1_0."cloth_type",c1_0."hidden_objects_game_drop_
rate",c1_0."wardrobe_order",c2_0."pet_id",c2_1."id",c2_1."cloth_type",c2_1."hidden_objects_
game_drop_rate",c2_1."wardrobe_order",p1_0."comment",p1_0."created_date",p1_0."drink",p1_0.
"drink_count",d1_0."pet_id",d1_0."drink_id",d1_0."id",d1_0."drink_count",d1_0."version",p1_0."
eat_count",p1_0."education",p1_0."every_day_login_count",p1_0."every_day_login_last",p1_0.
"experience",f1_0."pet_id",f1_0."food_id",f1_0."id",f1_0."food_count",f1_0."version",h1_0.
"id",h1_0."cloth_type",h1_0."hidden_objects_game_drop_rate",h1_0."wardrobe_order",p1_
0."hidden_objects_game_count",j1_0."pet_id",j1_0."journal_entry_id",j1_0."id",j1_0.
"created_at",j1_0."journal_entry_id",j1_0."readed",j1_0."version",l1_0."id",l1_0."experience",
p1_0."login_date",p1_0."mood",p1_0."name",p1_0."pet_type",p1_0."satiety",p1_0."session_key",
p1_0."teach_count",u1_0."id",u1_0."active_date",u1_0."birthdate",u1_0."city",u1_0."comment",
u1_0."country",u1_0."email",u1_0."login",u1_0."login_date",u1_0."name",u1_0."password",u1_
0."recover_password_key",u1_0."recover_password_valid",u1_0."registration_date",u1_0."role",
u1_0."sex",u1_0."unid",u1_0."version",p1_0."version" from
"virtualpets_server_springframework"."pet" p1_0 left join
"virtualpets_server_springframework"."level" l1_0 on l1_0."id"=p1_0."level_id" left join
"virtualpets_server_springframework"."cloth" h1_0 on h1_0."id"=p1_0."hat_id" left join
"virtualpets_server_springframework"."cloth" c1_0 on c1_0."id"=p1_0."cloth_id" left join
"virtualpets_server_springframework"."cloth" b1_0 on b1_0."id"=p1_0."bow_id" left join
"virtualpets_server_springframework"."user" u1_0 on u1_0."id"=p1_0."user_id" left join
("virtualpets_server_springframework"."pet_cloth" c2_0 join
"virtualpets_server_springframework"."cloth" c2_1 on c2_1."id"=c2_0."cloth_id") on
p1_0."id"=c2_0."pet_id" left join ("virtualpets_server_springframework"."pet_book" b2_0
join "virtualpets_server_springframework"."book" b2_1 on b2_1."id"=b2_0."book_id") on
p1_0."id"=b2_0."pet_id" left join "virtualpets_server_springframework"."pet_food" f1_0 on
p1_0."id"=f1_0."pet_id" left join "virtualpets_server_springframework"."pet_building_material"
b3_0 on p1_0."id"=b3_0."pet_id" left join
"virtualpets_server_springframework"."building_material" b4_0 on
b4_0."id"=b3_0."building_material_id" left join
"virtualpets_server_springframework"."pet_drink" d1_0 on p1_0."id"=d1_0."pet_id" left join
"virtualpets_server_springframework"."pet_journal_entry" j1_0 on p1_0."id"=j1_0."pet_id" left
join "virtualpets_server_springframework"."pet_achievement" a1_0 on p1_0."id"=a1_0."pet_id"
where p1_0."id"=?
```

Именованный запрос `Pet.findFullById` использовался здесь нами только для выборки питомца с принудительной выборкой всех связанных сущностей.

7.5.8. Графы сущностей

Принудительная выборка связанных сущностей возможна не только с помощью JPQL. Аналогичного результата позволяют добиться графы сущностей, используемые для описания заполняемых данными связанных сущностей.

Графы сущностей описываются аннотациями `@NamedEntityGraph`. Пример для сущности `Pet` приведен в листинге 7.59.

Листинг 7.59. Pet.java

```
...
@NamedEntityGraph(name = "pet.cloths",
        attributeNodes = @NamedAttributeNode("cloths")
)
public class Pet implements Serializable {
...
```

Имя графа сущностей `pet.cloths` указывается здесь в атрибуте `name` аннотации `@NamedEntityGraph`.

В атрибуте `attributeNodes` записываются поля сущности, которые необходимо принудительно заполнить данными при использовании графа сущности. Именованный граф `pet.cloths` принудительно заполняет данными коллекцию `cloths` сущности `Pet`.

Получение графа сущности осуществляется методом `getEntityGraph` менеджера сущностей, полученный экземпляр передается в список параметров метода `find` в ключе `jakarta.persistence.fetchgraph` экземпляра `Map`, как это показано в `PetDaoImpl.java` (листинг 7.60).

Листинг 7.60. PetDaoImpl.java

```
@Override
@Transactional(readOnly = true)
public Optional<Pet> findByIdWithFullCloths(Integer id) {
    Pet pet = em.find(Pet.class, id,
            Map.of(
                    "jakarta.persistence.fetchgraph",
                    em.getEntityGraph("pet.cloths"))
            );
    return Optional.ofNullable(pet);
}
```

Hibernate генерирует для именованного графа сущностей `pet.cloths` команду SQL, в которой через `left join` соединяет таблицы `pet`, `pet_cloth` и `cloth` (листинг 7.61).

Листинг 7.61. Генерируемая Hibernate команда SQL для графа сущностей pet.cloths

```
Hibernate: select
p1_0."id",p1_0."bow_id",p1_0."build_count",p1_0."cloth_id",c2_0."pet_id",c2_1."id",c2_1.
"cloth_type",c2_1."hidden_objects_game_drop_rate",c2_1."wardrobe_order",p1_0."comment",p1_0.
"created_date",p1_0."drink",p1_0."drink_count",p1_0."eat_count",p1_0."education",p1_0."every_
day_login_count",p1_0."every_day_login_last",p1_0."experience",p1_0."hat_id",p1_0."hidden_
objects_game_count",p1_0."level_id",p1_0."login_date",p1_0."mood",p1_0."name",p1_0."pet_type",
p1_0."satiety",p1_0."session_key",p1_0."teach_count",p1_0."user_id",p1_0."version" from
"virtualpets_server_springframework"."pet" p1_0 left join
("virtualpets_server_springframework"."pet_cloth" c2_0 join
"virtualpets_server_springframework"."cloth" c2_1 on c2_1."id"=c2_0."cloth_id")
on p1_0."id"=c2_0."pet_id" where p1_0."id"=?
```

> ***Задание***
>
> **Поставьте точку остановки на методе `findByIdWithFullCloths` и проверьте самостоятельно в консоли IDE команду SQL, сгенерированную Hibernate. Убедитесь, что происходит объединение данных из таблиц `pet`, `pet_cloth` и `cloth` через `left join`.**

Графы сущностей позволяют указывать несколько полей в атрибуте `attributeNodes` аннотации `@NamedEntityGraph` (листинг 7.62).

Листинг 7.62. Pet.java

```
...
@NamedEntityGraph(name = "pet.journalEntriesAndAchievements",
        attributeNodes = {
                @NamedAttributeNode("journalEntries"),
                @NamedAttributeNode("achievements")
        }

...
public class Pet...
```

Здесь именованный граф сущностей `pet.journalEntriesAndAchievements` принудительно заполняет поля `journalEntries` и `achievements` класса `Pet`. Его использование аналогично именованному графу сущностей `pet.cloths` (листинг 7.63).

Листинг 7.63. PetDaoImpl.java

```
@Override
@Transactional(readOnly = true)
public Optional<Pet> findByIdWithJournalEntriesAndAchievements(
        Integer id) {
    Pet pet = em.find(Pet.class, id,
            Map.of(
                    "jakarta.persistence.fetchgraph",
                    em.getEntityGraph(
                            "pet.journalEntriesAndAchievements"))
            );
    return Optional.ofNullable(pet);
}
```

Атрибут `subgraphs` аннотации `@NamedEntityGraph` позволяет описывать несколько уровней заполняемых сущностей. С помощью `subgraphs`, например, создан именованный граф сущностей `pet.foods`, в котором данными заполняется не только коллекция `foods` сущности `Pet`, но и поле `food` внутри элементов коллекции `foods` (листинг 7.64).

Листинг 7.64. Pet.java

```
...
@NamedEntityGraph(name = "pet.foods",
        attributeNodes = @NamedAttributeNode(
                value = "foods",
                subgraph = "pet.foods.food"),
        subgraphs = @NamedSubgraph(
                name = "pet.foods.food",
                attributeNodes = @NamedAttributeNode("food"))
)
...
public class Pet...
```

Синтаксис использования именованного графа при этом остается прежним (листинг 7.65).

Листинг 7.65. PetDaoImpl.java

```
@Override
@Transactional(readOnly = true)
public Optional<Pet> findByIdWithFullFoods(Integer id) {
    Pet pet = em.find(Pet.class, id,
            Map.of(
                    "jakarta.persistence.fetchgraph",
                    em.getEntityGraph("pet.foods"))
            );
    return Optional.ofNullable(pet);
}
```

7.5.9. Criteria API

В сервере виртуальных питомцев, кроме JPQL и графов сущностей, используется *criteria query*.

> **Criteria query — это аналог JPQL, но более безопасный с точки зрения ошибок программирования и типов данных способ создания запросов.**

Для работы с criteria query необходимо подключить дополнительный обработчик аннотаций Hibernate Metamodel Generator (`hibernate-jpamodelgen`) к плагину `maven-compiler-plugin`.

Пример подключения Hibernate Metamodel Generator в проекте virtualpets-server-springframework приведен в листинге 7.66.

Листинг 7.66. pom.xml

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.12.1</version>
    <configuration>
        <annotationProcessorPaths>
            <path>
                <groupId>org.hibernate.orm</groupId>
                <artifactId>hibernate-jpamodelgen</artifactId>
                <version>${hibernate.version}</version>
            </path>
        </annotationProcessorPaths>
        <source>17</source>
        <target>17</target>
    </configuration>
</plugin>
```

Обратите внимание на тег `annotationProcessorPaths`, где указываются `groupId`, `artifactId` и версия `hibernate-jpamodelgen`. В качестве версии берется версия `hibernate`, указанная в теге `properties` (листинг 7.67).

Листинг 7.67. pom.xml

```
<hibernate.version>6.5.2.Final</hibernate.version>
```

Для virtualpets-server-springboot подключение Hibernate Metamodel Generator выглядит аналогично.

Hibernate Metamodel Generator генерирует классы метамодели, используемые при построении запросов с помощью criteria query. Сгенерированные классы метамодели имеют имена, идентичные именам сущностей, с добавлением символа подчеркивания. Например: `Pet` → `Pet_`, `Book` → `Book_`.

> ***Классы метамодели не хранятся в системе контроля версий***
>
> Сгенерированные Hibernate Metamodel Generator классы метамодели находятся в каталоге target. Их не хранят в системе контроля версий (например, в Git) — в этом нет смысла, т. к. они сгенерируются сами при сборке проекта.

Пример сгенерированного класса метамодели приведен в листинге 7.68.

Листинг 7.68. Book_.java

```
package ru.urvanov.virtualpets.server.dao.domain;

import jakarta.persistence.metamodel.SingularAttribute;
import jakarta.persistence.metamodel.StaticMetamodel;
import javax.annotation.processing.Generated;

@Generated(value = "org.hibernate.jpamodelgen.JPAMetaModelEntityProcessor")
@StaticMetamodel(Book.class)
public abstract class Book_ {

	public static volatile SingularAttribute<Book, Float> hiddenObjectsGameDropRate;
	public static volatile SingularAttribute<Book, Integer> bookcaseLevel;
	public static volatile SingularAttribute<Book, String> id;
	public static volatile SingularAttribute<Book, Integer> bookcaseOrder;

	public static final String HIDDEN_OBJECTS_GAME_DROP_RATE = "hiddenObjectsGameDropRate";
	public static final String BOOKCASE_LEVEL = "bookcaseLevel";
	public static final String ID = "id";
	public static final String BOOKCASE_ORDER = "bookcaseOrder";
}
```

Рассмотрим выборку записи по одному полю как самый простой пример использования criteria query (листинг 7.69).

Листинг 7.69. RoomDaoImpl.java

```
@Override
@Transactional(readOnly = true)
public Optional<Room> findByPetId(Integer petId) {
    CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();                    // (1)
    CriteriaQuery<Room> criteriaQuery = criteriaBuilder.createQuery(Room.class);  // (2)
```

```
    Root<Room> rootRoom = criteriaQuery.from(Room.class);          // (3)
    criteriaQuery.select(rootRoom);                                // (4)
    Predicate predicate = criteriaBuilder.equal(
            rootRoom.get(Room_.petId),
            petId);                                                // (5)
    criteriaQuery.where(predicate);                                // (6)
    TypedQuery<Room> query = em.createQuery(criteriaQuery);        // (7)
    List<Room> rooms = query.getResultList();                      // (8)
    return DataAccessUtils.optionalResult(rooms);                  // (9)
}
```

Кода в этом примере гораздо больше, чем было в примерах для JPQL, но этот код безопаснее с точки зрения типов. Рассмотрим его подробнее по шагам:

1. Создание экземпляра класса, реализующего интерфейс `jakarta.persistence.criteria.CriteriaBuilder`, используемый для построения запросов, выборок, выражений, порядка сортировки и предикатов.
2. Создание экземпляра класса, реализующего интерфейс `jakarta.persistence.criteria.CriteriaQuery`, представляющего собой запрос.
3. Получение экземпляра класса, реализующего интерфейс `jakarta.persistence.criteria.Root`, с помощью метода `from` экземпляра `CriteriaQuery`, созданного на шаге 2, и представляющего собой сущность, из таблицы которой будет производиться выборка данных.
4. С помощью метода `select` интерфейса `CriteriaQuery` указывается, что необходимо выбрать данные для сущности `Room`.
5. Создание предиката фильтрации данных по полю `petId` — выбираются строки со значением в поле `petId`, равным переданному в аргументе `petId`.
6. С помощью метода `where` интерфейса `CriteriaQuery` передается предикат, созданный на шаге 5, в экземпляр `CriteriaQuery` из шага 2.
7. Создание экземпляра класса, реализующего интерфейс `jakarta.persistence.TypedQuery`, использующегося для выполнения типизированных запросов. Интерфейс `TypedQuery` позволяет указывать параметры запроса через `setParameter`, но в нашем случае параметр уже зашит переданным значением `petId` в самом предикате.
8. Метод `getResultList` интерфейса `TypedQuery` выполняет запрос в базе данных и возвращает список экземпляров `Room`, заполненный данными из запроса. Вместо него в нашем случае можно использовать метод `getSingleResult`, но он бросает исключение `NonUniqueResultException`, если находит несколько строк, либо исключение `NoResultException`, если сгенерированный запрос к базе данных не вернет ни одной строки.
9. `DataAccessUtils.optionalResult` — это новый метод, добавленный в Spring Framework 6.1. Метод позволяет преобразовать коллекцию, содержащую один элемент, в `Optional` с этим элементом либо преобразовать пустую коллекцию в пустой `Optional`.

Аналогичный вариант на JPQL:

```
"from Room r where r.petId = " + petId;
```

CriteriaQuery позволяет выбирать не только единичные сущности, но и агрегировать данные, вычисляя сумму, количество, среднее значение, минимальное значение и максимальное значение (листинг 7.70).

Листинг 7.70. RoomDaoImpl.java

```
@Override
@Transactional(readOnly = true)
public long existsByPetId(Integer petId) {
    CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();                    // (1)
    CriteriaQuery<Long> criteriaQuery = criteriaBuilder.createQuery(Long.class); // (2)
    Root<Room> rootRoom = criteriaQuery.from(Room.class);                         // (3)
    Expression<Long> count = criteriaBuilder.count(rootRoom.get(Room_.petId));    // (4)
    criteriaQuery.select(count);                                                  // (5)
    Predicate predicate = criteriaBuilder.equal(
            rootRoom.get(Room_.petId),
            petId);                                                 // (6)
    criteriaQuery.where(predicate);                                 // (7)
    TypedQuery<Long> query = em.createQuery(criteriaQuery);         // (8)
    return query.getSingleResult();                                 // (9)
}
```

Код метода `existsByPetId` выглядит похоже на код метода `findByPetId`. Единственное отличие заключается в том, что здесь на шаге 5 в метод `select` интерфейса `CriteriaQuery` передается экземпляр интерфейса `Expression`, созданный с помощью метода `count` интерфейса `CriteriaBuilder` на шаге 4.

До сих пор рассматривались только примеры с выборкой единственного результата (сущности или числа). Метод `getResultList` интерфейса `TypedQuery` позволяет выбрать список сущностей с данными, полученными в результате выполнения сгенерированного запроса (листинг 7.71).

Листинг 7.71. BookDaoImpl.java

```
@Override
@Transactional(readOnly = true)
public List<Book> findAllOrderByBookcaseLevelAndBookcaseOrder() {
    CriteriaBuilder cb = em.getCriteriaBuilder();                    // (1)
    CriteriaQuery<Book> criteriaQuery = cb.createQuery(Book.class); // (2)

    Root<Book> rootBook = criteriaQuery.from(Book.class); //          (3)
    criteriaQuery.select(rootBook); //                                (4)
    criteriaQuery.orderBy(
            cb.asc(rootBook.get(Book_.bookcaseLevel)),
            cb.asc(rootBook.get(Book_.bookcaseOrder)));              // (5)
    TypedQuery<Book> query = em.createQuery(criteriaQuery);          // (6)
    return query.getResultList();                                    // (7)
}
```

На шаге 7 используется метод `getResultList` интерфейса `TypedQuery`, возвращающий список, который и становится результатом метода `findAllOrderByBookcaseLevelAndBookcaseOrder`.

На шаге 5 указывается порядок сортировки результата. В результирующем списке `List<Book>` книги будут отсортированы по уровню книжного шкафа и по полю `bookcaseOrder` сущности `Book`. Обратите внимание, что для указания порядка сортировки используются классы метамодели `Book_`, сгенерированные с помощью Hibernate Metamodel Generator.

7.5.10. Нативные запросы

В *разд. 7.3* рассматривались классы `JdbcTemplate` и `JdbcClient`, используемые для выполнения запросов SQL. Методы `createNamedQuery` и `createNativeQuery` интерфейса `EntityManager` также позволяют выполнять SQL-запросы к базе данных:

- перегруженный метод `createNativeQuery` интерфейса `EntityManager` создает экземпляры `Query`, используемые для выполнения SQL-запросов, переданных в качестве параметра;
- метод `createNamedQuery` интерфейса `EntityManager` применяется для выполнения именованных SQL-запросов к базе данных. В этом случае сам текст запроса и отображение свойств результирующей сущности на колонки результата запроса указываются с помощью аннотаций к классу сущности.

Именованный SQL-запрос описывается с помощью аннотации `@NamedNativeQuery`, поставленной на классе сущности, как это сделано для сущности `Drink` в листинге 7.72.

Листинг 7.72. Drink.java

```
@NamedNativeQuery(
        name = "Drink.findAllOrderByMachineWithDrinksLevelAndMachineWithDrinksOrder",
        query = """
                SELECT
                    d.id,
                    d.machine_with_drinks_id,
                    d.machine_with_drinks_order,
                    d.hidden_objects_game_drop_rate
                FROM drink d
                ORDER BY d.machine_with_drinks_id,
                    d.machine_with_drinks_order
        """,
        resultSetMapping = "Drink.defaultMapping"
        )
```

В этом примере описывается именованный запрос с именем `Drink.findAllOrderByMachineWithDrinksLevelAndMachineWithDrinksOrder` и текстом SQL-запроса, выбирающим все записи из таблицы `drink` с сортировкой по колонкам `machine_with_drinks_id` и `machine_with_drinks_order`. Текст запроса указан в атрибуте `query` аннотации `@NamedNativeQuery`. В атрибуте `resultSetMapping` указывается имя отображения результата на поля сущности.

Отображение результа SQL-запроса на поля сущности описывается с помощью аннотации `@SqlResultSetMapping`, проставленной над классом сущности (листинг 7.73).

Листинг 7.73. Drink.java

```
@SqlResultSetMapping(
        name = "Drink.defaultMapping",
        entities = {
                @EntityResult(entityClass = Drink.class, fields = {
                        @FieldResult(
                                name = "id",
                                column = "id"),
                        @FieldResult(
                                name = "machineWithDrinksLevel",
                                column = "machine_with_drinks_id"),
                        @FieldResult(
                                name = "machineWithDrinksOrder",
                                column = "machine_with_drinks_order"),
                        @FieldResult(
                                name = "hiddenObjectsGameDropRate",
                                column = "hidden_objects_game_drop_rate")
                })
        }
)
```

В атрибуте `name` аннотации указывается имя отображения результата запроса, оно же определяется в атрибуте `resultSetMapping` аннотации `@NamedNativeQuery`. В атрибуте `entities` с помощью аннотации `@EntityResult` задается имя класса сущности и отображение полей на колонки результата запроса (в случае с `Drink.defaultMapping` — класс сущности `Drink.class`). Поля результата запроса указываются в атрибуте `column` аннотации `@FieldResult`, а в атрибуте `name` аннотации `@FieldResult` определяется имя соответствующего поля из класса сущности, на которое отображается эта колонка. Количество элементов `@FieldResult` должно совпадать с количеством отображаемых колонок результата — по одному элементу на каждую колонку.

Пример выполнения именованного запроса приведен в листинге 7.74.

Листинг 7.74. DrinkDaoImpl.java

```
@Override
@Transactional(readOnly = true)
public List<Drink> findAllOrderByMachineWithDrinksLevelAndMachineWithDrinksOrder() {
    TypedQuery<Drink> namedQuery = em.createNamedQuery(
                        "Drink.findAllOrderByMachineWithDrinksLevelAndMachineWithDrinksOrder",
                         Drink.class);
    return namedQuery.getResultList();
}
```

7.6. Hibernate

Hibernate — это одна из самых популярных реализаций JPA. При описании JPA в *разд. 7.4* и *7.5*, а также и в других главах книги в первую очередь рассматривается спецификация JPA на примере реализации Hibernate (рис. 7.10).

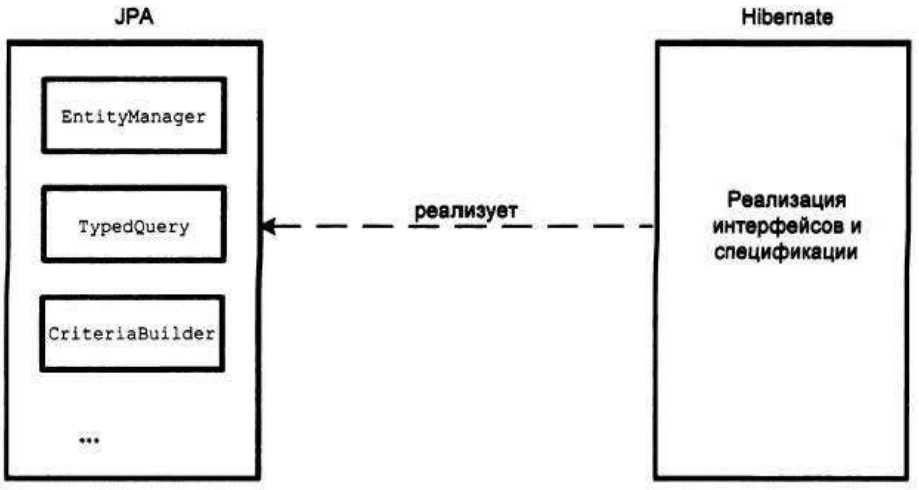


Рис. 7.10 Связь JPA и Hibernate друг с другом

Hibernate не только реализует спецификацию JPA, но и имеет собственные особенности и возможности — в частности, позволяет использовать интерфейсы `Session` и `SessionFactory` вместо интерфейсов `EntityManager` и `EntityManagerFactory`, предоставляемых спецификацией JPA. В большинстве случаев в этом нет особого смысла, т. к. интерфейсы EntityManager и EntityManagerFactory предоставляют достаточно возможностей. Если же имеется потребность использовать интерфейсы Hibernate, то их всегда можно получить с помощью методов `unwrap`.

В проекте virtualpets-server-springframework в качестве примера показано использование методов `unwrap` и получение интерфейсов `Hibernate` в методе `PetDaoImpl#updatePetsTask` (листинг 7.75).

Листинг 7.75. PetDaoImpl.java

```
@Override
@Transactional
public void updatePetsTask() {
    CriteriaBuilder cb = em.getCriteriaBuilder();
    CriteriaQuery<Object> cq = cb.createQuery();
    Root<Pet> rootPet = cq.from(Pet.class);
    cq.select(rootPet);
    Query query = em.createQuery(cq);
    org.hibernate.query.Query hibernateQuery = ((org.hibernate.query.Query) query
            .unwrap(org.hibernate.query.Query.class));
    hibernateQuery.setCacheMode(CacheMode.IGNORE);
```

```
    @SuppressWarnings("unchecked")
    ScrollableResults<Pet> sr = hibernateQuery.scroll(ScrollMode.FORWARD_ONLY);
    try {
        while (sr.next()) {
            try {
                Pet pet = (Pet) sr.get();
                pet.setMood(decParameter(pet.getMood()));
                pet.setDrink(decParameter(pet.getDrink()));
                pet.setSatiety(decParameter(pet.getSatiety()));
                pet.setEducation(decParameter(pet.getEducation()));
                em.merge(pet);
                Session session = em.unwrap(Session.class);
                session.flush();
                session.clear();
            } catch (Exception ex) {logger.error("updatePetsTask step failed.", ex);
            }
        }
    } finally {
        sr.close();
    }
}
```

Здесь с помощью метода `unwrap` интерфейса `jakarta.persistence.Query` осуществляется доступ к интерфейсу Hibernate `org.hibernate.query.Query`, а также с помощью метода `unwrap` интерфейса `jakarta.persistence.EntityManager` осуществляется доступ к интерфейсу Hibernate `org.hibernate.Session`.

> ***PetDaoImpl#updatePetsTask не обязательно реализовывать именно так***
>
> В показанном случае он приведен специально — для того чтобы показать, каким образом получать указатели на интерфейсы Hibernate из интерфейсов JPA. Существует множество способов пройтись по списку питомцев и обновить их поля. Это можно сделать с помощью Criteria API, запросов JPQL, нативных SQL-запросов, методов find и merge из EntityManager и даже хранимых процедур.

7.7. Spring Data

Spring Data — это проект, облегчающий создание слоя DAO, предоставляющий простую в стиле Spring модель для доступа к данным и при этом оставляющий доступными особенности хранилища данных.

Spring Data позволяет работать не только реляционными базами данных — он содержит множество модулей, специфичных для конкретных баз данных.

Вот его основные модули:

- Spring Data Commons — базовые механизмы, необходимые каждому модулю Spring Data;
- Spring Data JDBC — поддержка JDBC;
- Spring Data R2DBC — поддержка R2DBC;

- Spring Data JPA — поддержка JPA. Это основной модуль Spring Data, используемый в проекте virtualpets-server-springboot этой книги;
- Spring Data KeyValue — репозитории для хранилищ типа ключ-значение;
- Spring Data LDAP — поддержка Spring LDAP и работы со службами каталогов;
- Spring Data MongoDB — поддержка MongoDB;
- Spring Data Redis — простая настройка и работа с Redis из Spring-приложения;
- Spring Data REST — предоставление единой платформы для совершения операций CRUD (создание, чтение, изменение и удаление) над вашими сущностями;
- Spring Data for Apache Cassandra — поддержка Cassandra;
- Spring Data for Apache Geode — поддержка Apache Geode;
- Spring Data Aeroskipe — поддержка Aerospike;
- Spring Data ArangoDB — поддержка ArangoDB;
- Spring Data Couchbase — поддержка Couchbase;
- Spring Data Azure Cosmos DB — поддержка Microsoft Azure Cosmos DB;
- Spring Data Cloud Datastore — поддержка Google Datastore;
- Spring Data Cloud Spanner — поддержка Google Spanner;
- Spring Data DynamoDB — поддержка DynamoDB;
- Spring Data Elasticsearch — поддержка Elasticsearch;
- Spring Data Hazelcast — поддержка Hazelcast;
- Spring Data Jest — модуль для основанных на Elasticsearch клиентов Jest REST. Используется для Spring Data с Elasticsearch, доступными только по HTTP;
- Spring Data Neo4j — поддержка баз данных Neo4j Graph;
- Oracle NoSQL Database SDK for Spring Data — поддержка Oracle NoSQL Database и Oracle NoSQL Cloud Service;
- Spring Data Vault — поддержка Vault, построенная поверх Spring Data KeyValue;
- Spring Data YugabyteDB — поддержка YugabyteDB.

7.8. Модуль Spring Data JPA

7.8.1. Подключение зависимостей

Spring Data JPA — основной модуль, использующийся в проекте virtualpets-server-springboot для доступа к данным (листинг 7.76).

Листинг 7.76. pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

7.8.2. Интерфейс *Repository*

Основным интерфейсом Spring Data JPA является обобщенный интерфейс `Repository` (листинг 7.77).

Листинг 7.77. Repository.java

```
@Indexed
public interface Repository<T, ID> {
}
```

Параметры типа обобщенного интерфейса `Repository`:

- `T` — тип сущности;
- `ID` — тип первичного ключа сущности.

Сам интерфейс `Repository` не содержит методов. Главные цели этого интерфейса — захват типа сущности и типа первичного ключа, а также облегчение поиска дочерних интерфейсов при сканировании бинов.

Дочерние интерфейсы расширяют интерфейс `Repository` и добавляют свои методы.

7.8.3. Интерфейс *CrudRepository*

Дочерний интерфейс `CrudRepository` расширяет интерфейс `Repository` и добавляет:

- `Optional<T> findById(ID id);` — метод загрузки сущности по первичному ключу;
- `Iterable<T> findAll();` — метод загрузки всех сущностей из базы данных;
- `long count();` — метод подсчета количества сущностей в базе данных;
- `void deleteById(ID id);` — метод удаления сущности по первичному ключу;
- `<S extends T> S save(S entity);` — метод сохранения новой сущности в базе данных или изменения уже существующей;
- другие методы — например, получения, сохранения и удаления сущностей по списку первичных ключей и т. д.

Интерфейс `CrudRepository` для возвращения коллекций использует `Iterable`, но существует также интерфейс `ListCrudRepository`, вместо `Iterable` возвращающий интерфейсы `List`, с которыми может быть проще работать.

7.8.4. Интерфейс *JpaRepository*

Проект виртуальных питомцев virtualpets-server-springboot использует JPA для слоя постоянства. Spring Data JPA содержит специальный интерфейс `JpaRepository`, расширяющий интерфейс `ListCrudRepository` и добавляющий методы:

- `getReferenceById` — получение прокси-сущности по первичному ключу без обращения к базе данных;
- `flush` — применение всех изменений в базе данных;

- пакетного удаления сущностей;
- поиска сущностей по образцу и поиска сущностей по образцу с сортировкой результата.

В качестве примера интерфейса репозитория и работы с ним рассмотрим интерфейс `LevelDao` (листинг 7.78).

Листинг 7.78. LevelDaoImpl.java

```
package ru.urvanov.virtualpets.server.dao;

import org.springframework.data.jpa.repository.JpaRepository;
import ru.urvanov.virtualpets.server.dao.domain.Level;

public interface LevelDao extends JpaRepository<Level, Integer> {
}
```

Сам по себе интерфейс `LevelDao` не содержит никаких новых методов — все методы наследуются из `JpaRepository` и его суперклассов.

> ***Не нужно создавать реализацию***
>
> Обратите внимание, что при использовании Spring Data JPA описываются только интерфейсы репозиториев. Реализации этих интерфейсов создаются сами внутри Spring Data JPA.

При использовании созданного репозитория необходимо внедрить его интерфейс как бин и использовать его так, будто его реализация уже существует (листинг 7.79).

Листинг 7.79. TownServiceImpl.java

```
@Service
public class TownServiceImpl ...
    @Autowired
    private LevelDao levelDao;
...
    @Override
    @Transactional(rollbackFor = ServiceException.class)
    public GetTownInfoResult getTownInfo(UserPetDetails userPetDetails)
            throws ServiceException {
...
        Optional<Level> nextLevelLeague = levelDao.findById(pet.getLevel().getId() + 1);
```

7.8.5. Методы репозиториев

В интерфейсы, расширяющие `Repository` и его наследников, можно добавлять свои методы (листинг 7.80).

Листинг 7.80. RoomDao.java

```
package ru.urvanov.virtualpets.server.dao;

import java.util.Optional;

import org.springframework.data.repository.ListCrudRepository;
import org.springframework.transaction.annotation.Transactional;

import ru.urvanov.virtualpets.server.dao.domain.Room;

@Transactional(readOnly = true)
public interface RoomDao extends ListCrudRepository<Room, Integer> {
    Optional<Room> findByPetId(Integer petId);
}
```

Интерфейс `RoomDao` расширяет интерфейс `ListCrudRepository` и добавляет один новый метод `findByPetId`. Обратите внимание, что реализация метода нигде не описана — в интерфейсе `RoomDao` присутствует только описание метода без самого тела метода. Реализация интерфейса `RoomDao` и реализация метода `findByPetId` будут созданы самим Spring Data во время выполнения.

Метод `findByPetId` производит поиск сущности `Room` по полю `petId`, которое объявлено в сущности `Room` как обычное числовое поле. Конкретно в рассматриваемом случае нет особого смысла именно в методе `findByPetId`, т. к. поле `petId` по совместительству является и первичным ключом сущности `Room` (листинг 7.81).

Листинг 7.81. Room.java

```
@Id
private Integer petId;
```

Метод `findById`, унаследованный от расширяемого интерфейса `CrudRepository` интерфейсом `ListCrudRepository`, делает абсолютно то же самое, что и метод `findByPetId`.

Особое внимание стоит обратить на название метода, которое состоит из двух частей:

1. `findBy` — по этому началу названия метода Spring Data определяет, что метод возвращает данные типа `Room`, для которого создан репозиторий, а не считает количество записей, проверяет существование, удаляет, сохраняет и т. д.
2. `PetId` — поиск производится по полю `petId` в сущности `Room`, значение поля передается в качестве параметра метода.

Метод возвращает `Optional`, который либо содержит экземпляр сущности `Room`, найденный по условию, либо не содержит значения. Вместо `Optional` допускается возвращать сам тип `Room`, для которого создан репозиторий (листинг 7.82).

Листинг 7.82. RoomDao.java

```
Room findByPetId(Integer petId);
```

Рекомендуется использовать Optional

Для методов, возвращающих единственный экземпляр сущности, рекомендуется использовать `Optional`, т. к. это облегчает обработку ситуаций, в которых сущность не найдена.

7.8.6. Аннотация @Transactional

Методы стандартных интерфейсов `CrudRepository`, `ListCrudRepository` и `JpaRepository` получают аннотацию `@Transactional` из своей реализации `SimpleJpaRepository`. Методы чтения данных аннотированы как `@Transactional(readOnly = true)`, методы изменения данных аннотированы `@Transactional`.

Аннотацию @Transactional создаваемых методов необходимо проставлять самостоятельно

При создании своих методов необходимо иметь в виду, что эти методы не получают никакой настройки транзакционности по умолчанию. Самый надежный метод исправления этого — аннотировать весь создаваемый интерфейс `@Transactional(readOnly = true)`, а для методов изменения данных — проставлять `@Transactional` на уровне методов.

Интерфейс `RoomDao` аннотирован `@Transactional(readOnly = true)`, поэтому единственный его метод: `findByPetId` — выполняется внутри транзакции с подсказкой «только для чтения».

7.8.7. Сортировка

Более сложный пример, возвращающий коллекцию с сортировкой, приведен в листинге 7.83.

Листинг 7.83. BookDao.java

```
package ru.urvanov.virtualpets.server.dao;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.transaction.annotation.Transactional;

import ru.urvanov.virtualpets.server.dao.domain.Book;

@Transactional(readOnly = true)
public interface BookDao extends JpaRepository<Book, String> {
    List<Book> findByOrderByBookcaseLevelAscBookcaseOrderAsc();
}
```

Наименование метода `findByOrderByBookcaseLevelAscBookcaseOrderAsc` состоит из следующих частей:

1. `findBy` — по этой части Spring Data понимает, что метод ищет значения по условию.
2. `OrderBy` — результат необходимо вернуть в отсортированном виде.

3. `BookcaseLevel` — сортировка результата по полю `bookcaseLevel`.
4. `Asc` — сортировка по `bookcaseLevel` будет происходить в порядке возрастания.
5. `BookcaseOrder` — сортировка результата дополнительно по `bookcaseOrder` (после сортировки по `bookcaseLevel`).
6. `Asc` — сортировка по `bookcaseOrder` будет происходить в порядке возрастания.

При указании порядка сортировки по полю принимаются значения `Asc` (по возрастанию) и `Desc` (по убыванию). В примере `findByOrderByBookcaseLevelAscBookcaseOrderAsc` используется только сортировка по возрастанию.

Метод `findByOrderByBookcaseLevelAscBookcaseOrderAsc` возвращает список экземпляров `Book`. Для методов, возвращающих несколько значений, допустимо использовать:

- `Iterator<T>` — итератор, позволяющий пройтись по результату запроса;
- `Collection<T>` — базовый класс коллекций;
- `List<T>` — для упорядоченного списка результатов;
- `Set<T>` — множество неупорядоченных уникальных элементов;
- `Stream<T>` — Java 8 Stream;
- `Streamable<T>` — интерфейс из Spring Data, расширяющий Iterable.

Пример сервера виртуальных питомцев virtualpets-server-springboot использует интерфейс `List<T>` для методов, возвращающих коллекции, т. к. с ним проще работать, и для согласованности с `ListCrudRepository` и `JpaRepository`, использующими `List<T>`.

7.8.8. Репозиторий *UserDao*

Интерфейс-репозиторий `UserDao` (листинг 7.84) содержит больше методов, чем рассмотренные до этого интерфейсы.

Листинг 7.84. UserDao.java

```
package ru.urvanov.virtualpets.server.dao;

import java.time.OffsetDateTime;
import java.util.List;
import java.util.Optional;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.transaction.annotation.Transactional;

import ru.urvanov.virtualpets.server.dao.domain.User;

@Transactional(readOnly = true)
public interface UserDao extends JpaRepository<User, Integer> {
    Optional<User> findByLogin(String login);
```

```
    List<User> findByActiveDateAfter(OffsetDateTime offsetDateTime);

    Optional<User> findByLoginAndEmail(String name, String email);

    Optional<User> findByRecoverPasswordKey(String recoverKey);

    default Page<User> findLastRegisteredUsers(int page, int pageSize) {
        return this.findAll(PageRequest.of(page, pageSize,
                Sort.by("registrationDate").descending()));
    }
}
```

- Метод `UserDao#findByLogin` работает аналогично рассмотренному ранее методу `RoomDao#findByPetId`.
- Метод `findByActiveDateAfter` ищет сущности по полю `activeDate` и возвращает список сущностей, в которых значение поля `activeDate` содержит более позднюю дату и время, чем значение переданного параметра. Наименование метода `findByActiveDateAfter` состоит из следующих частей:
  1. `findBy` — метод ищет сущности по условию.
  2. `ActiveDate` — фильтрация по полю `activeDate`.
  3. `After` — ищутся сущности, в которых значение поля `activeDate` позже, чем переданное в параметр метода.
- Метод `findByLoginAndEmail` показывает пример метода поиска сущностей по нескольким полям. Наименование `findByLoginAndEmail` состоит из следующих частей:
  1. `findBy` — метод ищет сущность по условию.
  2. `Login` — поиск сущности по полю `login`, значение которого передается в параметр метода.
  3. `And` — логическое «И» для поиска по двум условиям.
  4. `Email` — поиск сущности по полю `password`, значение которого передается в параметр метода.
  5. Метод `findByLoginAndEmail` возвращает `Optional<User>`, в которой значение поля `login` и пароль равны значениям, переданным в параметры метода. Фильтрация производится аналогично клаузе `WHERE login = ?1 AND email = ?2`.

7.8.9. Постраничная разбивка

Метод `User#findLastRegisteredUsers` — это пример метода с постраничной разбивкой результата (листинг 7.85).

Листинг 7.85. UserDao.java

```
default Page<User> findLastRegisteredUsers(int page, int pageSize) {
    return this.findAll(PageRequest.of(page, pageSize,
            Sort.by("registrationDate").descending()));
}
```

Метод `findLastRegisteredUsers` — это default-метод, который принимает в качестве параметра номер страницы и размер страницы. Внутри себя он вызывает метод `findAll`, унаследованный интерфейсом `JpaRepository` от интерфейса `PagingAndSortingRepository`.

Метод `PagingAndSortingRepository#findAll` принимает в качестве параметра `Pageable`:

```
Page<T> findAll(Pageable pageable);
```

В качестве реализации `Pageable` передается экземпляр класса `PageRequest`, созданный с помощью фабричного метода `PageRequest#of`:

```
PageRequest.of(page, pageSize, Sort.by("registrationDate").descending()
```

В этом примере создается экземпляр `PageRequest` выборки данных для страницы `page`, размера страницы `pageSize` и с сортировкой записей по убыванию по полю `registrationDate`.

> ***При постраничной разбивке важно использовать сортировку***
>
> Нельзя пытаться получить данные с постраничной разбивкой, не накладывая при этом сортировку на исходные данные, т. к. в этом случае порядок записей не будет гарантирован ничем. Без сортировки порядок результирующих записей может меняться при последующих запросах, что приведет к тому, что постраничная разбивка будет выдавать странные результаты.

Метод `PagingAndSortingRepository#findAll` возвращает экземпляр `Page<T>`, который, кроме самих данных, содержит дополнительный метод `nextPageable` для получения `PageRequest` следующей страницы, номера текущей страницы, размера страницы, суммарного количества страниц и суммарного количества записей.

7.8.10. Именованные запросы

Интерфейс `PetDao` является самым большим и сложным интерфейсом-репозиторием в проекте virtualpets-server-springboot.

Первый метод: `PetDao#findFullById` — приведен в листинге 7.86.

Листинг 7.86. PetDao.java

```
Optional<Pet> findFullById(Integer id);
```

Метод `findFullById` использует здесь именованный запрос. Имя именованного запроса по умолчанию определяется как имя сущности и имя метода, разделенные точкой. Для метода `PetDao#findFullById` имя именованного запроса будет `Pet.findFullById`.

Сами именованные запросы объявляются с помощью аннотации `@NamedQuery`, поставленной на классе сущности. Пример для именованного запроса `Pet.findFullById` приведен в листинге 7.87.

Листинг 7.87. Pet.java

```
@Entity
@Table(name = "pet")
// ...
@NamedQuery(name = "Pet.findFullById", query = """
        from Pet p
        left outer join fetch p.level l
        left outer join fetch p.hat hl
        left outer join fetch p.cloth cl
        left outer join fetch p.bow bl
        left outer join fetch p.user u
        left outer join fetch p.cloths c
        left outer join fetch p.books b
        left outer join fetch p.foods f
        left outer join fetch p.buildingMaterials bm
        left outer join fetch bm.buildingMaterial
        left outer join fetch p.drinks d
        left outer join fetch p.journalEntries je
        left outer join fetch p.achievements ach
        where p.id = :id""")
// ...
public class Pet ...
```

Именованный запрос `Pet.findFullById` соединяет все связанные таблицы, заполняя всю сущность `Pet` вместе со всеми связанными коллекциями и сущностями. Единственный параметр запроса — `id`, в который подставляется значение из единственного параметра метода `PetDao#findFullById`, имеющего идентичное имя.

Сопоставление параметра именованного запроса и параметра метода идет именно по имени. Проект virtualpers-server-springboot использует проект spring-boot-starter-parent в качестве родительского, поэтому классы компилируются с опцией `-parameters` из Java 8, что делает имена параметров методов доступными во время выполнения.

> ***Именованные запросы имеют приоритет над построением запроса по имени метода***
>
> Если существует именованный запрос с именем `<имя_сущности>.<имя_метода>`, то будет использоваться именованный запрос, даже если из имени метода можно построить другой запрос.

Следующий метод: `PetDao#findByUserId` — также задействует именованный запрос (листинг 7.88).

Листинг 7.88. PetDao.java

```
List<Pet> findByUserId(Integer userId);
```

Именованный запрос `Pet.findByUserId`, используемый методом `PetDao#findByUserId`, приведен в листинге 7.89.

Листинг 7.89. PetDao.java

```
@NamedQuery(name = "Pet.findByUserId", query = "from Pet p where p.user.id = :userId")
```

7.8.11. JPQL-запросы

Кроме именованных запросов, методы интерфейсов-репозиториев могут использовать JPQL-запросы, описанные в аннотации `@Query`, указанной на самом методе, как это сделано в методе `PetDao#findByIdWithBuildingMaterials` (листинг 7.90).

Листинг 7.90. PetDao.java

```
@Query("from Pet p left outer join p.buildingMaterials bm where p.id = ?1")
Optional<Pet> findByIdWithBuildingMaterials(Integer id);
```

7.8.12. Подсчет количества

Метод `PetDao#countByUserId` подсчитывает количество созданных пользователем питомцев (листинг 7.91).

Листинг 7.91. PetDao.java

```
long countByUserId(Integer userId);
```

Имя метода `countByUserId` состоит из двух частей, по которым и строится результирующая выборка количества питомцев:

1. `countBy` — указывает Spring Data, что метод возвращает результат агрегирующей функции `count`, подсчитывающей количество.
2. `User` — фильтрация по полю `user`.
3. `Id` — фильтрация по полю `id` внутри поля `user`.

`UserId` в методе `countByUserId` обозначает не поле `userId`, а поле `id` внутри поля `user`. Иногда указание подобных вложенных полей порождает неоднозначность, т. к. `UserId` может указывать и на поле `userId`, и на вложенное в `user` поле `id`. Для исключения подобной неоднозначности используется символ подчеркивания, разделяющий поле и вложенное в него поле (листинг 7.92).

Листинг 7.92. PetDao.java

```
long countByUser_Id(Integer userId);
```

7.8.13. Механизм Criteria API

Интерфейс `JpaSpecificationExecutor` содержит методы, позволяющие использовать Criteria API для построения запросов. `PetDao` расширяет интерфейс `JpaSpecificationExecutor` и использует его метод `count` в методе получения количества новых записей в дневнике питомца.

Метод `JpaSpecificationExecutor#count` принимает интерфейс `org.springframework.data.jpa.domain.Specification` в качестве параметра. `Specification` имеет всего один метод — `toPredicate` (листинг 7.93).

Листинг 7.93. Specification.java

```
public interface Specification<T> {
  Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
          CriteriaBuilder builder);
}
```

Проще всего создать статический метод, который возвращает нужный экземпляр `Specification`, и использовать его (листинг 7.94).

Листинг 7.94. PetDao.java

```
private static Specification<Pet>
        getPetNewJournalEntriesSpecification(Integer petId) {
    return (rootPet, criteriaQuery, criteriaBuilder) -> {
        MapJoin<Pet, JournalEntryId,
                PetJournalEntry> joinPetJournalEntries = rootPet
                        .join(Pet_.journalEntries, JoinType.LEFT);
        return criteriaBuilder.and(
                criteriaBuilder
                        .and(criteriaBuilder.equal(
                                joinPetJournalEntries.get(
                                        PetJournalEntry_.readed),
                                false)),
                criteriaBuilder.equal(rootPet.get(Pet_.id), petId));
    };
}
```

Метод получения количества новых записей в дневнике питомца, использующий созданный ранее метод `#getPetNewJournalEntriesSpecification`, приведен в листинге 7.95.

Листинг 7.95. PetDao.java

```
default long getPetNewJournalEntriesCount(Integer petId) {
    return this.count(PetDao.getPetNewJournalEntriesSpecification(petId));
}
```

Метод `PetDao#findLastCreatedPets` возвращает список созданных питомцев, упорядоченный по дате создания от самых последних к самым ранним, с постраничной разбивкой (листинг 7.96).

Листинг 7.96. PetDao.java

```
default Page<Pet> findLastCreatedPets(int page, int pageSize) {
    return this.findAll(PageRequest.of(page, pageSize,
            Sort.by("createdDate").descending()));
}
```

Работает метод `PetDao#findLastCreatedPets` в плане постраничной разбивки аналогично методу `User#findLastRegisteredUsers`, рассмотренному в *разд. 7.8.9*.

7.8.14. Графы сущностей

Остальные методы интерфейса-репозитория `PetDao` используют графы сущностей и именованный запрос `Pet.findById` (листинг 7.97).

Листинг 7.97. Pet.java

```
@NamedQuery(name = "Pet.findById", query = "from Pet p where p.id = :id")
```

Сами графы сущностей описываются с помощью аннотации `@NamedEntityGraph` над классом сущности (листинг 7.98).

Листинг 7.98. Pet.java

```
@NamedEntityGraph(name = "pet.buildingMaterials",
        attributeNodes = @NamedAttributeNode(
                value = "buildingMaterials",
                subgraph = "pet.buildingMaterials.buildingMaterial"
        ),
        subgraphs = @NamedSubgraph(
                name = "pet.buildingMaterials.buildingMaterial",
                attributeNodes = @NamedAttributeNode("buildingMaterial")
        )
)
```

Используемый методом интерфейса-репозитория именованный граф указывается с помощью аннотации `@EntityGraph`, в котором указывается имя графа (листинг 7.99).

Листинг 7.99. PetDao.java

```
@EntityGraph("pet.buildingMaterials")
@Query(name = "Pet.findById")
Optional<Pet> findByIdWithFullBuildingMaterials(Integer id);
```

7.9. Управление транзакциями

Транзакции уже частично описаны в *главе 6* и ряде разделов этой главы ранее. Здесь же мы рассмотрим особенности транзакций и атрибуты аннотации `@Transactional`, которые необходимо знать для полноценной работы с ними.

- Атрибут `propagation` аннотации `@Transactional` из пакета `org.springframework.transaction.annotation` используется для управления распространением транзакции. Он принимает значения из перечисления `org.springframework.transaction.annotation.Propagation`:
  - `REQUIRED` — используется текущая транзакция или создается новая транзакция, если транзакции еще нет;

- SUPPORTS — используется текущая транзакция. Если транзакции нет, то метод выполняется вне какой-либо транзакции;
- MANDATORY — используется текущая транзакция. Если транзакции нет, то бросается исключение;
- REQUIRES_NEW — создается новая транзакция. Текущая транзакция, если она есть, приостанавливается;
- NOT_SUPPORTED — метод выполняется вне транзакции. Текущая транзакция, если она есть, приостанавливается;
- NEVER — метод выполняется вне транзакции. Бросается исключение, если вызов произошел в какой-либо транзакции;
- NESTED — метод выполняется во вложенной транзакции.

Значение атрибута `propagation` по умолчанию: `REQUIRED`, т. е. используется текущая транзакция, а если ее нет, то создается новая.

- Атрибут `isolation` аннотации `@Transactional` служит для указания уровня изоляции транзакции. Всего существуют четыре уровня — все они (а также значение `DEFAULT`, которое использует уровень изоляции базы данных по умолчанию) приведены в перечислении `org.springframework.transaction.annotation.Isolation`:
  - DEFAULT — используется уровень изоляции базы данных, принятый по умолчанию;
  - READ_UNCOMMITED («грязное» чтение, неповторяющееся чтение, чтение фантомов) — транзакция видит изменения, сделанные в других транзакциях, даже если они еще не были «закоммичены» в базе данных;
  - READ_COMMITED (неповторяющееся чтение и чтение фантомов) — транзакция видит изменения, сделанные в других транзакциях, если они уже были «закоммичены»;
  - REPEATABLE_READ (чтение фантомов) — транзакция не видит «незакоммиченные» изменения, сделанные в других транзакциях.

    Дополнительно *исключены* ситуации, при которых одна транзакция считывает строку, затем другая транзакция меняет считанную первой транзакцией строку, а первая транзакция снова считывает строку и видит «закоммиченные» изменения второй транзакции. При этом первая транзакция может по условию `WHERE` получить строки, которые были добавлены другими транзакциями (фантомы), т. е. получить больше строк, чем при первом выполнении запроса с тем же условием;
  - SERIALIZABLE — в дополнение к `REPEATABLE_READ` исключены ситуации, когда первая транзакция считывает строки по какому-либо условию, а затем при повторном считывании видит дополнительные строки, которые были добавлены в другой транзакции.

    Значение атрибута `isolation` по умолчанию: `Isolation.DEFAULT`.

- Атрибут `timeout` позволяет указать максимальное время выполнения транзакции.
- Атрибут `readOnly`, установленный в `true`, передает подсказку, что в транзакции не будет производиться изменение данных, — это позволит использовать некоторые оптимизации. Значение по умолчанию: `false`.
- Атрибут `rollbackFor` позволяет перечислить проверяемые исключения, при возникновении которых необходимо откатывать транзакцию.

> ***Откат транзакции***
>
> **По умолчанию транзакция откатывается только при возникновении `RuntimeException`, `Error` и их наследников, но не откатывается при возникновении проверяемых исключений.**

Пример использования транзакций в классе `RoomServiceImpl` проекта virtualpets-server-springframework приведен в листинге 7.100.

Листинг 7.100. RoomServiceImpl.java

```
@Override
@Transactional(rollbackFor = ServiceException.class)
public void buildRefrigerator(UserPetDetails userPetDetails,
        ru.urvanov.virtualpets.server.controller.api.domain.Point position)
        throws ServiceException {
    Pet pet = petDao
            .findByIdWithFoodsAndJournalEntriesAndBuildingMaterials(
                    userPetDetails.petId())
            .orElseThrow();
    Room room = roomDao.findByPetId(pet.getId()).orElseThrow();
    if (!pet.getJournalEntries()
            .containsKey(JournalEntryId.BUILD_REFRIGERATOR)) {
        throw new NotNowException();
    }

    final int DRY_FOOD_ADD_COUNT = 10;
    PetFood petDryFood = pet.getFoods().get(FoodId.DRY_FOOD);
    if (petDryFood == null) {
        petDryFood = new PetFood();
        petDryFood.setFood(foodDao.getReference(FoodId.DRY_FOOD));
        petDryFood.setPet(pet);
        petDryFood.setFoodCount(DRY_FOOD_ADD_COUNT);
        pet.getFoods().put(FoodId.DRY_FOOD, petDryFood);
    } else {
        petDryFood.setFoodCount(
                petDryFood.getFoodCount() + DRY_FOOD_ADD_COUNT);
    }

    Refrigerator refrigerator = refrigeratorDao.findFullById(1)
            .orElseThrow();
    petService.substractPetResources(pet, refrigerator);
    room.setRefrigerator(refrigerator);
    room.setRefrigeratorX(position.x());
    room.setRefrigeratorY(position.y());
```

```
    if (!pet.getJournalEntries()
            .containsKey(JournalEntryId.EAT_SOMETHING)) {
        PetJournalEntry newPetJournalEntry = new PetJournalEntry();
        newPetJournalEntry.setCreatedAt(OffsetDateTime.now(clock));
        newPetJournalEntry.setPet(pet);
        newPetJournalEntry
                .setJournalEntry(JournalEntryId.EAT_SOMETHING);
        newPetJournalEntry.setReaded(false);
        pet.getJournalEntries().put(
                newPetJournalEntry.getJournalEntry(),
                newPetJournalEntry);
        petService.addExperience(pet, 1);
    }
}
```

Здесь метод `buildRefrigerator` может бросить наследников исключения `ServiceException`, если у питомца недостаточно ресурсов для постройки холодильника либо ему еще не разрешено строить холодильник, т. к. нет соответствующего задания в дневнике. В обоих случаях необходимо откатить все изменения внутри транзакции, для чего исключение `ServiceException` указано в атрибуте `rollbackFor`.

Атрибут `noRollbackFor` позволяет указать исключения, при возникновении которых *не* нужно откатывать транзакцию.

7.10. Резюме

Spring позволяет интегрировать приложение с базами данных различных типов. В большинстве случаев используется реляционная база данных. В связи со сложившейся на текущий момент ситуацией в качестве базы данных зачастую выбирается PostgreSQL.

Интеграция Liquibase и Flyway облегчает поддержку схемы базы данных в актуальном состоянии с помощью миграций.

Spring позволяет работать с базами данных на уровне SQL-запросов с помощью классов `JdbcTemplate` и `JdbClient`, входящих в состав Spring JDBC.

Обширная поддержка JPA и Hibernate, а также проект Spring Data обеспечивают работу с данными на уровне объектов Java.

В большинстве современных проектов используется подход с Spring Data JPA. При этом активно применяются запросы JPQL, когда возможностей построения запросов на основе имени метода недостаточно.

Графы сущностей наравне с именованными запросами позволяют выбирать только необходимые данные, избегая лишних соединений таблиц в генерируемых SQL-запросах и лишних выборок данных.

ГЛАВА 8

Хранение настроек приложения

8.1. Файлы «пропертей»

В *разд 5.1.5* описан способ хранения настроек проекта virtualpets-server-springframework в property-файлах с их переключением для разных профилей.

Например, для профиля `development` используются файлы, прописанные в следующем файле «пропертей» (листинг 8.1).

Листинг 8.1. Файл properties.xml

```
<beans profile = "development">
    <context:property-placeholder
        location = "classpath:application_dev.properties"
        file-encoding = "utf-8"
        ignore-unresolvable = "true" />
    <context:property-placeholder
        location = "classpath:mail_dev.properties"
        file-encoding = "utf-8"
        ignore-unresolvable = "true" />
</beans>
```

Значения из файлов «пропертей» внедряются в бин либо в XML-конфигурации с помощью конструкции `${}` (листинг 8.2)...

Листинг 8.2. Файл root-context.xml

```
<bean id = "templateMessage"
        class = "org.springframework.mail.SimpleMailMessage">
    <property name = "from"
    value = "${virtualpets-server-springframework.mail.from}" />
    <property name = "subject" value = "Recover password" />
</bean>
```

...либо с помощью аннотации `@Value` и той же конструкции `${}` (листинг 8.3).

Листинг 8.3. PublicServiceImpl.java

```
@Value("${virtualpets-server-springframework.server.url}")
private String serverUrl;
```

Файлы application_dev.properties и mail_dev.properties содержат значения «пропертей», используемые на локальном компьютере разработчика для отладки.

Для профиля `production` применяются другие файлы, отличные от application_dev.properties и mail_dev.properties (листинг 8.4).

Листинг 8.4. Файл properties.xml

```
<beans profile = "production">
    <context:property-placeholder
        location = "classpath:application_prod.properties"
        file-encoding = "utf-8"
        ignore-unresolvable = "true" />
    <context:property-placeholder
        location = "classpath:mail_prod.properties"
        file-encoding = "utf-8"
        ignore-unresolvable = "true" />
</beans>
```

При этом в файлах application_prod.properties (листинг 8.5) и mail_prod.properties (листинг 8.6) подставляются другие значения — не те, что указывались в файлах application_dev.properties и mail_dev.properties для профиля `development`.

Листинг 8.5 Файл application_prod.properties

```
virtualpets-server-springframework.server.url=http://virtualpets.urvanov.ru/
virtualpets-server-springframework
virtualpets-server-springframework.play.url=http://virtualpets.urvanov.ru/virtualpets-client-js/
```

Листинг 8.6. Файл mail_prod.properties

```
virtualpets-server-springframework.mail.server=localhost
virtualpets-server-springframework.mail.port=587
virtualpets-server-springframework.mail.username=test
virtualpets-server-springframework.mail.password=test
```

Текущий активный профиль Spring описывается в файле src/main/webapp/WEB-INF/web.xml с помощью указания контекстного параметра `spring.profiles.active` (листинг 8.7).

Листинг 8.7. Файл web.xml

```
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>development</param-value>
</context-param>
```

8.2. Задание профиля и «пропертей» в Apache 8.3. Tomcat

Профиль можно переопределить в самом Apache Tomcat, поменяв это значение в файле context.xml, находящемся в подкаталоге conf от CATALINA_BASE (листинг 8.8).

Листинг 8.8. Файл context.xml

```
<?arameter
    name = "spring.profiles.active"
    value = "production"
    override = "false"/>
```

Обратите внимание на атрибут `override`, выставленный в `false`. Он указывает на то, что значение, прописанное в дескрипторе развертывания web.xml, *не* должно переопределять значение, содержащееся здесь.

Файл mail_prod.properties включает логин и пароль от сервера почты. Хранить настоящий логин и пароль в исходниках, а значит, и в системе контроля версий нельзя, поэтому в файле прописаны лишь заглушки значений. Реальные значения необходимо указывать в файле context.xml Apache Tomcat на самом сервере production с помощью тегов `Environment` (листинг 8.9).

Листинг 8.9. Файл context.xml

```
<Environment
    name = "virtualpets-server-springframework.mail.username"
    value = "realusername"
    type = "java.lang.String"
    override = "false" />
<Environment
    name = "virtualpets-server-springframework.mail.password"
    value = "realpassword"
    type = "java.lang.String"
    override = "false" />
```

Пример заполнения файла context.xml приведен в файле tomcat-config/context.xml исходников проекта virtualpets-server-springframework (листинг 8.10).

Листинг 8.10. Файл context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>

    <WatchedResource>WEB-INF/web.xml</WatchedResource>
    <WatchedResource>WEB-INF/tomcat-web.xml</WatchedResource>
    <WatchedResource>${catalina.base}/conf/web.xml</WatchedResource>

    <Resource
        auth = "Container"
        driverClassName = "org.postgresql.Driver"
```

```
        maxIdle = "10"
        maxTotal = "20"
        maxWaitMillis = "-1"
        name = "jdbc/virtualpetsDB"
        password = "postgres"
        type = "javax.sql.DataSource"
        url = "jdbc:postgresql://localhost:5432/postgres?currentSchema=virtualpets_server_
                                                                    springframework"
        username = "postgres"/>

    <!--<Environment
        name = "virtualpets-server-springframework.server.url"
        value = "http://localhost:8080/virtualpets-server-springframework"
        type = "java.lang.String"
        override = "false" />
    -->

    <Environment
        name = "virtualpets-server-springframework.mail.server"
        value = "realsmtphost"
        type = "java.lang.String"
        override = "false" />
    <Environment
        name = "virtualpets-server-springframework.mail.port"
        value = "587"
        type = "java.lang.Integer"
        override = "false" />
    <Environment
        name = "virtualpets-server-springframework.mail.username"
        value = "realusername"
        type = "java.lang.String"
        override = "false" />
    <Environment
        name = "virtualpets-server-springframework.mail.password"
        value = "realpassword"
        type = "java.lang.String"
        override = "false" />

    <Parameter
        name = "spring.profiles.active"
        value = "development"
        override = "false"/>
</Context>
```

8.3. Файлы «пропертей» в Spring Boot

Spring Boot считывает файлы application.properties, application.yml и application.yaml из следующих мест, которые обычно располагаются в classpath, но могут находиться и:

1. В корне classpath.
2. В пакете /config в classpath.

3. В текущем каталоге.
4. В подкаталоге config текущего каталога.
5. В непосредственных дочерних каталогах каталога config текущего каталога.

Список отсортирован в порядке приоритета — от самого низкого к самому высокому. Если файлы расположены в нескольких местах, то используется файл из места с наивысшим приоритетом.

Дополнительно к основному файлу application.yaml (расширение файла yml также допустимо) и файлу application.properties загружаются настройки из файлов application-<*имя-профиля*>.yaml и application-<*имя-профиля*>.properties соответственно.

Например, сервер виртуальных питомцев на Spring Boot хранит адрес сервера, версию протокола и адрес клиентской части в файле src/main/resources/application.yaml (листинг 8.11).

Листинг 8.11. Файл application.yaml

```
virtualpets-server-springboot:
  play:
    url: http://localhost:8081
  version: 0.21
  server:
   - url: http://localhost:8080
  mail:
    from: noemail@nowhere.com
    subject: Recover password
```

Аннотация `@Value` позволяет внедрять значения, считанные из этого файла в бины:

```
@Value("${virtualpets-server-springboot.version}")
private String version;
```

Аргумент командной строки `--spring.config.location` позволяет переопределить расположение файла с настройками вне jar-файла, указав:

```
java -jar virtualpets-server-springboot.jar
                                    --spring.config.location=file:/home/fedor/myconfig.yaml
```

Фрагмент `--spring.config.location` полностью заменяет значения из файла по умолчанию. Если вам нужно переопределить лишь часть свойств — например, только подключение к базе данных, то необходимо использовать фрагмент: `--spring.config.additional-location`:

```
java -jar virtualpets-server-springboot.jar
                            --spring.config.additional-location=file:/home/fedor/myconfig.yaml
```

8.4. Аннотация @ConfigurationProperties

Аннотация `@ConfigurationProperties` из Spring Boot, применяемая совместно с `@Configuration`, позволяет создать бин, в поля которого будут внедрены значения дочерних «пропертей» от указанного узла (листинг 8.12).

Листинг 8.12. VirtualpetsServerSpringBootProperties.java

```
package ru.urvanov.virtualpets.server.config;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;

@ConfigurationProperties("virtualpets-server-springboot")
@Configuration
public class VirtualpetsServerSpringBootProperties {

    private String version;

    private Server server;

    public class Server {
        private String url;

        public String getUrl() {
            return url;
        }

        public void setUrl(String url) {
            this.url = url;
        }
    }

    public String getVersion() {
        return version;
    }

    public void setVersion(String version) {
        this.version = version;
    }

    public Server getServer() {
        return server;
    }

    public void setServer(Server server) {
        this.server = server;
    }
}
```

В `@ConfigurationProperties` здесь передается корневой узел, дочерние «проперти» от которого будут внедрены в поля бина, — в нашем случае это узел `virtualpets-server-springboot`, совпадающий с именем проекта. В результате будет создан бин типа `VirtualpetsServerSpringBootProperties`, поля которого заполнят значения `version` и `servers` из файла application.yaml.

8.5. Проект Spring Cloud Config

Проект Spring Cloud Config позволяет хранить настройки проектов Spring Boot централизованно. Вместо того, чтобы каждому приложению подсовывать свой файл application.yaml с настройками, можно с помощью проекта Spring Cloud Bus получать настройки из централизованного хранилища, а также автоматически обнаруживать изменения в конфигурации и уведомлять об этом приложения.

Spring Cloud Bus использует брокер сообщений, наподобие Kafka или RabbitMQ, для уведомления об изменении состояния. Сами сервисы на Spring Boot аналогично получают от брокера сообщений через Spring Cloud Bus сообщения `RefreshRemoteApplicationEvent`, сигнализирующие о том, что им необходимо перечитать свои настройки. Общая схема всего процесса приведена на рис. 8.1.

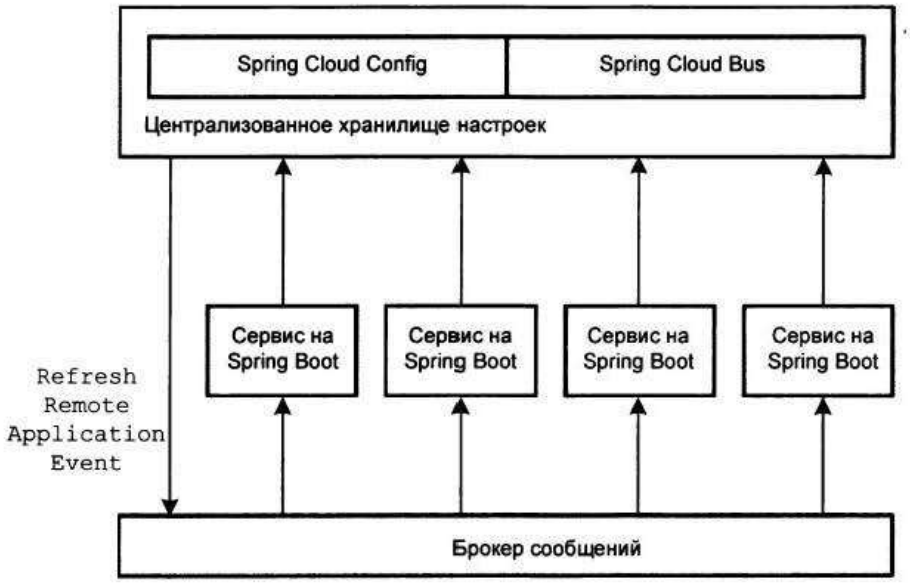


Рис. 8.1. Централизованное хранилище настроек Spring Cloud Config

Spring Cloud Config поддерживает разные хранилища:

- Git;
- файловую систему;
- Vault;
- AWS Secrets Manager;
- AWS Parameter Store;
- JDBC;
- Redis;
- AWS S3;
- CredHub;
- несколько разных хранилищ.

Проект virtualpets-server-springboot показывает пример использования Spring Cloud Config для централизованного хранения настроек:

- https://github.com/urvanov-ru/virtualpets-config — серверная часть на основе Spring Cloud Config. В дальнейшем в книге она обозначается как virtualpets-config —

т. е. всегда, когда далее упоминается virtualpets-config, имеется в виду проект в этом репозитории;

- https://github.com/urvanov.ru/virtualpets-config-repo — репозиторий с настройками для проекта virtualpets-server-springboot. В дальнейшем в книге он обозначается как virtualpets-config-repo.

Серверная часть virtualpets-config представляет собой обычный сервис на Spring Boot. Он скачивается и открывается аналогично основному проекту virtualpets-server-springboot.

Проект virtualpets-config использует зависимость `spring-cloud-config-server` как свою основную зависимость (листинг 8.13).

Листинг 8.13. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Обратите внимание, что версия `spring-cloud-config-server` не задана явно в самой зависимости. Версия берется из pom-файла, подключаемого в `dependencyManagement` (листинг 8.14)

Листинг 8.14. Файл pom.xml

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Значение `${spring-cloud.version}` задается в `properties` (листинг 8.15).

Листинг 8.15. Файл pom.xml

```
<properties>
    <java.version>17</java.version>
    <spring-cloud.version>2023.0.2</spring-cloud.version>
</properties>
```

Главный запускаемый файл сервиса VirtualpetsConfigApplication, помимо аннотации `@SpringBootApplication`, содержит аннотацию `@EnableConfigServer`, которая и создает сервер конфигурации из приложения (листинг 8.16).

Листинг 8.16. VirtualpetsConfigApplication.java

```
package ru.urvanov.virtualpets.config;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class VirtualpetsConfigApplication {

    public static void main(String[] args) {
        SpringApplication.run(VirtualpetsConfigApplication.class, args);
    }
}
```

По умолчанию сервис на Spring Boot запускается на порту 8080, но для сервиса конфигурации Spring Cloud Config стандартный порт — 8888, поэтому в конфигурационном файле application.properties необходимо прописать этот порт. Дополнительно файл содержит адрес репозитория Git с конфигурационными файлами (листинг 8.17).

Листинг 8.17. Файл application.properties

```
server.port: 8888
spring.application.name=virtualpets-config
spring.cloud.config.server.git.uri=https://github.com/urvanov-ru/virtualpets-config-repo
```

Репозиторий virtualpets-config-repo может содержать несколько файлов конфигурации. Конфигурация проекта, обращающегося к сервису конфигурации, определяется по имени обращающегося проекта. Имя указывается в конфигурации в свойстве `spring.application.name`. Адрес сервиса конфигурации задается с помощью настройки `spring.config.import`. Пример настройки на сервер конфигурации для virtualpets-server-springboot приведен в листинге 8.18.

Листинг 8.18. Файл application.yaml

```
spring:
  application:
    name: virtualpets-server-springboot
  config:
    import: optional:configserver:http://localhost:8888
```

Начальная часть адреса сервиса `optional:` указывает на то, что сервис конфигурации не обязателен — проект может запуститься без него. Если убрать эту часть, то при невозможности соединиться с сервисом virtualpets-config приложение virtualpets-server-springboot будет завершаться с ошибкой при запуске.

Поскольку здесь в `spring.application.name` указано имя `virtualpets-server-springboot`, то для игры виртуальных питомцев будет считываться файл virtualpets-server-springboot.yaml (листинг 8.19).

Листинг 8.19. Файл virtualpets-server-springboot.yaml

```
spring:
  mail:
    host: localhost
    port: 8888
    username: test
    password: test
virtualpets-server-springboot:
  play:
    url: http://localhost:8081
  version: 0.21
  server:
   - url: http://localhost:8080
  mail:
    from: springcloudmailfrom
    subject: Spring cloud mail subject
```

Значения, полученные с сервиса Spring Cloud Config (в нашем случае virtualpets-config), всегда имеют приоритет над значениями из файлов конфигураций, полученных другим путем.

Попытайтесь самостоятельно запустить сначала virtualpets-config, а затем virtualpets-server-springboot на своем компьютере.

При успешном запуске в логах virtualpets-server-springboot должны появиться строки вида:

```
2024-10-03T13:58:20.158+03:00  INFO 2090443 --- [virtualpets-server-springboot]
[main] o.s.c.c.c.ConfigServerConfigDataLoader   : Fetching config from server at :
http://localhost:8888
2024-10-03T13:58:20.158+03:00  INFO 2090443 --- [virtualpets-server-springboot]
[main] o.s.c.c.c.ConfigServerConfigDataLoader   : Located environment: name=virtualpets-
server-springboot, profiles=[default], label=null,
version=866bee892bc64a46e13db6a66a5c627c70220631, state=null
```

8.6. Резюме

Spring Framework предоставляет возможность хранения настроек в файлах «пропертей». Spring Cloud Config для Spring Boot позволяет вывести настройки нескольких сервисов в один централизованный репозиторий, а также автоматически проверять изменения в них.

ГЛАВА 9

Логирование

9.1. Хаос с библиотеками логирования

Java существует уже много лет. В этом языке программирования накопилось огромное количество древнего наследия, особенностей и запутывающих новичков «костылей». В первых версиях Java не было нормальной системы логирования, что привело к настоящему хаосу и порождению целого моря несовместимых друг с другом стандартов и библиотек, решающих одну и ту же задачу.

Вот перечень популярных библиотек логирования в Java:

- System.err — не совсем библиотека, а, скорее, простой способ логирования ошибок. Подходит только для лабораторных работ и простейших тестовых примеров;
- JUL (java.util.logging) — стандартный способ логирования, появившийся в Java 1.4. В реальности мало где используется, т. к. на момент его появления было уже слишком поздно, — он так и не стал действительно стандартным, потому что в момент его выхода уже существовало большое количество популярных библиотек логирования;
- Log4j 1.2 — популярная в свое время библиотека логирования. До сих пор может встретиться в старых проектах, но начинать на ней новые не имеет смысла. Log4j 1.2, начиная с 5 августа 2015 года не поддерживается;
- JCL (Apache Commons Logging, ранее Jakarta Commons Logging) — это даже не библиотека логирования, а API логирования, которое перенаправляет вызовы в лог в реальные библиотеки логирования. Реализация библиотеки логирования определяется во время запуска на основе библиотек логирования, доступных в `classpath`;
- Log4j 2 — современная библиотека логирования, в которой разделены API и реализация. Допускается использование API логирования из Log4j 2, а реализации из другой библиотеки;
- SLF4J (Simple Logging Facade for Java) — современное API логирования, позволяющее выбрать используемую библиотеку логирования во время сборки проекта;

- Logback — нативная реализация API SLF4J. Использование SLF4J совместно с Logback — это фактически негласный стандарт для современных приложений.

И это еще не всё...

Существуют и другие библиотеки логирования, как устаревшие и закончившие свой жизненный цикл, так и разрабатываемые до сих пор.

Выбор библиотеки логирования для проекта на Java всегда был довольно сложной задачей (рис. 9.1).

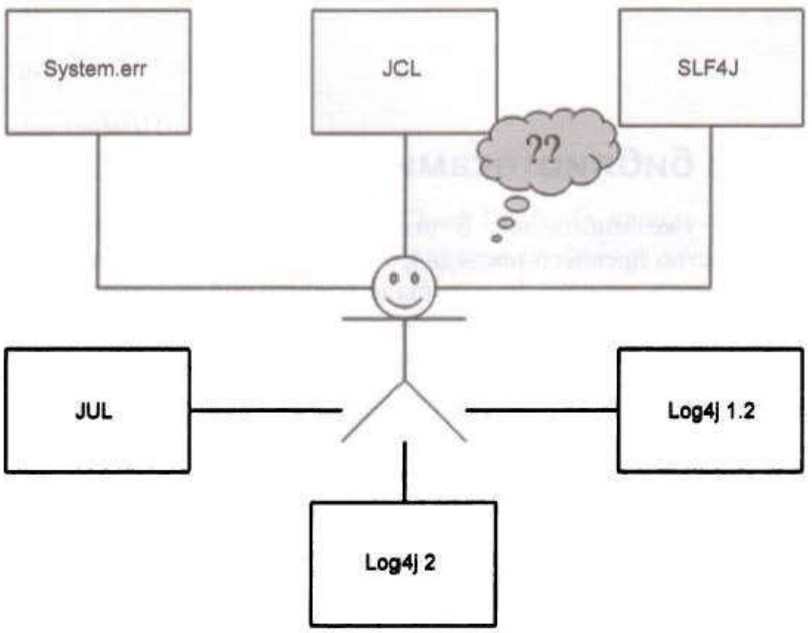


Рис. 9.1. Выбор библиотеки логирования в Java

9.2. Logback

На текущий момент наиболее распространенным подходом считается использование API SLF4j и его реализации Logback — именно этому подходу и следует Spring Framework в своих последних версиях.

Но что делать со всем старым кодом? Невозможно просто выбросить то огромное количество логгеров и библиотек, которое уже существует, — для них нужно подключить специальные зависимости, содержащие их API, но вместо реализации перенаправляющие вывод в SLF4j:

- jcl-over-slf4j.jar — содержит в себе API от Apache Commons Logging, но вместо его реализации просто перенаправляет все вызовы в SLF4j;
- log4j-over-slf4j.jar — содержит в себе API от Log4j, но вместо его реализации перенаправляет все вызовы в SLF4j;

- jul-to-slf4j.jar — содержит в себе обработчик (Handler) для JUL, который пишет все сообщения в SLF4j. Так как JUL встроен в JDK, то заменить его, как в случае Apache Commons Logging и Log4j, нельзя, именно поэтому просто добавляется новый Handler.

Кроме указанных зависимостей, перенаправляющих в SLF4j вызовы API других библиотек, существуют зависимости, которые реализуют API SLF4j:

- slf4j-log4j12.jar — перенаправляет вызовы SLF4j в Log4j12, т. е. позволяет использовать Log4j 1.2 в качестве реализации API SLF4j;
- slf4j-jdk14.jar — перенаправляет вызовы SLF4j в JUL, т. е. позволяет использовать JUL в качестве реализации API SLF4j;
- slf4j-nop.jar — просто игнорирует все вызовы SLF4j, что равносильно полному отключению логов;
- slf4j-simple.jar — перенаправляет вызовы SLF4j в System.err;
- slf4j-jcl.jar — перенаправляет вызовы SLF4j в Apache Commons Logging, т. е. позволяет использовать Apache Commons Logging в качестве реализации API SLF4j. Самое интересное в этом случае то, что Apache Commons Logging тоже является лишь оберткой с API, перенаправляющей выводы в другие реализации;
- logback-classic.jar — это библиотека логирования, напрямую реализующая API SLF4j. В современных приложениях, как правило, используют именно ее.

Итак, что нам нужно сделать, чтобы использовать связку Slf4j и Logback?

1. Подключить slf4j-api.
2. Подключить logback-classic.
3. Подключить jcl-over-slf4j, log4j-over-slf4j, чтобы сообщения логов от зависимостей, которые используют Apache Commons Logging и Log4j, перенаправлялись в SLF4j Можно еще подключить jul-to-slf4j, но это не рекомендуется, т. к. от него сильно падает производительность.
4. Из всех других подключаемых зависимостей убирать с помощью `exclude` в Maven зависимость от конкретной библиотеки логирования.
5. Настроить Logback.
6. Использовать slf4j-api для записи логов.

Хороший пример настройки SLF4J и его нативной реализации Logback можно найти на моем сайте[1].

При использовании Spring Framework вся необходимая настройка производится автоматически в зависимости от наличия SLF4J и Log4j 2 в `classpath`. Внутри себя Spring содержит модифицированную библиотеку spring-jcl, которая проверяет наличие Log4j 2 и SLF4J в `classpath` и использует первую из найденных как реализацию логирования.

---

[1] См. **https://urvanov.ru/2019/07/08/логирование-с-slf4j-и-logback/**.

Для того чтобы задействовать SLF4J и Logback в Spring Framework, как это принято в современных проектах, необходимо только добавить Logback в `classpath` без дополнительных зависимостей, перенаправляющих вывод логов. Вся остальная настройка с маршрутизацией логов из других библиотек будет произведена автоматически (листинг 9.1).

Листинг 9.1. Файл pom.xml

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.4.8</version>
</dependency>
```

В проектах на Spring Boot зависимость от Logback подтягивается автоматически при использовании стартеров.

Настройка Logback для Spring Framework производится в файле **src/main/resources/logback.xml**. Пример для virtualpets-server-springframework приведен в листинге 9.2.

Листинг 9.2. Файл logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{dd.MM.yyyy HH:mm:ss.SSS} [%thread] %-5level %logger{20} - %msg%n</pattern>
    </encoder>
  </appender>

  <appender name="FILE"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>virtualpets_server_springframework.log</file>
    <rollingPolicy
        class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!-- Новый файл каждый день -->
        <fileNamePattern>virtualpets_server_springframework.%d{yyyy-MM-dd}.log</fileNamePattern>

        <!-- Храним файлы логов 10 дней -->
        <maxHistory>10</maxHistory>

        <!-- Максимальный размер файлов лога 30 гигабайт -->
        <totalSizeCap>10GB</totalSizeCap>

    </rollingPolicy>
    <encoder>
      <pattern>%d{dd.MM.yyyy HH:mm:ss.SSS} [%thread] %-5level %logger{20} - %msg%n</pattern>
    </encoder>
</appender>

  <root level="INFO">
    <appender-ref ref="STDOUT" />
```

```
    <appender-ref ref="FILE" />
  </root>
</configuration>
```

В приведенном здесь примере файла logback.xml уровень логирования корневого логера в атрибуте `level` тега `root` выставлен в `INFO` — это означает, что в логе будут только предупреждения и ошибки.

К корневому логгеру добавлены два appender'а:

- `STDOUT` — пишет сообщения лога в стандартную консоль вывода;
- `FILE` — стандартный RollingFileAppender. Он пишет сообщения лога в файл virtualpets_server_springframework.log и обеспечивает создание нового файла каждый день с сохранением предыдущего лога в файле virtualpets_server_springframework.*ГГГГ-ММ-ДД*.log, где *ГГГГ-ММ-ДД* — год, месяц и число.

Для virtualpets-server-springboot вполне можно использовать тот же самый файл с теми же самыми настройками, но Spring Boot предоставляет облегченный способ настройки логирования через свойства в файле application.yaml (листинг 9.3).

Листинг 9.3. Файл application.yaml

```
logging:
  level:
    root : info
  file:
    name: ./log/virtualpets_server_springboot.log
```

Здесь явно указаны только уровень логирования и файл лога, но на самом деле уже автоматически настроены создание новых файлов каждый день и ограничение максимального размера файла в 10 Мбайт, по достижении которого начинается новый файл лога.

Если необходимо изменить настройки ротации файлов, то можно создать свой файл logback-spring.xml (обратите внимание на суффикс -spring в названии файла). Файл logback.xml тоже будет работать, но рекомендуется использовать именно файл с суффиксом, т. к. при этом будет правильно проинициализирована система логирования Spring Boot.

Вместо создания своего файла logback-spring.xml настройки ротации файлов для Logback и Spring Boot можно описать в настройках `logging.logback.rollingpolicy` файла application.yaml. Поддерживаются следующие настройки:

- `logging.logback.rollingpolicy.file-name-patterns` — шаблон файла для архивов при ротации файлов. По умолчанию: <logging.file.name>.%d{yyyy-MM-dd}.%i.gz.
- `logging.logback.rollingpolicy.clean-history-on-start` — очищать ли старые логи при запуске. По умолчанию: `false`;
- `logging.logback.rollingpolicy.max-file-size` — максимальный размер файла лога. По умолчанию: 10 Мбайт;

- `logging.logback.rollingpolicy.total-size-cap` — максимальный размер, который могут занять архивы логов перед их удалением.
- `logging.logback.rollingpolicy.max-history` — максимальное количество архивов логов. По умолчанию: 7.

9.3. Стек ELK

До текущего момента здесь описывалась только запись логов в файл. В варианте микросервисной архитектуры этим файлом станет файл внутри контейнера — например, в поде Kubernetes. Самих сервисов в этом случае может оказаться очень много, причем каждый из них может быть запущен в нескольких экземплярах, да еще каждый из экземпляров будет писать свой файл лога. В результате получится большое количество файлов логов, разбросанных по контейнерам, как показано на рис. 9.2. В этом случае разобраться, что произошло с каким-нибудь одним запросом, становится проблематично.

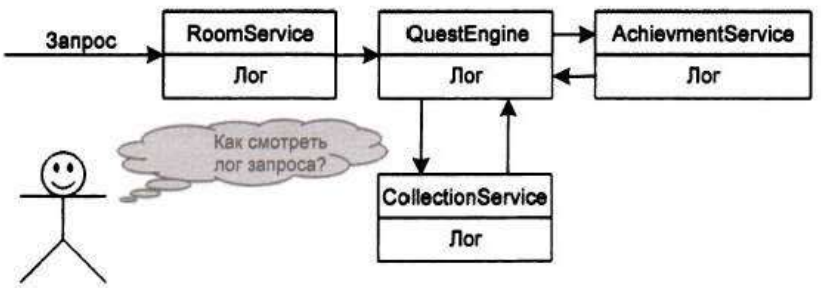


Рис. 9.2. Пример прохождения одного запроса по сервисам при микросервисной архитектуре

Для анализа пути прохождения запроса по сервисам необходимо объединить логи в какое-нибудь централизованное хранилище, позволяющее фильтровать сервисы по идентификаторам запросов и по сервисам. В качестве подобного хранилища в современном мире используется стек ELK:

1. Elasticsearch — выступает в качестве хранилища данных с инструментами поиска.
2. Logstash — конвейер обработки, фильтрации и нормализации логов, который принимает данные из нескольких логов и отправляет в Elasticsearch.
3. Kibana — интерфейс визуализации данных.

Отправкой данных в Logstash занимается обычно Filebeat, который работает на каждом из запущенных сервисов.

Результирующая схема работы с логами приведена на рис. 9.3, где показано множество сервисов, каждый из которых с помощью Filebeat отправляет логи в Logstash. Logstash, в свою очередь, после преобразования отправляет логи в Elasticsearch. Конечный пользователь смотрит логи запросов с помощью Kibana.

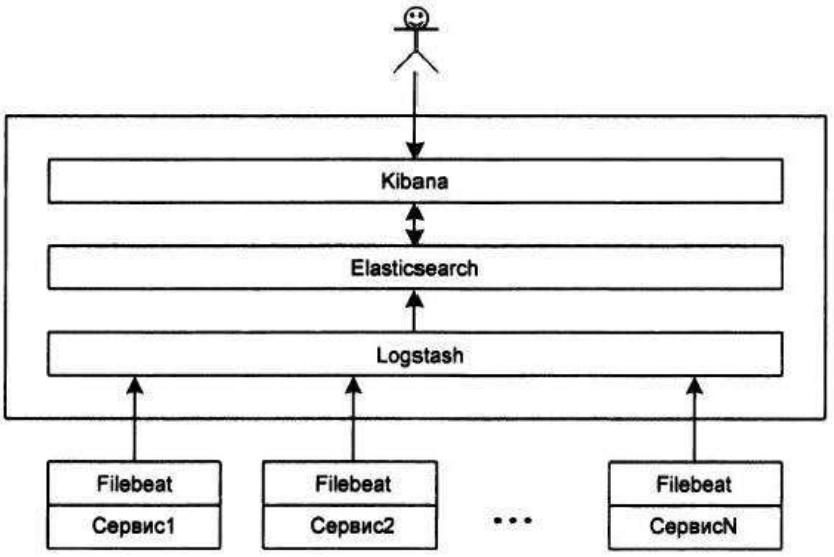


Рис. 9.3. Схема взаимодействия Filebeat и ELK

В файле docker-compose.yml, находящемся в каталоге docker проекта virtualpets-server-springboot, уже заложено подобное взаимодействие. Обратите внимание на настройки сервисов в этом файле (листинг 9.4).

Листинг 9.4. Файл docker-compose.yml

```
services:
 es01:
...
 kibana:
...
 filebeat01:
...
```

Именно в этих блоках настраиваются сервисы ELK и Filebeat, а также взаимодействие между ними.

Отдельного разговора заслуживает файл конфигурации filebeat.yml, в котором настраивается каталог log, из которого Filebeat забирает логи, — это каталог ingest_data внутри контейнера Filebeat, который настроен на каталог log из проекта virtualpets-server-springboot.

Отрывок содержимого файла filebeat.yml, настраивающего filebeat на хранение логов в каталоге ingest_data, приведен в листинге 9.5.

Листинг 9.5. Файл filebeat.yml

```
filebeat.inputs:
- type: filestream
  id: default-filestream
```

```
  paths:
    - ingest_data/*.log
```

Отрывок содержимого файла docker-compose.yml, настраивающего каталог log из проекта на каталог ingest_data внутри контейнера Filebeat, приведен в листинге 9.6.

Листинг 9.6. Файл docker-compose.yml

```
filebeat01:
...
   volumes:
...
     - "../log/:/usr/share/filebeat/ingest_data/"
```

Интерфейс Kibana доступен по адресу **http://localhost:5601**, логин и пароль для входа: `elastic` и `changeme` соответственно. После входа в Kibana отобразится его стартовая страница (рис. 9.4), в правой части которой необходимо выбрать блок **Analytics**, а на открывшейся странице (рис. 9.5) перейти по ссылке **Language: ES|QL**.

В результате откроется страница, отображающая логи сервиса virtualpets-server-springboot, которые Filebeat отправил в Logstash, а тот, в свою очередь, — в Elastic

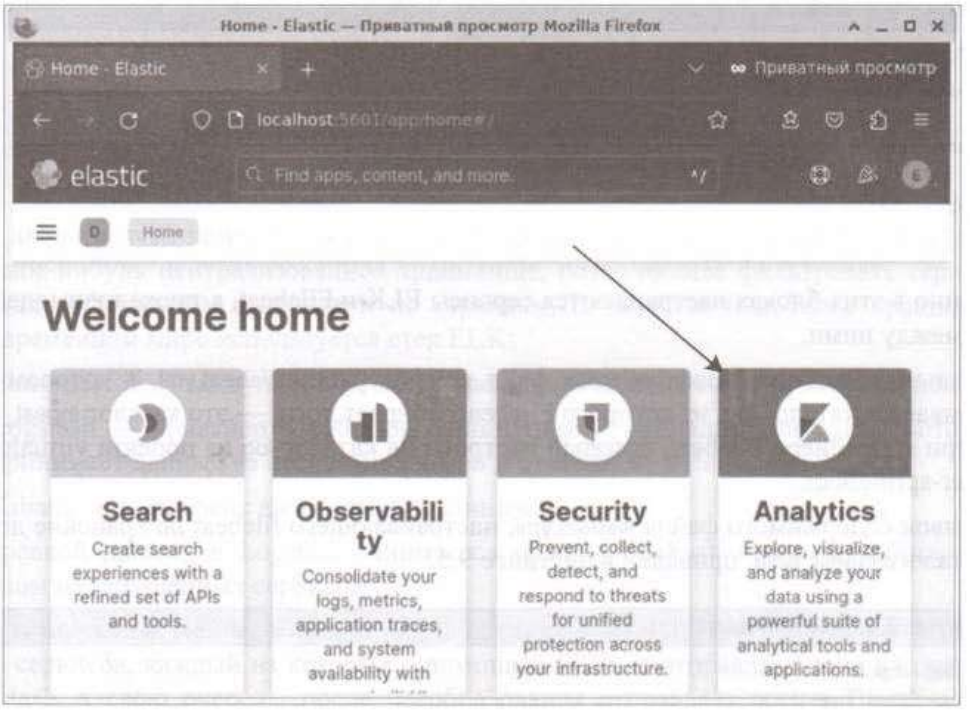


Рис. 9.4. Стартовая страница Kibana: выберите **Analytics**

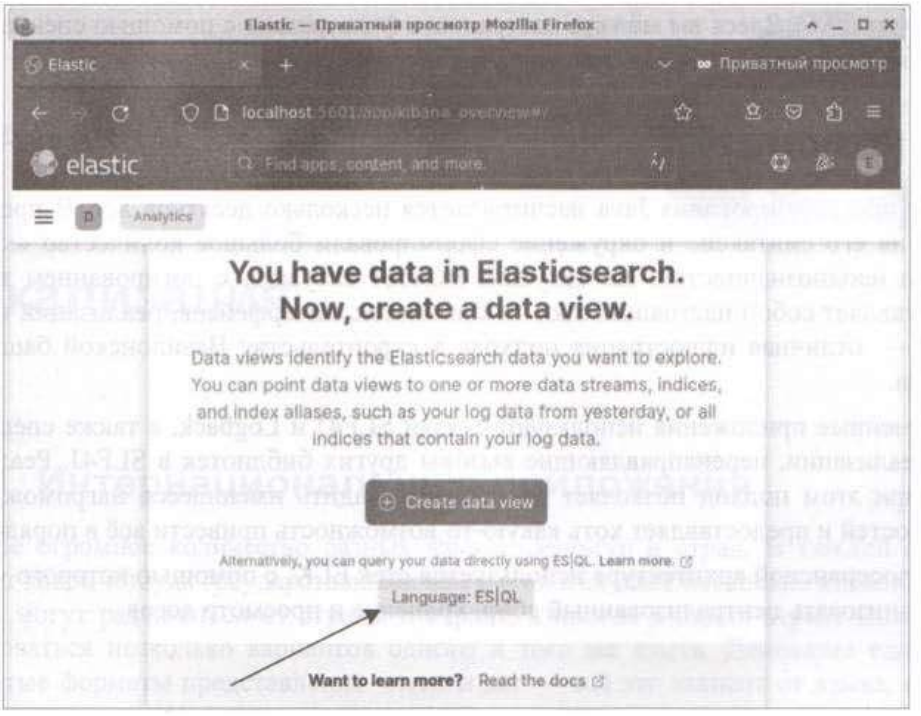


Рис. 9.5. В разделе Analytics перейдите по ссылке **Language: ES|QL**

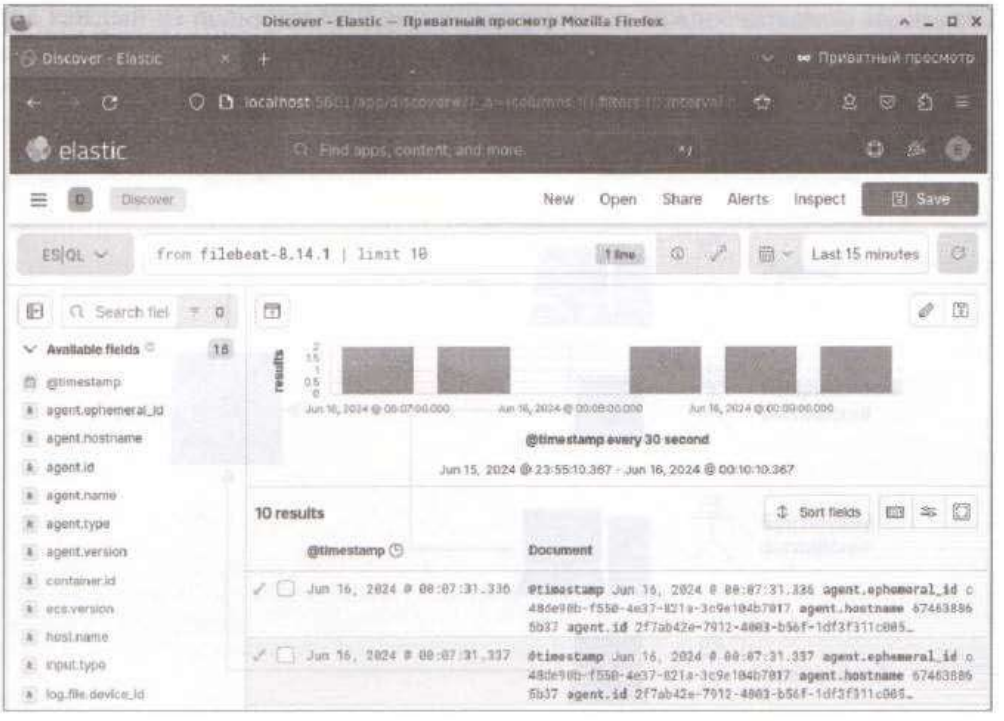


Рис. 9.6. Лог сервиса virtualpets-server-springboot в Kibana

search (рис. 9.6). Здесь вы можете настраивать фильтрацию с помощью специального языка, выбирать отображаемые поля и т. п.

9.4. Резюме

Языку программирования Java насчитывается несколько десятков лет. В процессе развития его синтаксис и окружение сформировали большое количество «костылей» и неоднозначностей. На текущий момент ситуация с логированием в Java представляет собой настоящий хаос из библиотек, интерфейсов, реализаций и подходов — отличная иллюстрация подхода к строительству Вавилонской башни из Библии.

Современные приложения используют связки SLF4J и Logback, а также специальные реализации, перенаправляющие вызовы других библиотек в SLF4J. Реализуемый при этом подход позволяет несколько сгладить имеющееся нагромождение сложностей и предоставляет хоть какую-то возможность привести всё в порядок.

В микросервисной архитектуре используется стек ELK, с помощью которого удается организовать централизованный сбор, хранение и просмотр логов.

ГЛАВА 10

Локализация

10.1. Интернациональные приложения

В мире огромное количество разных языков, культур и стран. В каждой стране принят какой-нибудь государственный язык, а иногда даже несколько языков. Сами языки могут различаться от страны к стране, а иногда в одной стране может использоваться несколько вариантов одного и того же языка. Денежные единицы, принятые форматы представления чисел и дат — всё это зависит от языка, народности, страны, региона и т. д.

Современные приложения часто поддерживают работу с несколькими языками, когда каждый из пользователей видит интерфейс на предпочитаемом им языке согласно настройкам системы либо на выбранном языке из списка доступных, как показано на рис. 10.1.

Интернационализация — адаптация продукта к использованию практически в любом месте, закладывание в нем условий, необходимых для добавления локализаций;

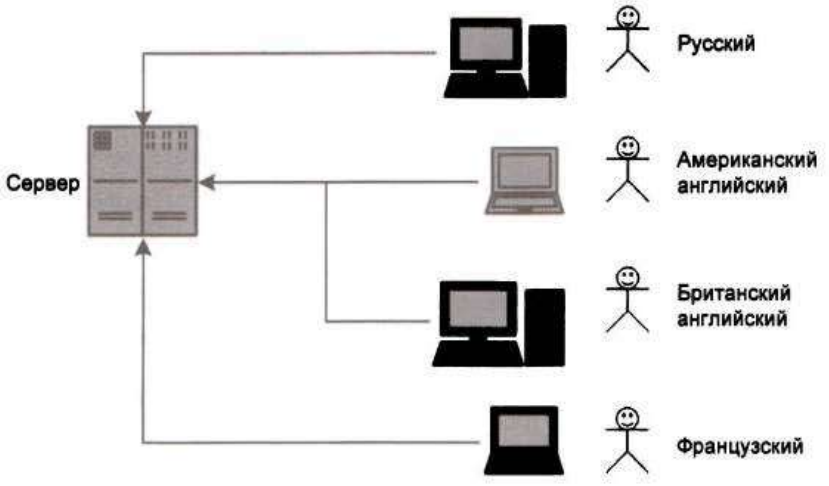


Рис. 10.1. Пользователи с разными предпочитаемыми локалями взаимодействуют с сервером

Локализация — адаптация продукта к использованию в какой-либо конкретной стране или регионе.

Для слова «интернационализация» в английском языке принято сокращение i18n — от слова internationalization.

Еще о локализации приложений

Локализация приложений на Java подробно рассматривалась в моей книге «Java. Состояние языка и его перспективы»[1]. Если вы ее еще не читали, то рекомендую ознакомиться с ней, — это может прояснить многие моменты на уровне языка Java.

Spring Framework использует похожий механизм, с реализациями как основанными на `java.util.ResourceBundle`, так и на других механизмах.

10.2. Интерфейс *MessageSource*

Spring Framework позволяет создавать интернациональные приложения с помощью интерфейса `org.springframework.context.MessageSource` и предоставляет три реализации этого интерфейса:

- `ResourceBundleMessageSource` — реализация на основе стандартного ResourceBundle из JDK;
- `ReloadableResourceBundleMessageSource` — реализация на основе Properties с возможностью повторного считывания изменившихся файлов на основе даты и времени их изменения;
- `StaticMessageSource` — простая реализация интерфейса `MessageSource`, позволяющая добавлять локализованные сообщения программно.

Локализованные сообщения программный код получает с помощью метода `getMessage` интерфейса `MessageSource`. Бин `messageSource` в virtualpets-server-springframework объявляется в файле /src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml (листинг 10.1).

Листинг 10.1. Файл servlet-context.xml

```
<beans:bean id = "messageSource"
    class =  "org.springframework.context.support.ReloadableResourceBundleMessageSource"
    p:basenames = "
        /WEB-INF/i18n/application,
        /WEB-INF/i18n/information,
        /WEB-INF/i18n/AchievementStrings"
    p:defaultEncoding = "UTF-8"
    p:fallbackToSystemLocale = "false" />
```

В атрибуте `basenames` здесь прописаны базовые имена файлов с локализованными сообщениями. Как можно видеть, приведенный пример использует три базовых имени:

---

[1] См. **https://bhv.ru/product/java-sostoyanie-yazyka-i-ego-perspektivy/**.

- /WEB-INF/i18n/application;
- /WEB-INF/i18n/information;
- /WEB-INF/i18n/AchievementStrings.

С помощью `defaultEncoding` указывается кодировка по умолчанию — `UTF-8`.

Атрибут `fallbackToSystemLocale` определяет, будет ли использоваться локаль системы по умолчанию, если в специфичных файлах сообщение с искомым кодом не будет найдено. Для серверных приложений всегда необходимо выставлять его в `false`, т. к. локаль сервера не имеет значения, имеет значение только локаль клиента. Обратите при этом внимание на рис. 10.1 — на сервере может быть язык по умолчанию китайский, немецкий или любой другой. Пользователю со знанием французского нет никакой пользы от немецкого, и уж точно он вряд ли поймет китайский.

Когда значение `fallbackToSystemLocale` принимается равным `false` — как в virtualpets-server-springframework, то при отсутствии значения в специфичных файлах локалей используется файл по умолчанию, без суффиксов с локалями. Обычно в этом файле содержатся строки сообщений с каким-либо из вариантов английского, т. к. этот язык на текущий момент считается стандартом для международного общения.

На основе базовых имен и текущей локали сообщения ищутся в следующих файлах в порядке очередности:

1. *<базовое_имя>*_*<язык>*_*<страна>*_*<вариант>*.properties;
2. *<базовое_имя>*_*<язык>*_*<страна>*.properties;
3. *<базовое_имя>*_*<язык>*.properties;
4. *<базовое_имя>*.properties — для `fallbackToSystemLocale = false`.

Пример игры с виртуальными питомцами содержит два языка: английский и русский. Английский язык — это язык по умолчанию. Если в системе у игрока в качестве предпочитаемого языка указан язык, отличный от русского, то используется английский язык.

Вот перечень файлов с переводами для virtualpets-server-springframework:

- AchievementStrings.properties;
- AchievementStrings_ru.properties;
- application.properties;
- application_ru.properties;
- information.properties;
- information_ru.properties.

Для локали `ru_RU` и базового имени файла /WEB-INF/i18n/application наличие локализованной строки проверяется в следующей очередности:

1. application_ru.properties;
2. application.properties.

Файлы, приведенные в basenames, рассматриваются последовательно в порядке записи. В качестве результата берется первое найденное сообщение для искомого кода.

Сами файлы с локализованными строками представляют собой обычные property-файлы — например src/main/webapp/WEB-INF/i18n/application.properties (листинг 10.2).

Листинг 10.2. Файл application.properties

```
virtualpets-server-springframework.app_name=Urvanov's Pretty Virtual Pets
virtualpets-server-springframework.site.home=Home
virtualpets-server-springframework.site.play=Play
virtualpets-server-springframework.site.recover_password=Recover password
virtualpets-server-springframework.site.recover_password_key=Recover password key
...
```

В листинге 10.3 приведен его вариант для русского языка — src/main/webapp/WEB-INF/i18n/application_ru.properties.

Листинг 10.3. Файл application_ru.properties

```
virtualpets-server-springframework.app_name=Urvanov's Pretty Virtual Pets
virtualpets-server-springframework.site.home=Главная
virtualpets-server-springframework.site.play=Играть
virtualpets-server-springframework.site.recover_password=Восстановление пароля
virtualpets-server-springframework.site.recover_password_key=Ключ
...
```

Бин messageSource внедряется из контейнера с помощью механизма внедрения зависимостей:

```
@Autowired
private MessageSource messageSource;
```

Локализованные сообщения формируются по коду сообщения с помощью одного из методов интерфейса MessageSource:

```
String getMessage(String code, Object[] args, String defaultMessage, Locale locale)
String getMessage(String code, Object[] args, Locale locale)
String getMessage(MessageSourceResolvable resolvable, Locale locale)
```

Параметры методов getMessage:

- code — код сообщения;
- args — подставляемые в текст локализованного сообщения параметры.

  Параметры в тексте сообщения выглядят так: {0}, {1,date}, {2,time}. В качестве args передается null, если параметров сообщения нет.

Интерфейс ApplicationContext расширяет интерфейс MessageSource, поэтому методы getMessage допустимо вызывать и на нем.

В большинстве случаев в использовании MessageSource напрямую нет необходимости, т. к. работа с локализованными строками происходит на уровне представления. JSP/

JSPX, Thymeleaf и другие шаблонизаторы предоставляют свои механизмы взаимодействия с бином `messageSource`.

Например, в листинге 10.4 показан вывод локализованного сообщения с кодом `virtualpets-server-springframework.site.server_info` на JSPX-странице.

Листинг 10.4. serverInfo.jspx

```
<spring:message
    code = "virtualpets-server-springframework.site.server_info"
    var = "server_info" />
<H1>${server_info}</H1>
```

Приведенный здесь код получает с помощью `messageSource` локализованное сообщение с кодом `virtualpets-server-springframework.site.server_info` и записывает его в переменную `server_info`. Затем следующей строкой содержимое переменной `server_info` выводится на страницу внутри тега `<H1>` (подробнее о JSPX и его тегах рассказано в *разд. 11.3*).

Аналогичный код для Thymeleaf показан в листинге 10.5.

Листинг 10.5. header.html

```
<H1 th:text = "#{virtualpets-server-springboot.app_name}"></H1>
```

Приведенный здесь код делает то же самое, что делал код для JSPX-страницы, но не записывает локализованное сообщение в промежуточную переменную.

В Spring Boot, в отличие от Spring Framework, бин `MessageSource` создается автоматически — его не нужно объявлять самостоятельно. Базовые имена файлов с локализованными сообщениями задаются с помощью атрибута `spring.messages.basename`, который по умолчанию смотрит файлы **messages** в `classpath`.

Как уже отмечалось ранее, обязательно необходимо указывать `spring.messages.fallback-to-system-locale` равным `false`. Значение по умолчанию `true` — если файлы, специфичные для предпочитаемой пользователем локали, не найдены, — приводит к ситуации, когда используются файлы с системной локалью. Но для сервера язык системной локали не имеет никакого значения — смысл есть только в локали, предпочитаемой пользователем.

Кодировка файлов с локализованными сообщениями для Spring Boot задается в атрибуте `spring.messages.encoding`. Значение по умолчанию равно UTF-8, значит, property-файлы должны содержать текст в кодировке UTF-8 — т. е. использовать `\u`-значения не нужно.

Файлы **messages.properties** и **messages_ru.properties** имеют содержимое, практически идентичное содержимому файлов с локализованными сообщениями для варианта с virtualpets-server-springframework.

> ***Внимание!***
>
> При локализации приложений важно также форматировать дату, время, денежные суммы и числа в соответствии с локалью клиента.

Сам сервер не форматирует дату, время и денежные суммы в формат, необходимый клиенту, — этим занимается слой представления в JSPX-страницах и в шаблонах Thymeleaf, которые будут рассмотрены позднее в соответствующих разделах книги.

В современных приложениях учета клиентская часть зачастую представляет собой одну HTML-страницу с большим количеством JavaScript-кода и CSS. Все содержимое в этом случае формируется внутри JavaScript. Локализацией соответственно занимается JavaScript с помощью технологий, принятых в нем. Соответствующий пример приведен в клиентской части игры с виртуальными питомцами на JavaScript. Сервер в этом случае возвращает JSON-ответ, содержащий только данные, коды ошибок, дату и время в формате ISO или в миллисекундах, а также денежные суммы в строке либо в числовых значениях.

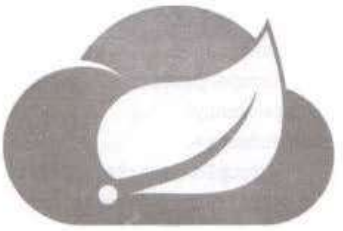
10.3. Резюме

При разработке интернациональных приложений важно изначально закладывать в архитектуру возможность добавления локализаций. Spring предоставляет все инструменты, необходимые для поддержки разных региональных стандартов и языков.

Добавление локализации подразумевает не только добавление перевода на другой язык, но и отображение дат, денежных сумм, чисел согласно принятым в добавляемом регионе нормам и правилам.

JSP/JSPX, Thymeleaf и другие шаблонизаторы позволяют форматировать дату и время в соответствии с выбранной локалью. Если клиентская часть написана на JavaScript, а сервер только предоставляет необходимые данные, то локализацией в первую очередь занимается именно JavaScript-клиент.

ГЛАВА 11



Разработка веб-приложения

11.1. Фреймворк Spring MVC

11.1.1. Настройка для Spring Framework

Spring MVC, или Spring Web MVC — это фреймворк разработки веб-приложений на Spring Framework, основанный на спецификации Jakarta Servlet API[1]. Spring MVC присутствовал в Spring Framework практически с самого начала.

В Spring Framework 5 появилась реактивная версия веб-фреймворка — Spring WebFlux, которая активно развивается и по сей день. Примеры из этой книги используют Spring MVC, но вы можете самостоятельно изучить и даже переписать весь код на Spring WebFlux, если вам будет это интересно.

К проекту virtualpets-server-springframework для работы с Spring MVC подключена зависимость `spring-webmvc` (листинг 11.1).

Листинг 11.1. Файл pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${org.springframework-version}</version>
</dependency>
```

Проект virtualpets-server-springframework использует сервер приложений Apache Tomcat, который предоставляет зависимости со спецификациями Jakarta Servlet API, необходимые для работы Spring MVC (листинг 11.2).

Листинг 11.2. Файл pom.xml

```
<!-- Servlet -->
<dependency>
    <groupId>jakarta.servlet</groupId>
```

[1] См. **https://jakarta.ee/specifications/servlet/**.

```
    <artifactId>jakarta.servlet-api</artifactId>
    <version>6.0.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>jakarta.annotation</groupId>
    <artifactId>jakarta.annotation-api</artifactId>
    <version>2.1.1</version>
    <scope>provided</scope>
</dependency>
```

Версии зависимостей, которые соответствуют каждой из конкретных версий Apache Tomcat, описаны в документации к этим версиям.

Основная обработка запросов происходит в `DispatcherServlet`, представленном в файле web.xml, который внутри себя вызывает методы контроллеров, занимающихся обработкой запроса.

Одного подключения зависимостей недостаточно. Проект virtualpets-server-springframework содержит настройки Spring MVC в файле конфигурации src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml, основное пространство имен которого: `http://www.springframework.org/schema/mvc` — назначено на префикс `mvc`, используемый по умолчанию в этом файле (листинг 11.3).

Листинг 11.3. Файл servlet-context.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans:beans xmlns = "http://www.springframework.org/schema/mvc"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xmlns:beans = "http://www.springframework.org/schema/beans"
    xmlns:context = "http://www.springframework.org/schema/context"
    xmlns:mvc = "http://www.springframework.org/schema/mvc"
    xmlns:p = "http://www.springframework.org/schema/p"
    xsi:schemaLocation = "http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc.xsd
        ">
```

Классы, обрабатывающие HTTP-запросы, в Spring MVC помечаются аннотациями `@Controller`, но для того чтобы эта аннотация заработала и фреймворк Spring MVC находил эти классы, необходимо сначала включить настройку Spring MVC на основе аннотаций (листинг 11.4).

Листинг 11.4. Файл servlet-context.xml

```
<!-- Поддержка @Controller-->
<annotation-driven>
    ...
</annotation-driven>
```

Тег `mvc:annotation-driven` включает поддержку аннотации `@Controller`. Однако, чтобы Spring Framework находил классы с этой аннотацией, они должны находиться в пакетах, в которых Spring производит сканирование бинов, поэтому в этом же файле конфигурации настраивается сканирование бинов на пакет с контроллерами (листинг 11.5).

Листинг 11.5. Файл servlet-context.xml

```
<context:component-scan
    base-package = "ru.urvanov.virtualpets.server.controller" />
```

11.1.2. Настройка для Spring Boot

В случае использования Spring Boot достаточно просто подключить *стартер* (листинг 11.6) — вся остальная необходимая конфигурация по умолчанию осуществляется автоконфигурацией.

Листинг 11.6. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

11.1.3. Контроллеры

Контроллеры помечаются аннотацией `@Controller` или `@RestController`. Методы контроллеров, обрабатывающие запросы, помечаются аннотацией `@RequestMapping` с указанием обрабатываемого ими окончания URL-запроса.

Аннотация `@RestController` работает как аннотация `@Controller`, в которой методы обработки запросов `@RequestMapping` по умолчанию функционируют так, будто помечены аннотацией `@ResponseBody` (листинг 11.7).

Листинг 11.7. PublicController.java

```
package ru.urvanov.virtualpets.server.controller.api;

import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
...
```

```
@RestController                                                    // (1)
@RequestMapping(value = "api/v1/PublicService",
        consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_JSON_VALUE)               // (2)
public class PublicController extends ControllerBase {            // (3)
```

Рассмотрим листинг 11.7 подробнее по отмеченным в нем пунктам:

1. Аннотация `@RestController` указывает, что `PublicController` — это контроллер Spring MVC, а его методы возвращают тело ответа сервера.
2. С помощью аннотации `@RequestMapping` у класса указывается базовый путь, обрабатываемый методами этого контроллера. Методы контроллера `PublicController` обрабатывают запросы к путям, начинающимся с `rest/v1/PublicService`.
3. Класс `PublicController` наследуется от базового класса `ControllerBase`, в котором описаны обработчики исключений.

11.1.4. Обработка HTTP GET

Пример контроллера и его метода `getServers`, обрабатывающего запрос HTTP GET, приведен в листинге 11.8.

Листинг 11.8. PublicController.java

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
...
@RequestMapping(method = RequestMethod.GET,
        value = "serverTechnicalInfo")                             // (1)
public ServerTechnicalInfo getServerTechnicalInfo()
        throws ServiceException {
    return publicService.getServerTechnicalInfo();                 // (2)
}
```

Рассмотрим листинг 11.8 подробнее:

1. С помощью аннотации `@RequestMapping` над методом `getServerTechnicalInfo` указывается, что метод обрабатывает GET-запросы к `api/v1/PublicService/serverTechnicalInfo`. HTTP-метод GET указан в атрибуте `method`, начальная часть обрабатываемых запросов `api/v1/PublicService` берется из `@RequestMapping` у самого класса `PublicController`, конечная часть `servers` указывается в атрибуте `value` аннотации `@RequestMapping` у самого метода
2. Метод `getServerTechnicalInfo` возвращает экземпляр `ServerTechnicalInfo`. Благодаря аннотации `@RestController` этот массив используется в качестве тела ответа сервера. При доступной в `classpath` библиотеке Jackson сериализация происходит в JSON. Библиотека Jackson подключена в файле pom.xml как зависимость в virtualpets-server-springframework и неявно подключается стартером `spring-boot-starter-web` в virtualpets-server-springboot.

Класс `ServerTechnicalInfo` представляет собой обычную запись (листинг 11.9). Записи добавлены в Java 16. Библиотека Jackson, используемая Spring Framework, начиная с версии Jackson 2.12, поддерживает сериализацию и десериализацию записей. Примеры запроса на метод `getServerTechnicalInfo` и ответа этого метода приведены в листингах 11.10 и 11.11 соответственно.

Листинг 11.9. ServerTechnicalInfo.java

```
package ru.urvanov.virtualpets.server.controller.api.domain;

import java.util.Map;

public record ServerTechnicalInfo(Map<String, String> info) {
};
```

Листинг 11.10. Пример запроса на метод getServerTechnicalInfo

```
HTTP GET-запрос на
http://localhost:8080/virtualpets-server-springframework/api/v1/PublicService/serverTechnicalInfo
```

Листинг 11.11. Пример ответа метода getServerTechnicalInfo

```
{
   "info":{
      "java.specification.version":"19",
      "sun.jnu.encoding":"UTF-8",
...
   }
}
```

Методы HTTP GET в большинстве случаев принимают параметры вида `?param1=value1&param2=value`. Для обработки подобных параметров используется аннотация `@RequestParam` (листинг 11.12).

Листинг 11.12. PetController.java

```
@RequestMapping(method = RequestMethod.GET,
        value = "getPetJournalEntries")
public GetPetJournalEntriesResult getPetJournalEntries(
        @AuthenticationPrincipal UserDetailsImpl userDetailsImpl,
        @RequestParam(name = "count") @Min(1) int count)
                throws ServiceException {
    return petService.getPetJournalEntries(
            new UserPetDetails(
                    userDetailsImpl.getUserId(),
                    selectedPet.getPetId()),
            count);
}
```

- Аннотация `@AuthentiationPrincipal` из Spring Security заполняется `userDetailsImpl` текущим пользователем и рассматривается в *разд. 12.6*.

- В атрибуте `name` аннотации `@RequestParam` указывается имя параметра запроса. Этот атрибут помечен как алиас для атрибута `value`, поэтому допустимо писать `@RequestParam("count")` — если требуется указать только имя параметра запроса.
- Аннотация `@Min` используется для проверки передаваемых значений и подробно рассматривается в *разд. 11.2.5*.

11.1.5. Обработка HTTP POST

Метод `register` контроллера `PublicController` обрабатывает запрос HTTP POST (листинг 11.13).

Листинг 11.13. PublicController.java

```
@ResponseStatus(HttpStatus.NO_CONTENT)                              // (1)
@RequestMapping(method = RequestMethod.POST, value = "register") // (2)
public void register(
        @RequestBody @Valid RegisterArgument registerArgument)    // (3)
                throws ServiceException {
    publicService.register(registerArgument);
}
```

Рассмотрим листинг 11.13 подробнее:

1. Аннотация `@ResponseStatus` позволяет указать статус HTTP, отличный от HTTP 200 OK. Метод `register` при своем успешном завершении возвращает результат с HTTP-статусом 204 No Content и без тела ответа.
2. Аналогично методу `getServers` для метода `register` в аннотации `@RequestMapping` в атрибуте `value` указывается конечная часть URL, а в атрибуте `method` отмечается, что обрабатываются запросы HTTP POST.
3. Аннотация `@RequestBody` утверждает, что тело запроса содержит JSON, который необходимо десериализовать в класс `RegisterArgument` и заполнить им параметр `registerArgument`. Аннотация `@Valid` указывает, что необходимо проверить соответствие полей класса `RegisterArgument` и аннотаций Jakarta Validation (листинг 11.14). Аннотации Jakarta Validation описываются в *разд. 11.2*. Пример запроса на метод `register` приведен в листинге 11.15.

Листинг 11.14. RegisterArgument.java

```
package ru.urvanov.virtualpets.server.controller.api.domain;

import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;

public record RegisterArgument(
        @NotNull @Size(min = 3, max = 50) String login,
        @NotNull @Size(min = 3, max = 50) String name,
        @NotNull @Size(min = 3, max = 50) String password,
        @NotNull @Size(min = 3, max = 50) String email,
        @NotNull @Size(min = 1, max = 50) String version) {
};
```

Листинг 11.15. Пример запроса на метод register

```
HTTP POST-запрос на
http://localhost:8080/virtualpets-server-springframework/api/v1/PublicService/register
(детальные заголовки опущены для наглядности – важно заполнить как минимум Content-Type:
application/json)
{
  "login":"user3000",
  "name":"User Userovich 3000",
  "password":"user3000",
  "email":"user3000@nowhere.com",
  "version":"0.21"
}
```

При успешном выполнении в качестве ответа возвращается HTTP 204 NoContent с пустым телом.

11.1.6. Архитектурный стиль REST

Атрибут `method` аннотации `@RequestMethod` может принимать значения не только POST и GET. Для него допустимы также следующие значения:

- GET;
- HEAD;
- POST;
- PUT;
- PATCH;
- DELETE;
- OPTIONS;
- TRACE.

Наиболее популярным архитектурным стилем взаимодействия сервисов в распределенных приложениях в современном мире является REST, при котором чаще всего в сервисах используются следующие типы HTTP-запросов.

- GET — для методов получения данных;
- POST — для методов добавления ресурса;
- PUT — для методов обновления всех полей ресурса;
- PATCH — для частичного изменения ресурса. При PATCH изменяются только поля, переданные в запросе, остальные остаются без изменений;
- DELETE — для методов удаления ресурса.

Сервер игры виртуальных питомцев из этой книги не совсем соответствует REST, так что не стоит его рассматривать именно как образец построения приложения в этом архитектурном стиле. В общем-то, делать тестовое приложение именно в соответствии с требованиями к REST не имеет особого смысла, поскольку сервер всё равно состоит из одного сервиса, да и в целом требования к нему не слишком высокие, поэтому можно рассматривать его всего лишь как пример.

> ***REST — это не стандарт***
>
> Нет официального стандарта REST, как в случае с SOAP. REST — это, скорее, архитектурный стиль, предполагающий следование определенным правилам построения

веб-служб. В различных источниках архитектурный стиль REST трактуется по-разному, и в реальных проектах соответственно никогда нет точного следования этому стилю.

Большая часть методов контроллеров сервера игры виртуальных питомцев использует только HTTP-методы GET и POST.

11.1.7. Обработка HTTP DELETE

Единственный хороший пример использования другого HTTP-метода, а конкретно метода `DELETE`, находится в `PetController` (листинг 11.16).

Листинг 11.16. PetController.java

```
@ResponseStatus(HttpStatus.NO_CONTENT)
@RequestMapping(value = "delete/{petId}",
        method = RequestMethod.DELETE)
public void delete(
        @AuthenticationPrincipal UserDetailsImpl userDetailsImpl,
        @PathVariable("petId") @Min(1) Integer petId)
                throws ServiceException {
    petService.delete(
            new UserPetDetails(
                    userDetailsImpl.getUserId(),
                    selectedPet.getPetId()),
            petId);
}
```

Выглядит этот пример аналогично примерам с GET и POST. Основные его отличия от них заключаются в следующем:

- в атрибуте `method` аннотации `@RequestMethod` указано значение `RequestMethod.DELETE`;
- параметр `petId` метода `delete` помечен аннотацией `@PathVariable`. Это означает, что значение переменной заполняется из переменной пути. Имя переменной пути указывается в аннотации `@PathVariable`. В нашем случае значение переменной берется из переменной пути `petId`.

> ***@PathVariable и @RequestParam***
>
> Имя переменной пути в аннотации `@PathVariable` указывать не обязательно, если ваш код компилируется с параметром `-parameters` из Java 8. До версии Spring 6.1 определение имени параметра происходило с помощью `LocalVariableTableParameterNameDiscoverer`, который анализировал байт-код для определения имени параметра метода. Самым надежным способом на текущий момент будет явное указание имени параметра метода в самом `@PathVariable` и в `@RequestParam`, как это показано в следующих примерах: `@PathVariable("petId")`, `@PathVariable(name = "petId")`, `@RequestParam("petId")`, `@RequestParam(name = "petId")`.

Примером запроса, обрабатываемого методом `delete`, может служить запрос HTTP DELETE на URL **http://localhost:8080/virtualpets-server-springframework/api/v1/PetService/delete/2**, где число **2** — это то самое значение, которое при обработке попадет в параметр `petId` метода `delete` контроллера `PetControllerImpl`.

При его успешном выполнении возвращается значение HTTP 204 NoContent с пустым телом ответа.

11.1.8. Сокращенные аннотации

Для `@RequestMethod` и различных значений атрибута `method` существуют сокращенные аннотации: `@GetMapping`, `@PostMapping` и т. д. Хороший пример использования аннотации `@PostMapping` приведен листинге 11.17.

Листинг 11.17. Файл HiddenObjectsController.java

```
@PostMapping("joinGame")
public HiddenObjectsGame joinGame(
        @AuthenticationPrincipal UserDetailsImpl userDetailsImpl,
        @RequestBody @Valid
        JoinHiddenObjectsGameArg joinHiddenObjectsGameArg)
        throws ServiceException {
    return hiddenObjectsService.joinGame(
            new UserPetDetails(
                    userDetailsImpl.getUserId(),
                    selectedPet.getPetId()),
            hiddenObjectsGameStatus, joinHiddenObjectsGameArg);
}
```

Здесь сокращенная аннотация `@PostMapping` работает аналогично аннотации `@RequestMapping(method = RequestMethod.POST)`.

Пример аннотации `@GetMapping` приведен листинге 11.18.

Листинг 11.18. HiddenObjectsController.java

```
@GetMapping("getGameInfo")
public HiddenObjectsGame getGameInfo(
        @AuthenticationPrincipal UserDetailsImpl userDetailsImpl)
        throws ServiceException {
    return hiddenObjectsService.getGameInfo(
            new UserPetDetails(
                    userDetailsImpl.getUserId(),
                    selectedPet.getPetId()),
            hiddenObjectsGameStatus);
}
```

Сокращенная аннотация `@GetMapping` работает здесь аналогично аннотации `@RequestMapping(method = RequestMethod.GET)`.

11.1.9. Обработка исключений

Методы с аннотацией `@ExceptionHandler` позволяют обрабатывать исключительные ситуации, возникающие в контроллерах при обработке запроса. Методы обработки исключений могут находиться как в самих контроллерах, так и в их родительских классах.

В качестве параметров подобные методы принимают практически всё необходимое для обработки исключения: само исключение, объект сессии, локаль и т. п. Полный список лучше посмотреть в JavaDoc аннотации `@ExceptionHandler`.

Возвращают методы `@ExceptionHandler` полноценный ответ сервера на подобную исключительную ситуацию: в случае применения Jakarta Pages или Thymeleaf это может быть имя представления, при использовании JSON в качестве ответа может выступать объект с `@ResponseBody` и т. д.

Пример обработки исключения в проекте virtualpets-server-springframework приведен в листинге 11.19.

Листинг 11.19. ControllerBase.java

```
package ru.urvanov.virtualpets.server.controller.api;

import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

import ru.urvanov.virtualpets.server.service.exception.IncompatibleVersionException;
import ru.urvanov.virtualpets.server.service.exception.PetNotFoundException;
import ru.urvanov.virtualpets.server.service.exception.ServiceException;
import ru.urvanov.virtualpets.server.service.exception.UserNotFoundException;

public class ControllerBase extends ResponseEntityExceptionHandler {

    @ExceptionHandler(ServiceException.class)
    public final ResponseEntity<Object> handleException(
            ServiceException ex, WebRequest request) throws Exception {
        HttpHeaders headers = new HttpHeaders();
        HttpStatus httpStatus;
        if (ex instanceof UserNotFoundException
                || ex instanceof PetNotFoundException) {
            httpStatus = HttpStatus.NOT_FOUND;
        } else {
            httpStatus = HttpStatus.BAD_REQUEST;
        }
        ProblemDetail problemDetail = ProblemDetail.forStatus(httpStatus);
        problemDetail.setDetail(ex.getErrorCode());
        if (ex instanceof IncompatibleVersionException ieEx) {
          problemDetail.setProperty("serverVersion", ieEx.getServerVersion());
          problemDetail.setProperty("clientVersion", ieEx.getClientVersion());
        }
        return this.createResponseEntity(problemDetail, headers,
                httpStatus, request);
    }
}
```

Сам класс `ControllerBase` расширяет класс `ResponseEntityExceptionHandler`, в котором находятся основные обработчики исключений из Spring MVC, преобразующие эти исключения в `ProblemDetail`.

Листинг 11.19 содержит `ControllerBase` для контроллеров из пакета ru.urvanov.virtualpets.server.controller.api, обрабатывающих запросы к API.

Проект virtualpets-server-springframework содержит также аналогичный контроллер с тем же именем `ControllerBase`, хранящий обработчики исключений для публичного сайта, но в другом пакете — ru.urvanov.virtualpets.server.controller.site (листинг 11.20).

Листинг 11.20. ControllerBase.java

```
package ru.urvanov.virtualpets.server.controller.site;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.servlet.NoHandlerFoundException;

import ru.urvanov.virtualpets.server.service.exception.PetNotFoundException;
import ru.urvanov.virtualpets.server.service.exception.UserNotFoundException;

public class ControllerBase {

...
    @ExceptionHandler(UserNotFoundException.class)
    @ResponseStatus(value = HttpStatus.NOT_FOUND,
            reason = "User was not found.")
    public void userNotFound() throws NoHandlerFoundException {
    }

    @ExceptionHandler(PetNotFoundException.class)
    @ResponseStatus(value = HttpStatus.NOT_FOUND,
            reason = "Pet was not found.")
    public void petNotFound() throws NoHandlerFoundException {
    }
}
```

Проект virtualpets-server-springboot вместо базового класса `ControllerBase` для обработчиков исключений сайта использует класс с аннотацией `@ControllerAdvice`. Аннотация `@ControllerAdvice` позволяет создать отдельный класс для глобальных обработчиков `@ExceptionHandler`, `@InitBinder` и `@ModelAttribute`, используемых всеми контроллерами (листинг 11.21).

Листинг 11.21. GlobalMethods.java

```
package ru.urvanov.virtualpets.server.controller.site;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.servlet.NoHandlerFoundException;

import ru.urvanov.virtualpets.server.service.exception.PetNotFoundException;
import ru.urvanov.virtualpets.server.service.exception.UserNotFoundException;
/**
 * Содержит общие методы
 * @ExceptionHandler, @InitBinder, @ModelAttribute
 * всех контроллеров.
 */
@ControllerAdvice
public class GlobalMethods {



    @ExceptionHandler(UserNotFoundException.class)
    @ResponseStatus(value = HttpStatus.NOT_FOUND,
            reason = "User was not found.")
    public void userNotFound() throws NoHandlerFoundException {
    }

    @ExceptionHandler(PetNotFoundException.class)
    @ResponseStatus(value = HttpStatus.NOT_FOUND,
            reason = "Pet was not found.")
    public void petNotFound() throws NoHandlerFoundException {
    }
}
```

11.2. Спецификация Jakarta Validation

11.2.1. Подключение зависимостей

Jakarta Validation, Jakarta Bean Validation или JSR 380 — это часть Jakarta EE, спецификация Java API для проверки полей бинов с помощью аннотаций вида `@NotNull`, `@Min`, `@Max` и `@Size`.

Для работы с этой спецификацией к проекту необходимо подключить зависимость `hibernate-validator`, которая подтянет свою зависимость от артефакта `jakarta.validation-api` (листинг 11.22).

Примечание

Hibernate Validator — совершенно отдельный проект, независимый от Hibernate.

Листинг 11.22. Файл pom.xml

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>${hibernate-validator.version}</version>
</dependency>
```

Здесь `hibernate-validator.version` — это версия `hibernate`, объявленная в секции `properties` того же файла pom.xml (листинг 11.23).

Листинг 11.23. Файл pom.xml

```
<properties>
...
<hibernate-validator.version>8.0.1.Final</hibernate-validator.version>
...
</properties>
```

Для проекта на Spring Boot вместо зависимости от `hibernate-validator` необходимо подключить зависимость от `spring-boot-starter-validation` (листинг 11.24).

Листинг 11.24. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

11.2.2. Аннотация *@Valid*

Не все входящие запросы проверяются согласно аннотациям Jakarta Validation. Для того чтобы проверка сработала, необходимо использовать аннотацию `@Valid`. В проектах virtualpets-server-springboot и virtualpets-server-springframework практически все контроллеры, принимающие запросы от клиентской части игры, используют аннотацию `@Valid` совместно с аннотацией `@RequestBody` — как, например, показано в листинге 11.25.

Листинг 11.25. PetController.java

```
@ResponseStatus(HttpStatus.NO_CONTENT)
@RequestMapping(value = "drink", method = RequestMethod.POST)
public void drink(
        @AuthenticationPrincipal UserDetailsImpl userDetailsImpl,
        @RequestBody @Valid DrinkArg drinkArg)
                throws ServiceException {
    petService.drink(
            new UserPetDetails(
                    userDetailsImpl.getUserId(),
                    selectedPet.getPetId()),
            drinkArg);
}
```

```
@ResponseStatus(HttpStatus.NO_CONTENT)
@RequestMapping(value = "satiety", method = RequestMethod.POST)
public void eat(
        @AuthenticationPrincipal UserDetailsImpl userDetailsImpl,
        @RequestBody @Valid SatietyArg satietyArg)
                throws ServiceException {
    petService.satiety(
            new UserPetDetails(
                    userDetailsImpl.getUserId(),
                    selectedPet.getPetId()),
            satietyArg);
}
```

11.2.3. Аннотация @NotNull

Сами классы, в которые десериализуются JSON-тела запросов, используют аннотации из пакета `jakarta.validation.constraints`. Каждая из этих аннотаций накладывает определенное ограничение на значения, принимаемые полем.

Например, аннотация `@NotNull` указывает, что поле не должно иметь `null`-значения. Примеры ее использования приведены в листингах 11.26 и 11.27.

Листинг 11.26. DrinkArg.java

```
package ru.urvanov.virtualpets.server.controller.api.domain;

import jakarta.validation.constraints.NotNull;
import ru.urvanov.virtualpets.server.dao.domain.DrinkId;

public record DrinkArg(@NotNull DrinkId drinkId) {};
```

Листинг 11.27. SatietyArg.java

```
package ru.urvanov.virtualpets.server.controller.api.domain;

import jakarta.validation.constraints.NotNull;
import ru.urvanov.virtualpets.server.dao.domain.FoodId;

public record SatietyArg(@NotNull FoodId foodId) {
};
```

Если поле `drinkId` записи `DrinkArg` (см. листинг 11.26) или поле `foodId` записи `SatietyArg` (см. листинг 11.27) придут со значением `null`, то методы `PetController#drink` или `PetController#eat` соответственно бросят исключение `MethodArgumentNotValidException`. Это исключение перехватится классом `ResponseEntityExceptionHandler`, от которого наследуется класс `ControllerBase`. Метод обработки исключения сформирует ответ `ProblemDetail`.

Если метод контроллера имеет параметр с типом `org.springframework.validation.BindingResult`, то исключение не бросается, а сам параметр заполняется значениями,

позволяющими проанализировать поля, содержащие ошибки. В проекте сервера виртуальных питомцев нет примера использования `BindingResult`, но JavaDoc даст полное представление о нем.

11.2.4. Аннотация @Size

Аннотация `@Size` позволяет задать минимальный и максимальный размер строк и коллекций: минимальный размер задается атрибутом `min`, максимальный — атрибутом `max`. Значения указываются как для минимального значения, так и для максимального. Размеры, указанные как в атрибуте `min`, так и в атрибуте `max`, считаются допустимыми размерами поля. Пример для строк приведен в листинге 11.28.

Листинг 11.28. Класс RegisterArgument.java

```
package ru.urvanov.virtualpets.server.controller.api.domain;

import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;

public record RegisterArgument(
        @NotNull @Size(min = 3, max = 50) String login,
        @NotNull @Size(min = 3, max = 50) String name,
        @NotNull @Size(min = 3, max = 50) String password,
        @NotNull @Size(min = 3, max = 50) String email,
        @NotNull @Size(min = 1, max = 50) String version) {
};
```

Здесь задаются следующие ограничения на строковые поля класса `RegisterArgument`:

- поля `login`, `password`, `email` не должны содержать значения `null`, и их длина должна находиться в диапазоне от трех до пятидесяти символов;
- поле `version` не должно быть `null`, и его длина должна должна находиться в диапазоне от одного до пятидесяти символов.

11.2.5. Аннотации @Min и @Max

Аннотации `@Min` и `@Max` позволяют задать минимальное и максимальное значения для поля. Например, поле `objectId` в листинге 11.29 должно содержать значения больше или равные нулю.

Листинг 11.29. CollectObjectArg.java

```
package ru.urvanov.virtualpets.server.controller.api.domain;

import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Min;

public record CollectObjectArg(
        @NotNull @Min(0) Integer objectId
        ) {};
```

Отображение ошибок

Spring Framework добавляет специальные теги, позволяющие выводить ошибки, связанные с проверкой полей на допустимые значения, на страницах Jakarta Pages под полями. Отображение подобных ошибок описывается в *разд. 11.3.22*.

Thymeleaf также позволяет использовать результат проверки JakartaValidation и выводить сообщения под полями с ошибками. Отображение ошибок проверки полей с Thymeleaf описывается в *разд. 11.4.14*.

11.3. Технология Jakarta Pages

11.3.1. Введение

Jakarta Pages, Jakarta Server Pages (ранее — JavaServer Pages), аббревиатура JSP — это технология, предоставляющая простой и быстрый способ создания динамически генерируемых веб-страниц. По принятому подходу к разработке технология похожа на PHP и ASP, но использует язык программирования Java.

JSP-страницы могут использоваться самостоятельно либо в качестве слоя представления.

11.3.2. Примеры в Apache Tomcat

Самая простая JSP-страница разрабатывается аналогично PHP-файлам. Если запустить Apache Tomcat не из IDE, а самостоятельно — из каталога CATALINA_HOME/bin (листинги 11.30 и 11.31), то по адресу **http://localhost:8080** будет доступна страница (рис. 11.1) с ссылками на документацию по Apache Tomcat (поз. 1) и примеры приложений для него (поз. 2).

Листинг 11.30. Запуск Apache Tomcat для Linux

```
./catalina.sh start
```

Листинг 11.31. Запуск Apache Tomcat для Windows

```
catalina.bat start
```

Щелкните мышью на ссылке **Examples** (поз. 2 на рис. 11.1), и отобразится страница с ссылкой на примеры JSP-страниц (рис. 11.2).

Погружаться слишком сильно в разработку страниц JSP и JSPX не стоит. В современном мире интерфейсы веб-приложений обычно разрабатываются на фреймворках наподобие React или Vue. Jakarta Pages — это хорошая технология, но встретите вы ее, скорее всего, только в старых проектах.

11.3.3. Настройка для Spring Framework

Необходимые зависимости проекта virtualpets-server-springframework содержатся в файле pom.xml (листинг 11.32).

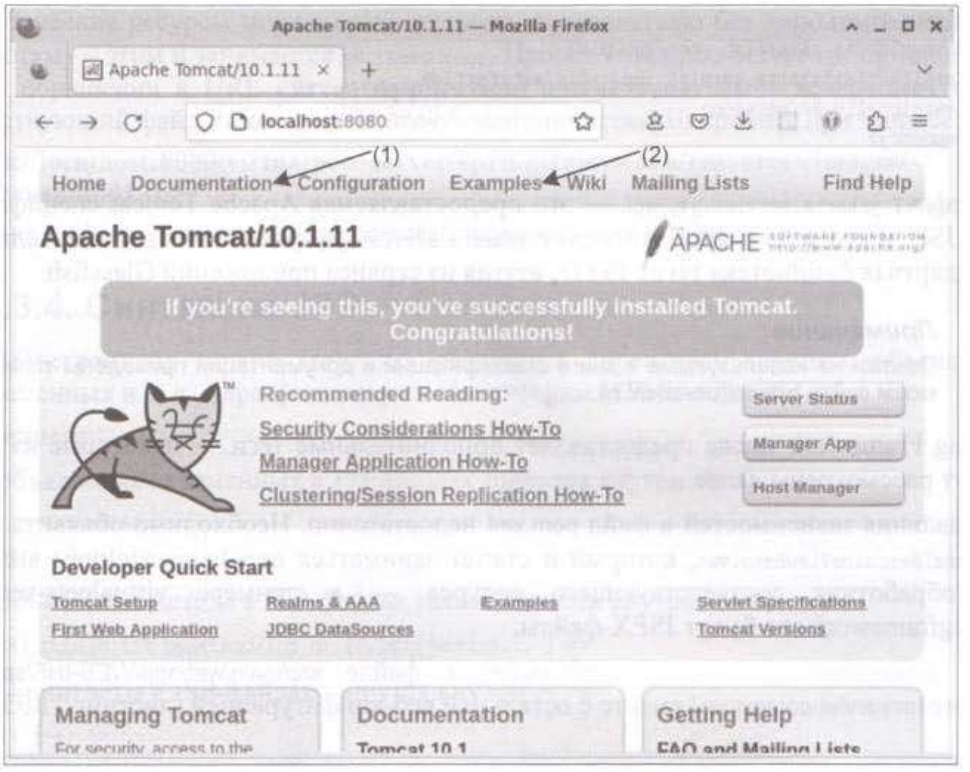


Рис. 11.1. Ссылки на документацию и примеры на **http://localhost:8080** с запущенным сервисом приложений Apache Tomcat

Рис. 11.2. Ссылка на примеры JSP-страниц

Листинг 11.32. Файл pom.xml

```
<dependency>
    <groupId>jakarta.servlet.jsp</groupId>
    <artifactId>jakarta.servlet.jsp-api</artifactId>
    <version>3.1.1</version>
    <scope>provided</scope>
</dependency>
```

```
<dependency>
    <groupId>org.glassfish.web</groupId>
    <artifactId>jakarta.servlet.jsp.jstl</artifactId>
    <version>2.0.0</version>
</dependency>
```

Артефакт `jakarta.servlet.jsp-api` — это предоставляемая Apache Tomcat спецификация JSP[1] и его реализация. Артефакт `jakarta.servlet.jsp.jstl` — это дополнительная стандартная библиотека тегов JSTL[2], взятая из сервиса приложений Glassfish.

> ***Примечание***
>
> **Ссылки на используемые в книге спецификации и документации приведены также на моем сайте https://urvanov.ru.**

Spring Framework также предоставляет дополнительные теги — некоторые из них будут рассмотрены далее.

Добавления зависимостей в файл pom.xml недостаточно. Необходимо объявить бин `InternalResourceViewResolver`, который и станет заниматься перенаправлением вызова на обработчик соответствующего ресурса — в примере virtualpets-server-springframework это будут JSPX-файлы.

Бин `InternalResourceViewResolver` объявлен в файле **src/main/webapp/WEB-INF/spring/appServlet/servlet-context.xml** вместе с остальной веб-конфигурацией (листинг 11.33).

Листинг 11.33. Файл servlet-context.xml

```
<!-- Обрабатывает запросы HTTP GET /resources/**
    для статических ресурсов в  ${webappRoot}/resources -->
<resources mapping = "/resources/**" location = "/resources/" />

<beans:bean id = "viewResolver"
        class = "org.springframework.web.servlet.view.InternalResourceViewResolver">
    <beans:property name = "viewClass"
        value = "org.springframework.web.servlet.view.JstlView"/>
    <beans:property name = "prefix"
        value = "/WEB-INF/views/"/>
    <beans:property name = "suffix"
        value = ".jspx"/>
</beans:bean>
```

Свойства бина `prefix` и `suffix` определяют расположение JSPX-файлов в проекте. В нашем случае при определении JSPX-файла будет добавляться префикс **WEB-INF/views** и суффикс **.jspx**. Например, имя представления `information/serverInfo` преобразуется в файл **WEB-INF/views/information/serverInfo.jspx**, который и станет заниматься генерацией результирующей HTML-страницы.

---

[1] Эти спецификации доступны на **https://jakarta.ee/specifications/pages/**.

[2] Ее спецификации доступны на **https://jakarta.ee/specifications/tags/**.

Статические ресурсы можно сразу отдавать пользователю без дополнительной обработки — этим и занимается `mvc:resources`. Проект virtualpets-server-springframework при обращении к URL **currentApplicationContext/resources/**** возвращает соответствующий файл из каталога src/main/webapp/resources. Например, при запросе:

http://localhost:8080/virtualpets-server-springframework/resources/images/screenshot001.jpg

вернется файл src/main/webapp/resources/images/screenshot001.jpg.

11.3.4. Синтаксис JSP

В JSP-страницах можно встретить различные директивы, скрипты, объявления переменных и т. д., чередующиеся со статическим HTML-кодом.

Например:

- объявления переменных в JSP-файлах пишутся внутри `<%! %>`:

```
<%! int n; %>
<%! int n = 0; %>
```

- объявление метода в JSP-файлах также пишется внутри `<%! %>`:

```
<%! public int myMethod(int n) {...код_метода... } %>
```

- скриптлеты в JSP-файлах пишутся внутри `<% %>`:

```
<% i++; %>
```

- вычисляемые выражения с выводом результата в генерируемую страницу пишутся внутри `<%= %>`:

```
<%= (new java.util.Date()).toLocaleString() %>
```

- комментарии записываются так:

```
<%-- Какой-нибудь комментарий --%>
```

В проекте virtualpets-server-springframework для генерации страниц сайта используются JSPX-файлы.

> ***JSPX-файлы***
>
> JSPX-файлы — это те же JSP-файлы, но с дополнительным требованием: они должны быть корректным XML-документом. Подобное условие накладывает определенные ограничения. Кроме того, определенные элементы из JSP-страниц в JSPX-страницах описываются иначе.

11.3.5. Синтаксис JSPX

Как уже отмечалось ранее, проект virtualpets-server-springframework использует JSPX-файлы. Их синтаксис имеет определенные отличия от JSP-файлов. Так, объявления переменных в JSPX-файлах пишутся внутри тега `jsp:declaration`:

```
<jsp:declaration>
int x;
int y = 0;
</jsp:declaration>
```

```
<jsp:declaration>
public int myMethod(int n) { return 1; } >
</jsp:declaration>
```

Если текст объявления содержит символы, которые недопустимы внутри XML, то необходимо либо использовать `>` и `<`, либо поместить все объявление в `CDATA`:

```
<jsp:declaration> public int compareMethod(int x, y) { return x &gt; y ;}
</jsp:declaration>
<jsp:declaration> <![CDATA[ public int compareMethod(int x, y) { return x > y ;} ]]>
</jsp:declaration>
```

Скриптлеты в JSPX-страницах пишутся внутри тегов `jsp:scriptlet`:

```
<jsp:scriptlet> x++; </jsp:scriptlet>
```

Вычисляемые выражения с выводом результата в генерируемую страницу пишутся внутри тегов `jsp:expression`:

```
<jsp:expression> x + 3 </jsp:expression>
<jsp:expression> java.time.LocalDate.now() </jsp:expression>
```

> ***Проект virtualpets-server-springframework не использует скриптлеты и выражения***
>
> Пользовательские теги, используемые в virtualpets-server-springframework, не допускают наличия внутри себя скриптлетов и выражений, поэтому тестовый проект virtualpets-server-springframework не содержит примеров скриптлетов и выражений, но содержит примеры выражений Jakarta Expression Language.

То есть проект virtualpets-server-springframework использует JSPX-файлы для формирования HTML-страниц сайта, но не задействует объявления, скриптлеты и выражения, описанные в этом разделе. Именно поэтому все примеры в нем до сих пор были абстрактные.

11.3.6. Пользовательские теги

Тестовое приложение virtualpets-server-springframework использует пользовательские теги. Дело в том, что внутри тега `jsp:doBody`, в котором и происходит формирование основного содержимого страницы, все скриптлеты, объявления и выражения не работают, поэтому в проекте нет скриптлетов, а значит, примеров скриптлетов из проекта нет и в этой книге.

Пользовательские теги описываются в каталоге WEB-INF/tags или в каталоге META-INF/tags. Проект virtualpets-server-springframework содержит один-единственный пользовательский тег в файле src/main/webapp/WEB-INF/tags/defaultLayout.tagx (листинг 11.34).

Листинг 11.34. Файл defaultLayout.tagx

```
<?xml version = "1.0" encoding = "UTF-8"?>
<html xmlns:jsp = "http://java.sun.com/JSP/Page"
    xmlns:c = "http://java.sun.com/jsp/jstl/core"
    xmlns:spring = "http://www.springframework.org/tags">
```

```
<jsp:output doctype-root-element = "html"
    doctype-system = "about:legacy-compat"
    omit-xml-declaration = "yes" />
<jsp:directive.attribute name="headAdditional" fragment = "true" />
<head>
    <jsp:invoke fragment = "headAdditional" />
    <meta http-equiv = "Content-Type"
        content = "text/html;charset=UTF-8" />
    <spring:theme code = "stylesheet" var = "stylesheet_var" />
    <spring:url value = "${stylesheet_var}" var = "stylesheet_url" />
    <link rel = "stylesheet" href = "${stylesheet_url}" type = "text/css"
        media = "screen" />
</head>
<body>
    <div class = "header">
        <jsp:include page = "/WEB-INF/views/header.jspx"/>
    </div>
    <div class = "menu">
        <jsp:include page = "/WEB-INF/views/menu.jspx"/>
    </div>
    <div class = "body">
        <jsp:doBody />
    </div>
    <div class = "footer">
        <jsp:include page = "/WEB-INF/views/footer.jspx"/>
    </div>
</body>
</html>
```

Файл *.tagx — это XML-файл, поэтому он начинается со строки:

```
<?xml version = "1.0" encoding = "UTF-8"?>
```

Следующая за строкой с версией строка объявляет тег `html` для результирующей HTML-страницы и подключает библиотеки тегов:

- `xmlns:jsp = "http://java.sun.com/JSP/Page"` — **стандартные теги JSP ассоциируются с префиксом** `jsp`;
- `xmlns:c = "http://java.sun.com/jsp/jstl/core"` — **теги библиотеки JSTL ассоциируются с префиксом** `c`;
- `xmlns:spring = "http://www.springframework.org/tags"` — **теги Spring Framework ассоциируются с префиксом** `spring`.

Тег `jsp:output` с помощью атрибутов `doctype-root-element` и `doctype-system` генерирует первую строку с `DOCTYPE` выходной HTML-страницы:

```
<!DOCTYPE html SYSTEM "about:legacy-compat">
```

Атрибут `omit-xml-declaration = "yes"` убирает объявление `xml` из выходного файла. Строка `<?xml version = "1.0" encoding = "UTF-8"?>` будет исключена из генерируемого HTML.

С помощью тега `jsp:directive.attribute` объявляется фрагмент с именем `headAdditional` — он позволяет добавить дополнительную разметку внутрь тега `head`

генерируемой HTML-страницы. Вставляемое содержимое указывается на страницах, которые используют наш пользовательский тег defaultLayout, с помощью тега jsp:attribute — например, в файле src/main/webapp/WEB-INF/views/home.jspx (листинг 11.35).

Листинг 11.35. Файл home.jspx

```
<?xml version = "1.0" encoding = "UTF-8"?>
<custom:defaultLayout
...
<jsp:attribute name = "headAdditional" >
    <spring:message
        code = "virtualpets-server-springframework.app_name"
        var = "title_var"
        htmlEscape = "true" />
    <title>${title_var}</title>
</jsp:attribute>
...
```

В самом пользовательском теге defaultLayout содержимое фрагмента headAdditional после его объявления вставляется с помощью jsp:invoke внутри тега head (листинг 11.36).

Листинг 11.36. Файл defaultLayout.tagx

```
<jsp:invoke fragment = "headAdditional" />
```

Оставшаяся часть содержимого head описывает Content-Type выходной HTML-страницы, а также подключает стили CSS с поддержкой тем (поддержка тем рассмотрена в *разд. 11.3.25*).

Теги jsp:include включают JSPX-файлы, описывающие генерацию шапки, бокового меню со ссылками и подвала выходной страницы, которые повторяются на каждой странице сайта сервера виртуальных питомцев.

Содержимое самой страницы выводится тегом <jsp:doBody />. Генерация содержимого страницы описывается в отдельных JSPX-файлах, использующих пользовательский тег defaultLayout.

В результате пользовательский тег defaultLayout совместно с подключаемой им таблицей стилей создает пользовательское представление с подвалом, меню со ссылками и шапкой, которые повторяются на каждой странице (рис. 11.3).

11.3.7. Главная страница сайта

В листинге 11.37 приведен пример генерации главной страницы сайта — файл src/main/webapp/WEB-INF/views/home.jspx, в котором как раз и подключается тег defaultLayout.

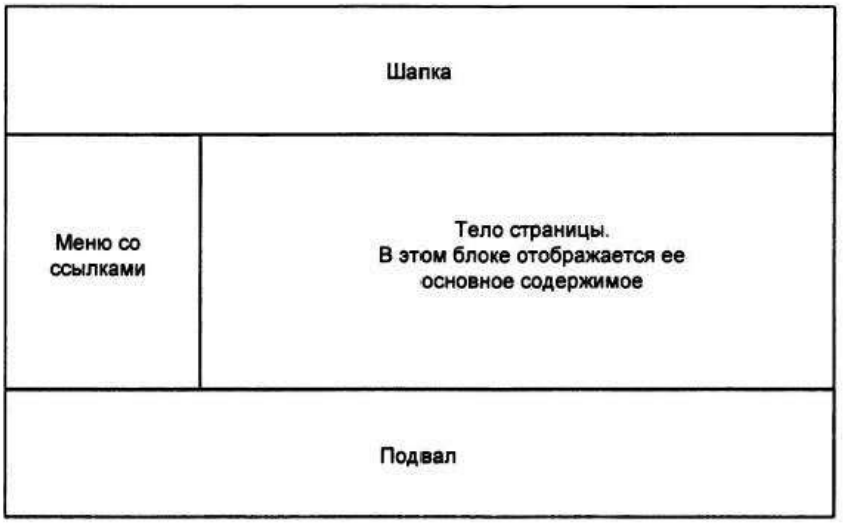


Рис. 11.3. Схема страницы, генерируемой пользовательским тегом defaultLayout **совместно с подключаемыми таблицами стилей**

Листинг 11.37. Файл home.jspx

```
<?xml version = "1.0" encoding = "UTF-8"?>
<custom:defaultLayout xmlns:jsp = "http://java.sun.com/JSP/Page"
    xmlns:c = "http://java.sun.com/jsp/jstl/core"
    xmlns:spring = "http://www.springframework.org/tags"
    xmlns:custom = "urn:jsptagdir:/WEB-INF/tags">
<!--... генерация содержимого ... -->
</custom:defaultLayout>
```

Обратите внимание на объявленные здесь префиксы пространств имен. Префиксы `jsp`, `c` и `spring` уже использовались в файле defaultLayout. Новый префикс и новое пространство имен: `xmlns:custom = "urn:jsptagdir:/WEB-INF/tags"`, назначает префикс `custom` для тегов из каталога WEB-INF/tags, в котором и содержится пользовательский тег `defaultLayout`.

Внутри тега `custom:defaultLayout` в файле home.jspx описывается генерация блока «Тело страницы», показанного на рис. 11.3.

Самым первым действием задается генерация заголовка с элементом `title` (листинг 11.38).

Листинг 11.38. Файл home.jspx

```
<jsp:attribute name = "headAdditional" >
...
</jsp:attribute>
```

Приведенный здесь код с помощью `jsp:attribute` описывает фрагмент `headAdditional`, который используется пользовательским тегом `defaultLayout` для добавления дополнительного содержимого в HTML-элемент `head`.

11.3.8. Локализованные сообщения

Фрагмент `headAdditional` файла home.jspx генерирует HTML-элемент `title` с локализованной строкой (листинг 11.39).

Листинг 11.39. Файл home.jspx

```
<spring:message
    code = "virtualpets-server-springframework.app_name"
    var = "title_var"
    htmlEscape = "true" />
<title>${title_var}</title>
```

Тег `message` с префиксом `spring` используется здесь как аналог `MessageSource` для получения локализованного сообщения с кодом `virtualpets-server-springframework.app_name` и сохранения его в переменной `title_var`. В атрибуте `htmlEscape` указывается, что необходимо экранировать текст сообщения для вставки в HTML-страницу. По умолчанию `htmlEscape` имеет значение `false`, означающее, что экранировать текст не нужно.

После выполнения `spring:message` переменная `title_var` будет хранить локализованное сообщение названия приложения с игрой про виртуальных питомцев, в котором экранированы все необходимые символы для безопасной вставки в текст HTML-страницы.

11.3.9. Выражения Jakarta Expression Language

Выражения Jakarta Expression Language позволяют не только выводить значения переменных в генерируемую страницу, но и осуществлять различные операции сложения, вычитания, умножения, деления, доступ к членам класса и т. д. Для записи выражений Jakarta Expression Language используются знак доллара и фигурные скобки.

При формировании HTML-элемента `title` для главной страницы в генерируемую страницу с помощью выражения Jakarta Expression Language просто выводится содержимое переменной `title_var` (листинг 11.40).

Листинг 11.40. Файл home.jspx

```
<title>${title_var}</title>
```

А в файле src/main/webapp/WEB-INF/views/information/pet.jspx выражение Jakarta Expression Language используется для проверки на `null` с помощью тернарного оператора (листинг 11.41).

Листинг 11.41. Файл pet.jspx

```
<spring:escapeBody>${pet.experience == null
    ? 0 : pet.experience}</spring:escapeBody>
```

Примеры использования выражений Jakarta Expression Language для вычисления атрибута `test` тега `c:if` приведены в файле src/main/webapp/WEB-INF/views/information/statistics.jspx (листинг 11.42).

Листинг 11.42. Файл statistics.jspx

```
<c:if test = "${users.size() > 0}">
...
</c:if>
...
<c:if test = "${pets.size() > 0}">
...
</c:if>
```

11.3.10. Тег *jsp:directive.page*

Следующими строками с помощью тега `jsp:directive.page` в файле home.jspx указывается Content-Type — кодировка самого файла home.jspx. Атрибут `session` определяет, что страница не требует HTTP-сессии (листинг 11.43).

Листинг 11.43. Файл home.jspx

```
<jsp:directive.page
    contentType = "text/html;charset=UTF-8"
    pageEncoding = "UTF-8"
    session = "false" />
```

> ***Обязательно указывайте атрибут session для jsp:directive.page***
>
> Если `session = true`, а это значение по умолчанию, то при первом обращении к JSP/JSPX-странице ко всем статическим ресурсам в конце, через точку с запятой, добавится `jsessionid`, что может повлиять на отображение страницы. Подробнее проблема `jsessionid` описана на моем сайте[1].

Может показаться, что `jsp:directive.page` разумнее было бы указать в файле defaultLayout.tagx, но это не так, — тег `jsp:directive.page` нельзя использовать в файлах тегов, поэтому его приходится повторять в каждом из JSPX-файлов.

11.3.11. Тег *jsp:output*

Тег `jsp:output` уже использовался в `defaultLayout`. В файле home.jspx он выполняет ту же роль — избавляется от определения XML и версии в генерируемой HTML-странице:

```
<jsp:output omit-xml-declaration = "yes" />
```

---

[1] См. **https://urvanov.ru/2024/01/02/jsessionid-в-url-к-статическим-ресурсам/**.

11.3.12. Основное содержимое файла home.jspx

После тега `jsp:output` в файле home.jspx описывается HTML-элемент `div` с `id = main` (листинг 11.44).

Листинг 11.44. Файл home.jspx

```
<div id = "main">
    <spring:message
        code = "virtualpets-server-springframework.site.welcome_message1"
        var = "welcome_message1_var" />
    ${welcome_message1_var}

    <spring:message
        code = "virtualpets-server-springframework.site.welcome_message2"
        var = "welcome_message2_var" />
    ${welcome_message2_var}
    <spring:url value = "/resources/images/screenshot001.jpg"
        var = "screenshot001_url" />
    <img alt = "screenshot001" src = "${screenshot001_url}"
        width = "457" height = "351" />

    <spring:message
        code = "virtualpets-server-springframework.site.welcome_message3"
        var = "welcome_message3_var" />
    ${welcome_message3_var}
    <spring:url value = "/resources/images/screenshot002.jpg"
        var = "screenshot002_url" />
    <img alt = "screenshot002" src = "${screenshot002_url}"
        width = "457" height = "351" />

    <spring:message
        code = "virtualpets-server-springframework.site.welcome_message4"
        var = "welcome_message4_var" />
    ${welcome_message4_var}
    <spring:url value = "/resources/images/screenshot003.jpg"
        var = "screenshot003_url" />
    <img alt = "screenshot003" src = "${screenshot003_url}"
        width = "457" height = "351" />
</div>
```

Большая часть тегов из этого листинга вам должна быть знакома — в частности, тег `spring:message` и выражения Jakarta Expression Language уже описывались ранее. В приведенном случае различие с предыдущим описанием заключается лишь в том, что здесь `spring:message` не указывает атрибут `htmlEscape`, а значит, текст вставляется в генерируемую страницу без изменений, что позволяет в тексте локализованных строк использовать HTML-теги.

11.3.13. Тег *spring:htmlEscape*

Значение атрибута `htmlEscape`, используемое как значение по умолчанию, меняется с помощью тега `spring:htmlEscape`:

```
<spring:htmlEscape defaultHtmlEscape = "true" />
```

В проекте virtualpets-server-springframework тег `spring:htmlEscape` не применяется.

11.3.14. Тег *spring:url*

Тег `spring:url` служит для генерации ссылок на ресурсы относительно текущего развернутого приложения. Так, следующий код:

```
<spring:url value = "/resources/images/screenshot001.jpg"
    var = "screenshot001_url" />
```

запишет в переменную `screenshot001_url` значение вида:

```
/virtualpets-server-springframework/resources/images/screenshot001.jpg
```

Это значение и будет использоваться в качестве интернет-адреса (URL) изображения в следующих строках:

```
<img alt = "screenshot001" src = "${screenshot001_url}"
        width = "457" height = "351" />
```

11.3.15. Контроллер *HomeController*

Остался последний шаг формирования первой страницы сайта. К текущему моменту нами уже описаны:

- шаблон страницы, формируемый пользовательским тегом `defaultLayout.tagx`;
- тело страницы, формируемое файлом `home.jspx`.

Для формирования страницы необходим контроллер, который возвращает имя представления (листинг 11.45).

Листинг 11.45. HomeController.java

```
package ru.urvanov.virtualpets.server.controller.site;

import java.util.Locale;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("site")
public class HomeController extends ControllerBase {

    private static final Logger logger = LoggerFactory.getLogger(HomeController.class);
```

```
    @RequestMapping(value = "/home", method = RequestMethod.GET)
    public String home(Locale locale) {
        logger.info("Welcome home! The client locale is {}.", locale);
        return "home";
    }
}
```

Метод `HomeController#home` обрабатывает запросы вида *контекст_приложения*`/site/home` (например, **http://localhost:8080/virtualpets-server-springframework/site/home**) и возвращает имя представления `home`, которое `InternalResourceViewResolver` превращает в имя файла src/main/webapp/WEB-INF/views/home.jspx — файла home.jspx, содержащего теги, формирующие результирующую HTML-страницу.

11.3.16. Тег *mvc:view-controller*

Существует упрощенный вариант создания контроллеров для случаев, когда необходимо только вернуть имя представления без дополнительных атрибутов модели и какой-либо другой обработки, для чего в конфигурации Spring MVC описывается сопоставление URL и представлений с помощью тега `mvc:view-resolver` (листинг 11.46).

Листинг 11.46. Файл servlet-context.xml

```
<view-controller
    path = "site/download"
    view-name = "download"/>
<view-controller
    path = "site/information"
    view-name = "information/list" />
<view-controller
    path = "site/information/gameHelp"
    view-name = "information/gameHelp" />
<view-controller
    path = "site/login"
    view-name = "login" />
```

Здесь атрибут `path` тега `mvc:view-controller` указывает обрабатываемый URL, а атрибут `view-name` задает имя представления для этого URL. Например, адрес *контекст_приложения*`/site/information` (**http://localhost:8080/virtualpets-server-springframework/site/information**) отображает представление `information/list` (файл src/main/webapp/WEB-INF/views/information/list.jspx).

11.3.17. Атрибуты модели

До сих пор JSPX-страницы генерировали HTML-содержимое без дополнительных данных, получаемых с сервера. Страница информации о сервере, доступная в меню сайта **Информация | О сервере**, получает данные из контроллера `ServerInfoController`.

Сам запрос на отображение страницы **О сервере** выглядит как GET-запрос на URL вида:

http://localhost:8080/virtualpets-server-springframework/site/information/serverInfo

и поступает в метод `serverInfo` контроллера `ServerInfoController` (листинг 11.47).

Листинг 11.47. ServerInfoController.java

```
package ru.urvanov.virtualpets.server.controller.site;

import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.stream.Collectors;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("site")
public class ServerInfoController extends ControllerBase {

    private static final Logger logger = LoggerFactory.getLogger(HomeController.class);

    private static final String HTML_ESCAPE_TEST = "<H1><тест>экранирование</тест></H1>";

    public class Info {
        String key;
        String value;

        public Info(String key, String value) {
            this.key = key;
            this.value = value;
        }

        public String getKey() {
            return key;
        }

        public void setKey(String key) {
            this.key = key;
        }

        public String getValue() {
            return value;
        }
```

```
        public void setValue(String value) {
            this.value = value;
        }

    }

    @RequestMapping(value = "/information/serverInfo",
            method = RequestMethod.GET)
    public String serverInfo(Locale locale, Model model) {
        logger.info("Server info. The client locale is {}.", locale);

        String[] propertyNames = { "java.version", "java.vendor",
                "os.name", "os.arch", "os.version" };
        List<Info> properties = java.util.Arrays.stream(propertyNames)
                .map((key) -> new Info(key, System.getProperty(key)))
                .collect(Collectors.toList());
        List<Info> infos = new ArrayList<>();
        infos.addAll(properties);
        infos.add(new Info(HTML_ESCAPE_TEST, HTML_ESCAPE_TEST));
        model.addAttribute("infos", infos);

        return "information/serverInfo";
    }
}
```

Константа `HTML_ESCAPE_TEST` в контроллере `ServerInfoController` служит для добавления в `infos` элемента с текстом, на котором удобно проверять экранирование HTML-элементов.

Параметр `locale` метода `serverInfo` заполнится предпочитаемой локалью пользователя браузера из HTTP-заголовка `Accept-Language` либо выбранной пользователем локалью из сессии. В методе `serverInfo` параметр `locale` применяется только для вывода полученной информации в лог, но `locale` также можно задействовать, например для получения локализованных строк из `MessageSource` или для форматирования дат и денежных сумм.

Параметр `model` типа `org.springframework.ui.Model` используется для добавления атрибутов к модели. Добавленные атрибуты доступны в JSPX-странице. В нашем случае добавляется атрибут `infos`, содержащий список экземпляров класса `Info` с информацией о версии Java, операционной системе и т. п.

Атрибуты модели добавляются не только с помощью параметра `model`. Для добавления атрибутов модели также используются методы контроллера, помеченные аннотацией `@ModelAttribute`, как это сделано в базовом классе контроллеров страниц сайта проекта virtualpets-server-springframework (листинг 11.48).

Листинг 11.48. ControllerBase.java

```
package ru.urvanov.virtualpets.server.controller.site;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.servlet.NoHandlerFoundException;

import ru.urvanov.virtualpets.server.service.exception.PetNotFoundException;
import ru.urvanov.virtualpets.server.service.exception.UserNotFoundException;

public class ControllerBase {

    @ModelAttribute("menu_play_url")
    public String menuPlayUrl(
            @Value("${virtualpets-server-springframework.play.url}")
            String playUrl) {
        return playUrl;
    }
    ...
}
```

Все другие контроллеры наследуются от класса `ControllerBase`, который добавляет атрибут `menu_play_url` моделям всех обрабатываемых ими запросов. Каждая страница использует этот атрибут в меню сайта со ссылками в качестве адреса JavaScript-клиента игры про виртуальных питомцев.

11.3.18. Тег *c:forEach*

В JSPX-странице с помощью тега `c:forEach` организуется проход по элементам `infos` и выполняется генерация строк таблицы, содержащих значения из этого списка. В атрибуте `items` тега `c:forEach` указывается коллекция элементов, по которой необходимо пройтись. Атрибут `var` служит для указания имени переменной, в которую помещается текущий элемент из коллекции при прохождении по ней (листинг 11.49).

Листинг 11.49. Файл serverInfo.jspx

```
<table>
    <caption>${server_info}</caption>
    <c:forEach var = "info" items = "${infos}">
        <tr>
            <td>
                <c:out value = "${info.key}" />
            </td>
            <td>
                <spring:escapeBody>
```

```
                ${info.value}
            </spring:escapeBody>
        </td>
    </tr>
</c:forEach>
</table>
```

11.3.19. Тег *c:out*

Тег `c:out` из библиотеки тегов JSTL в листинге 11.49 экранирует HTML из выводимого значения. Использование `c:out` часто встречается в JSP/JSPX-файлах.

> ***Уточнение***
> На самом деле `c:out` экранирует XML, а не HTML.

Вариант применения тега `c:out` приведен в листинге 11.50.

Листинг 11.50. Файл serverInfo.jspx

```
<c:out value = "${info.key}" />
```

11.3.20. Тег *spring:escapeBody*

Тег `spring:escapeBody` изначально создан для экранирования HTML или JavaScript. У него есть два атрибута:

- `htmlEscape` — для переопределения значения по умолчанию из тега `spring:htmlEscape`. В файле serverInfo.jspx значение `defaultHtmlEscape` тега `spring:htmlEscape` выставлено в true: `<spring:htmlEscape defaultHtmlEscape = "true" />`, поэтому в листинге 11.49 из *разд. 11.3.18* осуществляется экранирование HTML-элементов;
- `javaScriptEscape` — для экранирования JavaScript. Значение по умолчанию: `false`.

Отображаемая страница информации о сервере, описываемая в *разд. 11.3.18–11.3.20*, показана на рис. 11.4.

11.3.21. Формы

Spring Framework добавляет в JSP/JSPX библиотеку тегов для создания форм. Пример создания формы с помощью этих тегов приведен в файле src/main/webapp/WEB-INF/views/information/statistics.jspx. Форма доступна при переходе из меню сайта **Информация | Статистика**. Для использования тегов создания формы предварительно подключается пространство имен `http://www.springframework.org/tags/form` для префикса `form` (листинг 11.51).

Листинг 11.51. Файл statistics.jspx

```
<?xml version = "1.0" encoding = "UTF-8"?>
<custom:defaultLayout xmlns:jsp = "http://java.sun.com/JSP/Page"
    ...
    xmlns:form = "http://www.springframework.org/tags/form"
    ...>
```

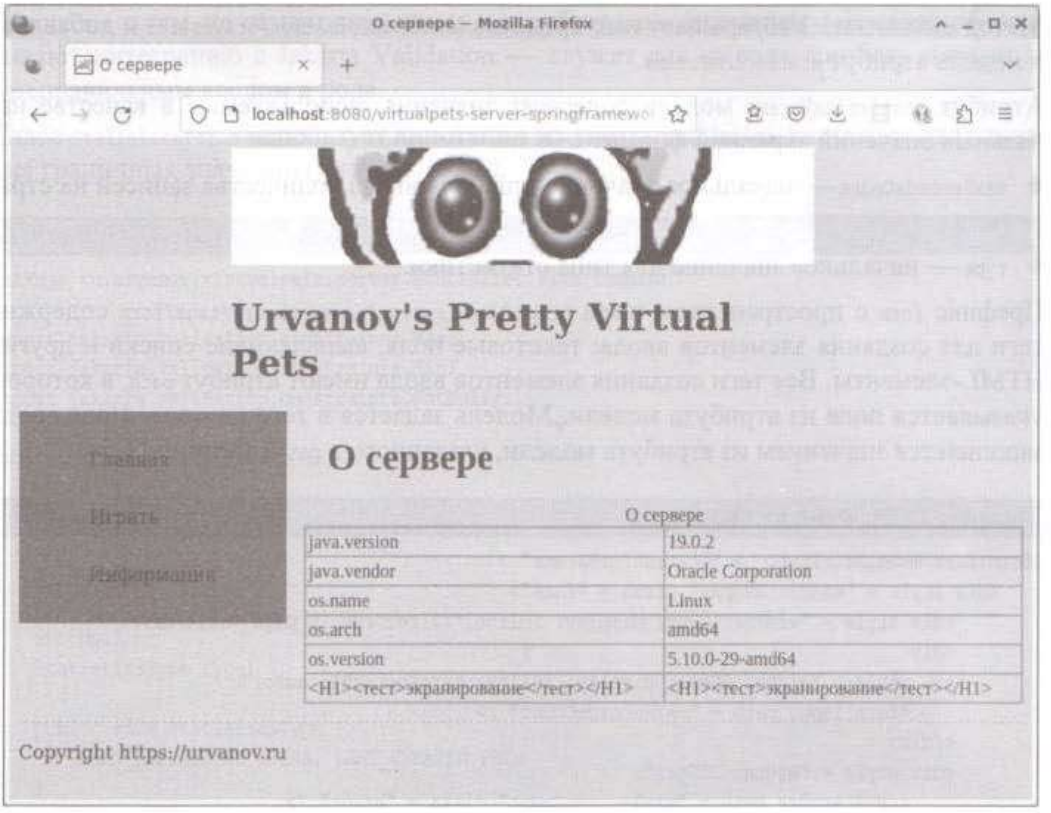


Рис. 11.4. Пример страницы с информацией о сервере

Форма создается с помощью тега `form:form`, в атрибуте `modelAttribute` которого указываются наименование атрибута модели, данные из этого атрибута и заполняются поля формы (листинг 11.52).

Листинг 11.52. Файл statistics.jspx

```
<form:form modelAttribute  = "statisticsParams" >
...
</form:form>
```

Данные для формы возвращают `StatisticsController` (листинг 11.53).

Листинг 11.53. StatisticsController.java

```
@RequestMapping(value = "/information/statistics",
        method = RequestMethod.GET)
public String showStatistics(Locale locale, Model model) {
    StatisticsParams statisticsParams = new StatisticsParams(
            100, StatisticsType.LAST_REGISTERED_USERS);
    model.addAttribute("statisticsParams", statisticsParams);
    return "information/statistics";
}
```

Метод showStatistics возвращает имя представления information/statistics и добавляет в модель атрибут statisticsParams.

Атрибут statisticsParams модели содержит значения, используемые в качестве начальных значений экранной формы:

- maxRecordsCount — начальное значение для поля ввода количества записей на странице;
- type — начальное значение для типа статистики.

Префикс form с пространством имен http://www.springframework.org/tags/form содержит теги для создания элементов ввода: текстовые поля, выпадающие списки и другие HTML-элементы. Все теги создания элементов ввода имеют атрибут path, в котором указывается поле из атрибута модели. Модель задается в теге form:form. Поле ввода заполняется значением из атрибута модели, указанного в path (листинг 11.54).

Листинг 11.54. Файл statistics.jspx

```
<form:form modelAttribute  = "statisticsParams" >
    <div style = "width: 300px;" class = "form">
        <div style = "width: 100px; display: inline;">${records_in_page_var}</div>
        <div
            style = "width: 200px; display: inline; margin-right: auto;">
            <form:input path = "maxRecordsCount" />
        </div>
        <div style = "width: 300px;">
            <form:errors path = "maxRecordsCount" class = "error" />
        </div>
        <div style = "width: 100px; display: inline;">${statistics_type_var}</div>
        <div
            style = "width: 200px; display: inline; margin-right: auto;">
            <form:select path = "type" multiple = "false">
                <option value = "LAST_REGISTERED_USERS">${last_registered_users_var}</option>
                <option value = "LAST_CREATED_PETS">${last_created_pets_var}</option>
            </form:select>
        </div>
        <div style = "width: 300px;">
            <form:errors path = "type" class = "error" />
        </div>
        <div
            style = "width: 300px; display: inline; margin-left: auto;">
            <form:button>${show_statistics_var}</form:button>
        </div>
    </div>
</form:form>
```

11.3.22. Интеграция с Jakarta Validation

Кроме элементов ввода form:input и form:select, для каждого поля ввода из кода формы, прописанного в файле statistics.jspx (см. листинг 11.54), имеется связанное поле

form:errors с тем же самым значением в атрибуте path. Элементы form:errors обеспечивают интеграцию с Jakarta Validation — служат для вывода ошибок, связанных с неправильным вводом в поле.

Класс StatisticsParam использует аннотации из jakarta.validation.constraints для указания граничных значений (листинг 11.55).

Листинг 11.55. StatisticsParam.java

```
package ru.urvanov.virtualpets.server.controller.site.domain;

import jakarta.validation.constraints.Max;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotNull;

public record StatisticsParams (
    @NotNull
    @Min(1)
    @Max(1000)
    Integer maxRecordsCount,

    @NotNull
    StatisticsType type) {

    public enum StatisticsType {
        LAST_REGISTERED_USERS, LAST_CREATED_PETS
    }
}
```

По нажатию на кнопку **Показать статистику** отправляется POST-запрос, обрабатываемый StatisticsController. В методе обработки POST-запроса StatisticsController#showStatistics параметр StatisticsParams указан с аннотацией @Valid, поэтому его поля проходят проверку на правильность заполнения с учетом аннотаций jakarta.validation.constraints (листинг 11.56).

Листинг 11.56. StatisticsController.java

```
@RequestMapping(value = "/information/statistics",
        method = RequestMethod.POST)
public String showStatistics(Locale locale, Model model,
        @Valid @ModelAttribute StatisticsParams statisticsParams,
        BindingResult statisticsParamsBindingResult) {

    List<LastRegisteredUser> users = new ArrayList<>();
    List<LastCreatedPet> pets = new ArrayList<>();
    if (!statisticsParamsBindingResult.hasErrors()) {
        switch (statisticsParams.type()) {
        case LAST_REGISTERED_USERS:
            users = jdbcReportDao.findLastRegisteredUsers(0,
                    statisticsParams.maxRecordsCount());
            break;
```

```
            case LAST_CREATED_PETS:
                pets = petService.findLastCreatedPets(0,
                        statisticsParams.maxRecordsCount());
            }
        }
        model.addAttribute("users", users);
        model.addAttribute("pets", pets);
        return "information/statistics";
    }
```

Если ввести в поле **Записей на странице** значение, выходящее за диапазон значений, указанных в аннотациях `@Min` и `@Max`, то в элементе `form:errors` с атрибутом `path=maxRecordsCount` отобразится соответствующее сообщение (рис. 11.5).

Рис. 11.5. Сообщение о превышении диапазона значений поля Записей на странице

11.3.23. Тег *c:if*

При успешной обработке POST-запроса `StatisticsController` заполняет либо атрибут модели `users`, либо атрибут модели `pets` — в зависимости от типа запрашиваемой статистики.

Код JSPX-страницы должен каким-либо образом определить, заполнен атрибут `users` или атрибут `pets`. Развилка осуществляется с помощью тега `c:if` (листинг 11.57).

Листинг 11.57. Файл statistics.jspx

```
<c:if test = "${users.size() > 0}">
... формирование таблицы со списком пользователей
</c:if>
<c:if test = "${pets.size() > 0}">
... формирование таблицы со списком питомцев
</c:if>
```

Страница с выведенным списком пользователей показана на рис. 11.6.

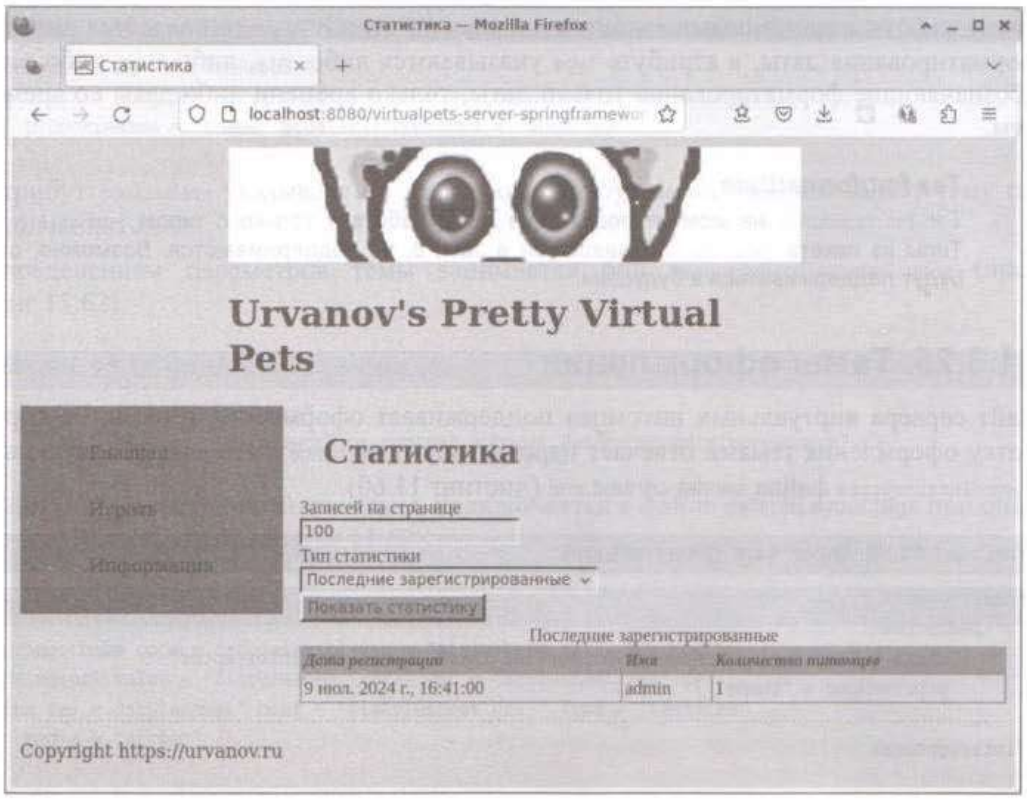


Рис. 11.6. Страница статистики с выведенным списком пользователей

11.3.24. Тег *fmt:formatDate*

Обратите внимание на вывод даты регистрации пользователя на рис. 11.6. Для форматирования даты служит тег `fmt:formatDate` (листинг 11.58).

Листинг 11.58. Файл statistics.jspx

```
<fmt:formatDate value = "${user.registrationDate}" type = "both" />
```

Связывание префикса `fmt` с пространством имен `http://java.sun.com/jsp/jstl/fmt` показано в листинге 11.59.

Листинг 11.59. Файл statistics.jspx

```
<?xml version = "1.0" encoding = "UTF-8"?>
<custom:defaultLayout xmlns:jsp = "http://java.sun.com/JSP/Page"
    ...
    xmlns:fmt = "http://java.sun.com/jsp/jstl/fmt"
    ...>
```

Префикс `fmt`, указанный в начале файла statistics.jspx для пространства имен `http://java.sun.com/jsp/jstl/fmt`, содержит теги для форматирования и парсинга дат,

чисел, работы с временны́ми зонами и локалью. В атрибуте `pattern` задается шаблон форматирования даты, в атрибуте `type` указываются либо `date`, либо `time`, либо `both`, обозначающие форматирование только даты, только времени либо даты со временем.

> ***Тег fmt:formatDate***
>
> Тег `fmt:formatDate` на момент подготовки книги работает только с типом `java.util.Date`. Типы из пакета `java.time`, появившиеся в Java 8, не поддерживаются. Возможно, они будут поддерживаться в будущем.

11.3.25. Темы оформления

Сайт сервера виртуальных питомцев поддерживает оформление темами. За обработку оформления темами отвечает перехватчик `ThemeChangeInterceptor`, объявленный в `mvc:interceptors` файла servlet-context.xml (листинг 11.60).

Листинг 11.60. Файл servlet-context.xml

```
<interceptors>
    <beans:bean
        class = "org.springframework.web.servlet.theme.ThemeChangeInterceptor"
        p:paramName = "theme" />
    ...
</interceptors>
```

В приведенном здесь коде настраивается перехватчик `ThemeChangeInterceptor`, который меняет текущую выбранную тему на тему, указанную в параметре `theme` запроса, — например: `somerequest?theme=pinky`.

Темы отвечают за оформление CSS-стилями HTML-элементов. Всего сайт сервера виртуальных питомцев поддерживает четыре темы:

- `pinky`;
- `school`;
- `standard`;
- `wood`.

Попробуйте самостоятельно переключаться по темам, добавляя параметр **theme=<*название_темы*>** к адресам страницы, — например:

- **http://virtualpets.urvanov.ru/virtualpets-server-springframework/site/home?theme=pinky** — для выбора розовой темы оформления;

  или

- **http://virtualpets.urvanov.ru/virtualpets-server-springframework/site/home?theme=school** — для выбора школьной темы оформления.

Выбранная тема сохраняется в куках браузера, чем занимается бин `CookieThemeResolver` (листинг 11.61).

Листинг 11.61. Файл servlet-context.xml

```
<beans:bean id = "themeResolver"
    class = "org.springframework.web.servlet.theme.CookieThemeResolver"
    p:cookieName = "theme" p:defaultThemeName = "wood" />
```

Атрибут `cookieName` указывает на имя куки, атрибут `defaultThemeName` задает тему по умолчанию `wood`.

Определением параметров темы занимается бин `ResourceBundleThemeSource` (листинг 11.62).

Листинг 11.62. Файл servlet-context.xml

```
<beans:bean id = "themeSource"
    class = "org.springframework.ui.context.support.ResourceBundleThemeSource" />
```

Файл CSS, специфичный для темы, подключается в файле defaultLayout.tagx при описании элемента `HEAD` (листинг 11.63).

Листинг 11.63. Файл defaultLayout.tagx

```
<spring:theme code = "stylesheet" var = "stylesheet_var" />
<spring:url value = "${stylesheet_var}" var = "stylesheet_url" />
<link rel = "stylesheet" href = "${stylesheet_url}" type = "text/css"
    media = "screen" />
```

Здесь тег `spring:theme` сохраняет значение переменной с кодом `stylesheet` из текущей темы в переменную `stylesheet_var`, затем формирует из нее URL с помощью тега `spring:url` и использует его для описания файла CSS-стилей.

Значения переменных, соответствующие разным темам, хранятся в каталоге src/main/webapp/WEB-INF/classes/. В нем находятся четыре файла:

- pinky.properties;
- school.properties;
- standard.properties;
- wood.properties.

Каждый property-файл описывает переменные со значениями, специфичными для темы. В проекте virtualpets-server-springframework используется только одна переменная `stylesheet` — например, для темы `wood` она показана в листинге 11.64.

Листинг 11.64. Файл wood.properties

```
stylesheet=/resources/styles/wood/wood.css
```

Задание

Добавьте самостоятельно в шапку каждой страницы сайта ссылки, позволяющие переключаться между темами оформления. Необходимый для этого код допишите в файл *header.jspx*, чтобы они появились в шапке каждой страницы.

11.3.26. Интернационализация

Сайт сервера виртуальных питомцев поддерживает русский и английский языки. За обработку интернационализации отвечает перехватчик `LocaleChangeInterceptor`, объявленный в `mvc:interceptors` файла servlet-context.xml (листинг 11.65).

Листинг 11.65. Файл servlet-context.xml

```
<interceptors>
...
    <beans:bean
        class = "org.springframework.web.servlet.i18n.LocaleChangeInterceptor"
        p:paramName = "locale" />
</interceptors>
```

Перехватчик `LocaleChangeInterceptor` меняет текущую локаль на локаль, переданную в параметре `locale` запроса, — например: `somerequest?locale=ru` или `somerequest?locale=en`.

Фиксацией выбранной локали занимается бин `CookieLocaleResolver`, сохраняющий ее в куках браузера (листинг 11.66).

Листинг 11.66. Файл servlet-context.xml

```
<beans:bean id = "localeResolver"
    class = "org.springframework.web.servlet.i18n.CookieLocaleResolver"
    p:cookieName = "locale" />
```

Попробуйте самостоятельно попереключаться между языками, добавляя параметр **locale** с выбранным языком к адресу страницы:

- **http://virtualpets.urvanov.ru/virtualpets-server-springframework/site/home?locale=en**;

  или

- **http://virtualpets.urvanov.ru/virtualpets-server-springframework/site/home?locale=ru**.

> ***Задание***
>
> Добавьте самостоятельно в шапку каждой страницы сайта ссылки, позволяющие переключаться между русским и английским языком. Необходимый для этого код допишите в файл *header.jspx*, чтобы они появились в шапке каждой страницы.

11.4. Шаблонизатор Thymeleaf

11.4.1. Thymleaf как современная замена Jakarta Pages

Jakarta Pages допускается использовать и в проектах Spring Boot, но есть определенные ограничения — как минимум необходимо собирать war-файл вместо jar-файла и запускать собранный war-файл с помощью `java -jar`.

В реальности технология JSP/JSPX не используется при создании новых проектов. Обычно, как уже упоминалось ранее, фронтенд пишется на JavaScript-фреймворках типа React или Vue и им подобных. Если же в проекте приходится отображать и обрабатывать пользовательский интерфейс силами разработчиков бэкенда, то лучше воспользоваться другими шаблонизаторами.

Spring Framework из их числа поддерживает:

- Thymeleaf (рекомендуется к использованию);
- FreeMarker;
- Groovy Markup;
- Script Views;
- JSP/ JSTL.

Thymeleaf — это современный серверный шаблонизатор для веб- и десктопной разработки. Целью Thymeleaf является полная замена Jakarta Pages и создание таких файлов шаблонов, содержимое которых отображается браузерами как корректные HTML-страницы.

> ***Thymeleaf генерирует не только HTML***
>
> Thymeleaf позволяет создавать шаблоны не только HTML, но и XML, JavaScript, CSS или обычного текста. Проект virtualpets-server-springboot использует его лишь для генерации HTML, но вы можете генерировать и другие типы файлов в своих проектах.

11.4.2. Настройка для Spring Boot

Поддержка Thymeleaf подключается к проекту на Spring Boot добавлением стартера `spring-boot-starter-thymeleaf`, как это сделано в проекте virtualpets-server-springboot (листинг 11.67).

Листинг 11.67. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Вся необходимая конфигурация в последних версиях Spring Boot происходит автоматически. После добавления зависимости можно сразу приступать к созданию файлов шаблонов. Шаблоны Thymeleaf представляют собой обычные HTML-файлы с дополнительными инструкциями, обрабатываемыми шаблонизатором. При конфигурации по умолчанию эти шаблоны размещаются в каталоге src/main/resources/templates.

11.4.3. Контроллер

Запрос к главной странице сайта обрабатывается контроллером `HomeController`, который выглядит аналогично контроллеру из проекта на Spring Framework (листинг 11.68).

Листинг 11.68. HomeController

```
package ru.urvanov.virtualpets.server.controller.site;

import java.util.Locale;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("site")
public class HomeController {

    private static final Logger logger = LoggerFactory
            .getLogger(HomeController.class);

    @RequestMapping(value = "/home",
            method = RequestMethod.GET)
    public String home(Locale locale) {
        logger.info("Welcome home! The client locale is {}.", locale);
        return "home";
    }
}
```

11.4.4. Префикс *th*

Метод `HomeController#home` просто возвращает имя шаблона. Сам шаблон home.html начинается с `DOCTYPE` и элемента `html` (листинг 11.69).

Листинг 11.69. Файл home.html

```
<!DOCTYPE html>
<html xmlns:th = "http://www.thymeleaf.org"
    th:lang = "${#locale.toLanguageTag()}">
```

Приведенный здесь код выглядит как обычная разметка HTML-страницы — различие лишь в дополнительных атрибутах `xmlns:th` и `th:lang`.

Атрибут `xmlns:th = "http://www.thymeleaf.org"` никак не влияет на итоговую генерируемую Thymeleaf разметку — он нужен лишь для того, чтобы подсветка синтаксиса в IDE не подчеркивала все префиксы `th` в остальной части шаблона.

Атрибут `th:lang` генерирует на выходе атрибут `lang` элемента `html` (`<html lang = "ru-RU">`).

Внутри атрибута `th:lang` используется выражение Spring Expression Language (SpEL), а `locale` — это переменная типа `java.util.Locale`, содержащая выбранную пользователем локаль — английскую или русскую (по умолчанию локаль, переданную браузером, как предпочитаемую).

11.4.5. Контекстно-относительные ссылки

В следующих строках файл home.html описывает HTML-элемент head (листинг 11.70).

Листинг 11.70. Файл home.html

```
<head>
  <meta charset = "utf-8">
  <link rel = "stylesheet" th:href = "@{/wood.css}" type = "text/css"
    media = "screen" />
</head>
```

Приведенный здесь код содержит пример новой конструкции th:href = "@{/wood.css}", где @{/wood.css} — это создание контекстно-относительной ссылки, которая преобразуется в ссылку относительно контекста развернутого приложения. Например, если сервер виртуальных питомцев развернут на **http://localhost:8080/virtualpets-server-springboot**, то это ссылка virtualpets-server-springboot. Проект virtualpets-server-springboot по умолчанию запускается с пустым контекстом, т. е. по адресу **http://localhost:8080**, поэтому генерируемая ссылка выглядит как href = "/wood.css".

Конструкция создания контекстно-относительных ссылок поддерживает добавление параметров, которые указываются в скобках после основной части ссылки. В качестве значений параметров допускается использовать значения атрибутов модели. Например, в шаблоне information/statistics.html передача параметров служит для указания идентификатора питомца в ссылке:

```
th:href = "@{/site/information/pet(id=${pet.id})}"
```

11.4.6. Фрагменты

Элементы div с атрибутами th:replace используются для вставки в генерируемую страницу шапки сайта, меню и подвала (листинг 11.71).

Листинг 11.71. Файл home.html

```
<body>
    <div th:replace = "fragments/header :: header">
        Шапка сайта
    </div>
    <div th:replace = "fragments/menu :: menu">
        Меню сайта
    </div>
    <div class = "body">
        ...
    </div>
    <div th:replace = "fragments/footer :: footer">
    </div>
</body>
</html>
```

Обратите здесь внимание на атрибуты th:replace — они указывают путь к шаблону, из которого необходимо взять фрагмент и имя фрагмента, разделенные двумя символами двоеточия. Например, для шапки сайта берется фрагмент с именем header из шаблона src/main/resources/templates/fragments/header.html.

Внутри тега div описывается содержимое страницы, которое отображается при открытии шаблона в браузере. Сам браузер, разумеется, не может выполнить инструкции Thymeleaf по вставке фрагмента, но он может отобразить HTML-содержимое из самого шаблона. В файле home.html в качестве содержимого фрагментов для браузера просто указаны строки вида Шапка сайта и Меню сайта.

11.4.7. Элемент *th:block*

HTML-элемент файла home.html — это основное содержимое страницы (листинг 11.72).

Листинг 11.72. Файл home.html

```
<div class = "body">
    <div id = "main">
    <th:block th:utext
        = "#{virtualpets-server-springboot.site.welcome_message1}">
        <p>Текст приветственного сообщения 1.</p>
    </th:block>

    <th:block th:utext
        = "#{virtualpets-server-springboot.site.welcome_message2}" >
        <p>Текст приветственного сообщения 2.</p>
    </th:block>
    <img alt = "screenshot001" th:src = "@{/screenshot001.jpg}"
        width = "457" height = "351" />

    <th:block th:utext
        = "#{virtualpets-server-springboot.site.welcome_message3}">
        <p>Текст приветственного сообщения 3.</p>
    </th:block>
    <img alt = "screenshot002" th:src = "@{/screenshot002.jpg}"
        width = "457" height = "351" />

    <th:block th:utext
        = "#{virtualpets-server-springboot.site.welcome_message4}">
        <p>Текст приветственного сообщения 4.</p>
    </th:block>
    <img alt = "screenshot003" th:src = "@{/screenshot003.jpg}"
        width = "457" height = "351" />
    </div>
</div>
```

Элемент th:block из приведенного здесь кода — это синтетический элемент, необходимый только для указания ему атрибутов. Thymeleaf после выполнения атрибу-

тов полностью уберет этот элемент из результирующей страницы и заменит на сгенерированное на основе его атрибутов содержимое. Строки `Текст приветственного сообщения N.` нужны только при открытии шаблона браузером. Вместо них можно разметить любую полноценную разметку HTML.

11.4.8. Локализованные сообщения

Рассмотрим отдельно один блок `th:block` из файла home.html (листинг 11.73).

Листинг 11.73. Файл home.html

```
<th:block th:utext
    = "#{virtualpets-server-springboot.site.welcome_message1}">
    <p>Текст приветственного сообщения 1.</p>
</th:block>
```

Атрибут `th:utext` используется для вставки в содержимое тега локализованных сообщений из MessageSource. Код локализованного сообщения указывается внутри фигурных скобок выражения `#{}`. Существует похожий на него элемент `th:text`, который делает то же самое, но осуществляет экранирование HTML-элементов. В нашем случае используется именно `th:utext`, т. к. локализованные сообщения содержат в том числе и разметку HTML, которую необходимо интегрировать в страницу.

11.4.9. Фрагмент *header*

HTML-элементы `img` в атрибутах `th:src` реализуют способ формирования ссылок, аналогичный тому, который использовался при формировании ссылки на файл со стилями wood.css.

Файл fragments/header.html содержит разметку шапки сайта (листинг 11.74).

Листинг 11.74. Файл header.html

```
<div class = "header"
    xmlns:th = "http://www.thymeleaf.org"
    th:fragment = "header">
    <img alt = "virtualpets logo" th:src = "@{/logo.jpg}"
        width = "500" height = "100" />
    <H1 th:text = "#{virtualpets-server-springboot.app_name}"></H1>
</div>
```

Обратите внимание на атрибут `th:fragment`, в котором указано имя фрагмента `header`. В одном файле может находиться несколько фрагментов, каждый со своим именем. Thymeleaf при вставке фрагмента через `th:replace` осуществляет поиск фрагмента по имени и вставляет вместо элемента с атрибутом `th:replace` содержимое найденного фрагмента.

11.4.10. Фрагмент *menu*

Файл фрагмента fragments/menu.html организован аналогично фрагменту с шапкой сайта (листинг 11.75).

Листинг 11.75. Файл menu.html

```
<div class = "menu"
    xmlns:th = "http://www.thymeleaf.org"
    th:fragment = "menu">
    <ul>
    <li><a th:href = "@{/site/home}"
        th:text = "#{virtualpets-server-springboot.site.home}"></a></li>
    <li><a th:href = "${menu_play_url}"
        th:text = "#{virtualpets-server-springboot.site.play}"></a></li>
    <li><a th:href = "@{/site/information}"
        th:text = "#{virtualpets-server-springboot.site.information}">
        </a></li>
    </ul>
</div>
```

Единственное действительно существенное отличие от рассмотренных ранее листингов заключается в формировании ссылки **Играть**. В качестве ссылки используется атрибут модели `menu_play_url`:

```
<a th:href = "${menu_play_url}"
       th:text = "#{virtualpets-server-springboot.site.play}"></a>
```

Атрибут модели `menu_play_url` мы могли бы добавить в модель с базовым классом контроллеров, как это было сделано в проекте virtualpets-server-springframework, но в проекте virtualpets-server-springboot используется `@ControllerAdvice` специально для того, чтобы показать другой способ.

Аннотация `@ControllerAdvice` — это спецификация `@Component`, в которой все методы с аннотациями `@ExceptionHandler`, `@InitBinder` и `@ModelAttribute` используются всеми другими классами с аннотацией `@Controller` (листинг 11.76).

Листинг 11.76. GlobalMethods.java

```
package ru.urvanov.virtualpets.server.controller.site;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.servlet.NoHandlerFoundException;

import ru.urvanov.virtualpets.server.service.exception.PetNotFoundException;
import ru.urvanov.virtualpets.server.service.exception.UserNotFoundException;
```

```
/**
 * Содержит общие методы
 * @ExceptionHandler, @InitBinder, @ModelAttribute
 * всех контроллеров.
 */
@ControllerAdvice
public class GlobalMethods {

    /**
     * Добавляет атрибут menu_play_url к модели всех контроллеров.
     * Атрибут menu_play_url используется фрагментом
     * src/main/resources/templates/fragments/menu.html
     * в качестве ссылки на клиентскую часть игры.
     * @param playUrl Ссылка на клиентскую часть игры.
     * @return Значение для атрибута menu_play_url
     */
    @ModelAttribute("menu_play_url")
    public String menuPlayUrl(
            @Value("${virtualpets-server-springboot.play.url}")
            String playUrl) {
        return playUrl;
    }
...
```

11.4.11. Фрагмент *footer*

Файл фрагмента fragments/footer.html организован аналогично файлу фрагмента с шапкой сайта (листинг 11.77).

Листинг 11.77. Файл footer.html

```
<div class = "footer"
    xmlns:th = "http://www.thymeleaf.org"
    th:fragment = "footer">
    <p>Copyright https://urvanov.ru</p>
</div>
```

11.4.12. Атрибут *th:each*

Thymeleaf имеет атрибут th:each который, аналогично тегу Jakarta Pages c:forEach позволяет пройтись по коллекции, хранящейся в атрибуте модели, и для каждого элемента коллекции сформировать разметку. Пример подобной генерации показан в файле information/serverInfo.html (листинг 11.78).

Листинг 11.78. Файл serverInfo.html

```
<table>
    <caption th:text = "${server_info}"></caption>
    <tr th:each = "info : ${infos}">
        <td th:text = "${info.key}"></td>
```

```
        <td th:text = "${info.value}"></td>
    </tr>
</table>
```

Элемент `tr` повторяется для каждого элемента из коллекции, хранящейся в атрибуте `infos` модели. На каждой итерации в переменную `info` записывается текущее значение из коллекции, которое используется для формирования строки таблицы.

Контроллер `ServerInfoController`, заполняющий атрибут `infos` модели и возвращающий имя шаблона, выглядит аналогично своей версии для Jakarta Pages из проекта virtualpets-server-springframework (листинг 11.79).

Листинг 11.79. ServerInfoController.java

```
package ru.urvanov.virtualpets.server.controller.site;

import java.util.List;
import java.util.Locale;
import java.util.stream.Collectors;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import jakarta.servlet.http.HttpServletRequest;

@Controller
@RequestMapping("site")
public class ServerInfoController {

    private static final Logger logger = LoggerFactory
            .getLogger(HomeController.class);

    public class Info {
        String key;
        String value;

        public Info(String key, String value) {
            this.key = key;
            this.value = value;
        }

        public String getKey() {
            return key;
        }

        public void setKey(String key) {
            this.key = key;
        }
```

```
        public String getValue() {
            return value;
        }

        public void setValue(String value) {
            this.value = value;
        }

    }

    @RequestMapping(value = "/information/serverInfo",  method = RequestMethod.GET)
    public String serverInfo(Locale locale, Model model,
            HttpServletRequest request) {
        logger.info("Server info. The client locale is {}.", locale);

        String[] propertyNames = { "java.version", "java.vendor",
                "os.name", "os.arch", "os.version" };
        List<Info> infos = java.util.Arrays.stream(propertyNames)
                .map((key) -> new Info(key, System.getProperty(key)))
                .collect(Collectors.toList());
        model.addAttribute("infos", infos);

        return "information/serverInfo";
    }
}
```

Файл шаблона information/pet.html также содержит пример прохода по элементам коллекции (листинг 11.80).

Листинг 11.80. Файл pet.html

```
<div  th:each = "info : ${pet.achievements}">
    <div class = "achievementUnlocked" th:if = "${info.unlocked}">
        <p th:text = "#{virtualpets-server-springboot.achievement.__${info.code}__}">
        </p>
        <p th:text = "#{virtualpets-server-
springboot.achievement.__${info.code}___DESCRIPTION}">
        </p>
    </div>
    <div class = "achievementLocked" th:if = "${!info.unlocked}">
        <p th:text = "#{virtualpets-server-springboot.achievement.__${info.code}__}">
        </p>
        <p th:text = "#{virtualpets-server-springboot.achievement.__${info.code}___DESCRIPTION}">
        </p>
    </div>
    <p>
    </p>
</div>
```

Атрибут `th:each` **тега** `div` **в приведенном здесь коде инструктирует Thymeleaf пройти по элементам коллекции из атрибута модели** `pet.achievements`**, содержащего список**

всех возможных достижений с флагом получения этого достижения питомцем в поле unlocked. На каждой итерации в переменной info сохраняется значение текущего элемента. Элемент div вместе с содержимым генерируется для каждой итерации с новым значением info.

> ***Выражения SpEL внутри кодов локализованных сообщений***
>
> В коде локализованного сообщения допускается использовать выражения Spring Expression Language внутри конструкции __${}__, как это показано в листинге 11.80. Например, выражение:
>
> th:text = "#{virtualpets-server-springboot.achievement.__${info.code}__DESCRIPTION}"
>
> для кода достижения BUILD_1, хранящегося в info.code, приведет к выводу в генерируемую страницу локализованной строки с кодом virtualpets-server-springboot.achievement.BUILD_1_DESCRIPTION.

11.4.13. Формы

Thymeleaf поддерживает создание форм с помощью атрибутов th:object, th:field и th:errors. Пример создания формы с помощью этих атрибутов приведен в файле src/main/resources/templates/information/statistics.html. Форма доступна при переходе из меню сайта **Информация | Статистика**.

Атрибут th:object используется для указания имени атрибута модели, содержащего объект, полями которого заполняется форма (листинг 11.81).

Листинг 11.81. Файл statistics.html

```
<form th:object = "${statisticsParams}" method = "post">
...
</form>
```

С помощью атрибута th:field связывается имя атрибута модели и поле формы. Пример для поля statisticsParams.maxRecordsCount приведен в листинге 11.82.

Листинг 11.82. Файл statistics.html

```
<form th:object = "${statisticsParams}" method = "post">
...
<input th:field = "*{maxRecordsCount}" />
...
</form>
```

Приведенный здесь код связывает поле ввода в генерируемой HTML-форме с полем maxRecordsCount объекта statisticsParams модели.

Полный код шаблона формы приведен в листинге 11.83.

Листинг 11.83. Файл statistis.html

```
<form th:object = "${statisticsParams}" method = "post">
    <div style = "width: 300px;" class = "form">
        <div style = "width: 100px; display: inline;" th:text
                        = "#{virtualpets-server-springboot.site.records_in_page}"></div>
        <div
            style = "width: 200px; display: inline; margin-right: auto;">
            <input th:field = "*{maxRecordsCount}" />
        </div>
        <div style = "width: 300px;">
            <p th:if = "${#fields.hasErrors('maxRecordsCount')}" th:errors
                                                = "*{maxRecordsCount}">Incorrect value</p>
        </div>
        <div style = "width: 100px; display: inline;" th:text
                        = "#{virtualpets-server-springboot.site.statistics_type}"></div>
        <div
            style = "width: 200px; display: inline; margin-right: auto;">
            <select th:field = "*{type}" size = "1">
                <option th:value = "LAST_REGISTERED_USERS" th:text
                    = "#{virtualpets-server-springboot.site.last_registered_users}"></option>
                <option th:value = "LAST_CREATED_PETS" th:text
                    = "#{virtualpets-server-springboot.site.last_created_pets}"></option>
            </select>
        </div>
        <div style = "width: 300px;">
            <p th:if = "${#fields.hasErrors('type')}" th:errors = "*{type}">Incorrect type</p>
        </div>
        <div style = "width: 300px;">
            <input type = "hidden"
                th:name = "${_csrf.parameterName}"
                th:value = "${_csrf.token}"/>
        </div>
        <div
            style = "width: 300px; display: inline; margin-left: auto;">
            <input type = "submit" th:value
                            = "#{virtualpets-server-springboot.site.show_statistics}" />
        </div>
    </div>
</form>
```

Обработка данных формы на стороне контроллера происходит аналогично обработке данных с формы Jakarta Pages (листинг 11.84).

Листинг 11.84. StatisticsController.java

```
@RequestMapping(value = "/information/statistics",
        method = RequestMethod.GET)
public String showStatistics(Locale locale, Model model) {
    StatisticsParams statisticsParams = new StatisticsParams();
    statisticsParams.setMaxRecordsCount(100);
    statisticsParams.setType(StatisticsType.LAST_REGISTERED_USERS);
```

```
    model.addAttribute("statisticsParams", statisticsParams);
    return "information/statistics";
}

@RequestMapping(value = "/information/statistics", method = RequestMethod.POST)
public String showStatistics(Locale locale, Model model,
        @Valid @ModelAttribute StatisticsParams statisticsParams,
        BindingResult statisticsParamsBindingResult) {

    Iterable<LastRegisteredUser> users = new ArrayList<>();
    Iterable<Pet> pets = new ArrayList<Pet>();
    if (!statisticsParamsBindingResult.hasErrors()) {
        switch (statisticsParams.getType()) {
        case LAST_REGISTERED_USERS:
            users = jdbcReportDao.findLastRegisteredUsers(0,
                    statisticsParams.getMaxRecordsCount());
            break;
        case LAST_CREATED_PETS:
            pets = petService.findLastCreatedPets(0,
                    statisticsParams.getMaxRecordsCount());
        }
    }
    model.addAttribute("users", users);
    model.addAttribute("pets", pets);
    return "information/statistics";
}
```

11.4.14. Интеграция с Jakarta Validation

С каждым полем могут быть связаны ошибки проверки допустимых значений Jakarta Validation, описанные в *разд. 11.2*. С помощью метода `hasErrors` объекта `fields` проверяется наличие ошибок, связанных с полем. Вывод текста ошибок в генерируемую форму осуществляется с помощью атрибута `th:errors` (листинг 11.85).

Листинг 11.85. Файл statistics.html

```
<p th:if = "${#fields.hasErrors('maxRecordsCount')}" th:errors
                                                    = "*{maxRecordsCount}">Incorrect value</p>
```

Логика работы с проверкой допустимых значений на серверной части работает аналогично подобной логике из *разд. 11.3.22*, где описывалась интеграция Jakarta Validation и Jakarta Pages.

11.4.15. Атрибут *th:if*

Аналогично тегу `c:if` из Jakarta Pages в шаблонах Thymeleaf поддерживается атрибут `th:if`. Thymeleaf вычисляет значение выражения SpEL атрибута `th:if` и выводит элемент только в случае, если результатом будет значение `true`. Проект virtualpets-server-springboot использует атрибут `th:if` в шаблоне information/statistics.html (листинг 11.86).

Листинг 11.86. Файл statistics.html

```
<table th:if = "${not #lists.isEmpty(users)}">
    ... формирование таблицы со списком пользователей
</table>
<table th:if = "${not #lists.isEmpty(pets)}">
    ... формирование таблицы со списком питомцев
</table>
```

11.5. Модуль Spring WebFlux

Фреймворк Spring Web MVC построен на спецификации Jakarta Servlet, обработка запросов в котором происходит в синхронном режиме. Для каждого запроса при этом создается или достается из пула отдельный поток. Каждый запрос обрабатывает полностью отдельный, специально выделенный для этого запроса поток, как показано на рис. 11.7.

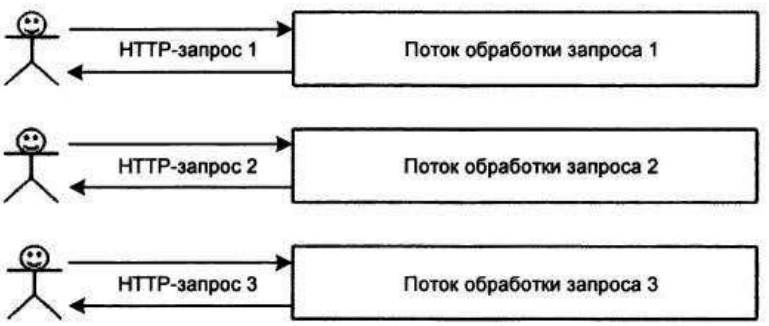


Рис. 11.7. Spring Web MVC обрабатывает каждый запрос отдельным потоком

Модуль Spring WebFlux использует *реактивный подход* к обработке запросов с небольшим количеством потоков и с меньшим расходом аппаратных ресурсов (рис. 11.8).

Рис. 11.8. Обработка запросов реактивными потоками Spring WebFlux

Реактивный подход

Термином «реактивный» называют программную модель, построенную на реакции на события:

- сетевые компоненты реагируют на получение и завершение отправки данных;
- контроллеры пользовательского интерфейса реагируют на нажатия клавиш на клавиатуре, на события компьютерной мыши и т. п.;
- компоненты работы с базой данных реагируют на получение порции данных из базы;

и т. д.

Другим важным механизмом, который Spring ассоциирует с термином «реактивный», является *неблокирующая нагрузка* поставщика. При синхронной обработке запросов сама блокировка потока вынуждает вызывающую сторону ожидать снятия блокировки и ограничивает количество поступающих данных в обработчик.

При реактивной обработке важно контролировать частоту возникновения событий, чтобы не перегрузить обрабатывающую сторону.

Проекты virtualpets-server-springframework и virtualpets-server-springboot не задействуют Spring WebFlux и не содержат примеров его использования. Вы можете попробовать самостоятельно перевести их на реактивный стек, если вас интересует подобный опыт.

Для перевода проектов на реактивный стек необходимо как минимум подключить зависимость от `spring-boot-starter-webflux` — для Spring Boot и от `spring-webflux` — для Spring Framework.

После подключения зависимости следует во всех контроллерах заменить возвращаемые методами значения на `Mono`. Допускается использовать и Flux, если метод возвращает несколько значений, но в этом случае могут появиться проблемы с возвратом корректной ошибки, если она возникнет в процессе формирования очередного значения.

Например, в методе `getUserPets` контроллера `PetController` (листинг 11.87)...

Листинг 11.87. PetController.java

```
@RequestMapping(value = "getUserPets", method = RequestMethod.GET)
public PetListResult getUserPets(
        @AuthenticationPrincipal UserDetailsImpl userDetailsImpl)
                throws ServiceException {
    return petService.getUserPets(
            new UserPetDetails(
                userDetailsImpl.getUserId(),
                selectedPet.getPetId()));
}
```

...необходимо сменить тип возвращаемого значения `PetListResult` на `Mono<PetListResult>` (листинг 11.88).

Листинг 11.88. PetController.java

```
@RequestMapping(value = "getUserPets", method = RequestMethod.GET)
public Mono<PetListResult> getUserPets(
        @AuthenticationPrincipal UserDetailsImpl userDetailsImpl)
                throws ServiceException {
    return petService.getUserPets(
            new UserPetDetails(
                userDetailsImpl.getUserId(),
                selectedPet.getPetId()));
}
```

При этом, разумеется, потребуется переделать метод сервиса `PetServiceImpl#getUserPets` так, чтобы он возвращал `Mono` и использовал только неблокирующие вызовы. Переделывая подобным образом постепенно каждый из методов, вы в конечном итоге приведете весь проект к реактивному стилю. При этом необходимо быть очень внимательным, чтобы случайно не оставить вызов синхронного блокирующего API какой-либо библиотеки или фреймворка, что повлечет за собой сильнейшую деградацию производительности всего реактивного приложения.

Если существует уже созданное и успешно эксплуатируемое приложение на Spring Web MVC, то стоит ли его переписывать на Spring WebFlux? Рефакторинг приложения с одного стека на другой — это весьма ресурсоемкая операция. В реальности при этом придется практически полностью переписать всю логику приложения и сменить все используемые блокирующие технологии на их реактивную альтернативу:

- Spring Web MVC → Spring WebFlux;
- JDBC → R2DBC;
- Spring Security → переделать настройку на реактивный стек;
- Spring Data → Spring Data R2DBC;
- RestClient → WebClient;

и т. д.

В каких случаях стоит использовать Spring WebFlux или переходить на него?

- Если ваше Spring MVC приложение уже существует и справляется со своей работой, то нет необходимости что-либо менять. Любой рефакторинг, любой переход с одного стека на другой, любой переход с одного фреймворка на другой, любой переход с одного языка программирования на другой должны иметь определенную цель. Просто переписывание без конкретной цели не имеет смысла. К тому же при стандартной синхронной обработке входящего запроса в одном, специально для этого выделенном потоке вам будет доступно максимальное количество библиотек, т. к. исторически обработка была именно блокирующей, синхронной.
- При использовании микросервисной архитектуры имеется возможность для каждого сервиса выбирать, на какой технологии он будет построен: Spring MVC

или Spring WebFlux — в зависимости от специфики работы сервиса и его нагрузки.

- Один из самых простых способов проверить пригодность вашего приложения для реактивного стека — это посмотреть его зависимости. Если используются блокирующие API, наподобие JPA, JDBC, сетевые и т. д., то лучшим выбором станет использование Spring MVC.
- Если ваша команда достаточно большая, то следует иметь в виду, что процесс изучения реактивного неблокирующего программирования достаточно сложен. Каждому члену команды придется усвоить идею реактивного программирования, особенности его использования и в первую очередь понять, почему нельзя допускать блокирования реактивных потоков.

11.6. Резюме

Spring содержит все необходимое для создания современных веб-приложений. Аннотация `@RestController` позволяет обрабатывать HTTP-запросы. Интеграция с Jackson позволяет использовать JSON для сериализации и десериализации тела запросов и ответов. Поддержка Jakarta Validation задействуется для проверки допустимых значений.

Spring поддерживает различные шаблонизаторы, позволяющие формировать HTML-страницы на сервере. В современной разработке веб-приложений для полноценных интерфейсов зачастую используются различные библиотеки и фреймворки JavaScript, но поддержка шаблонизаторов позволяет при необходимости создавать фронтенд силами бэкенд-разработчиков:

- в *разд. 11.3* рассматривается разработка фронтенда с помощью Jakarta Pages на примере проекта virtualpets-server-springframework;
- в *разд. 11.4* рассматривается разработка фронтенда с помощью Thymeleaf на примере проекта virtualpets-server-springboot.

Проект Spring WebFlux позволяет использовать реактивный подход к разработке веб-приложений, в отличие от стандартного синхронного подхода Spring Web MVC.

ГЛАВА 12

Фреймворк Spring Security

12.1. Архитектура Spring Security

Spring Security — это фреймворк, обеспечивающий аутентификацию, авторизацию и защиту от стандартных атак. Spring Security фактически является стандартом для слоя безопасности приложений Spring — как для Spring Web MVC, так и для Spring WebFlux.

Для начала необходимо определиться с терминами «аутентификация» и «авторизация». Они обозначают совершенно разные процедуры, несмотря на то, что пишутся довольно похоже, — из-за схожести написания их значения часто путают.

> *Аутентификация* — процедура проверки подлинности пользователя по введенному паролю, по сертификату и т. д.
>
> *Авторизация* — предоставление определенному лицу или группе лиц прав на выполнение какой-либо операции, а также процесс проверки этих прав перед выполнением действий.

Проекты virtualpets-server-springframework и virtualpets-server-springboot активно используют Spring Security. При этом virtualpets-server-springframework ориентирован на XML-конфигурацию, а virtualpets-server-springboot — на Java-конфигурацию.

XML-конфигурация и Java-конфигурация взаимозаменяемы — для каждого XML-тега существует способ достичь аналогичного результата с помощью Java-конфигурации. В дальнейшем в книге при описании настройки Spring Security приводятся листинги как для XML-конфигурации — для проекта virtualpets-server-springframework, так и соответствующая Java-конфигурация, используемая в проекте virtualpets-server-springboot.

Архитектура Spring Security построена на основе фильтров из спецификации Jakarta EE. Под *фильтром* понимается класс, реализующий интерфейс `jakarta.servlet.Filter`. Каждый запрос клиента проходит через `jakarta.servlet.FilterChain`, содержащий экземпляры интерфейса `Filter`, и только после прохождения цепочки фильтров `FilterChain` он попадает в `Servlet`. В качестве Servlet'а в приложении Spring MVC выступает экземпляр `DispatcherServlet`. Каждый из экземпляров `Filter` проделывает

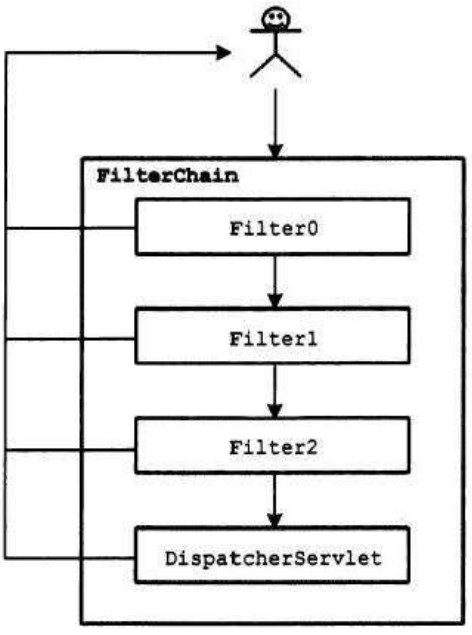


Рис. 12.1. Прохождение запроса по цепочке фильтров `FilterChain`

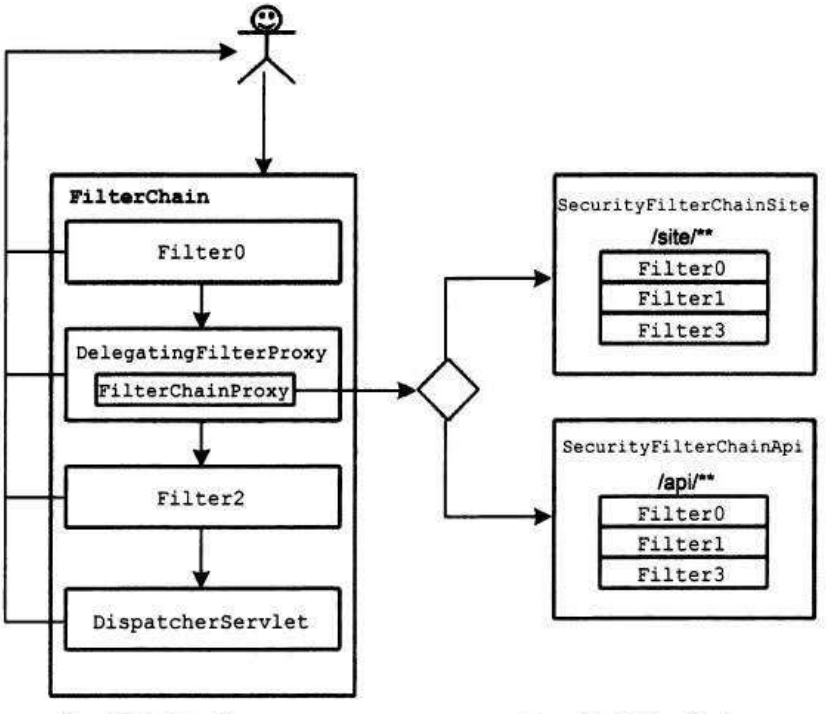


Рис. 12.2. Обработка запросов с несколькими `SecurityFilterChain`

свою работу и либо передает запрос следующему фильтру из цепочки вызовом метода `FilterChain#doFilter`, либо возвращает ответ. Последний фильтр из `FilterChain` передает запрос на обработку в `DispatcherServlet`. Общая схема прохождения запроса по фильтрам приведена на рис. 12.1.

Реализации `Filter` работают вне контекста Spring, поэтому им недоступны его бины. Для решения этой проблемы Spring предоставляет класс `org.springframework.web.filter.DelegatingFilterProxy`, реализующий интерфейс `Filter`. Класс `DelegatingFilterProxy` ищет бины, реализующие интерфейс `Filter`, и затем вызывает их. Бины `Filter`, находящиеся в контексте Spring, могут использовать все механизмы автосвязывания и все возможности, предоставляемые другим бинам.

Spring Security содержит класс `FilterChainProxy`, реализующий интерфейс бинов. Этот класс использует классы `SecurityFilterChain` для определения цепочки фильтров, которые необходимо выполнить для текущего запроса. В приложении может быть настроено несколько `SecurityFilterChain` для разных запросов (рис. 12.2).

12.2. Подключение к проекту

Подключение необходимых зависимостей в virtualpets-server-springframework показано в листинге 12.1.

Листинг 12.1. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>${org.springframework-security-version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>${org.springframework-security-version}</version>
</dependency>
```

При этом версия Spring Security задается в разделе `properties` (листинг 12.2).

Листинг 12.2. Файл pom.xml

```
<properties>
...
    <org.springframework-security-version>6.3.1</org.springframework-security-version>
...
</properties>
```

Проект virtualpets-server-springboot использует Spring Boot, поэтому ему достаточно подключить только зависимость от стартера (листинг 12.3).

Листинг 12.3. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

12.3. Конфигурация

Конфигурация Spring Security в проекте virtualpets-server-springframework находится в файле src/main/webapp/WEB-INF/spring/security.xml.

Проект virtualpets-server-springboot содержит идентичную конфигурацию Spring Security, но в виде Java-конфигурации в файле src/main/java/ru/urvanov/virtualpets/server/config/SecurityConfig.java.

Как уже отмечалось ранее, оба проекта настраивают Spring Security практически идентично. При желании можно сравнивать оба способа и смотреть, как сделана определенная настройка в XML и как она же осуществляется с помощью Java-конфигурации.

Основное пространство имен XML-конфигурации Spring Security обычно ассоциируется с префиксом `security` (листинг 12.4).

Листинг 12.4. Файл security.xml

```
xmlns:security = "http://www.springframework.org/schema/security"
```

В листинге 12.5 показано начало конфигурации с указанием всех используемых файлом security.xml пространств имен.

Листинг 12.5. Файл security.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns = "http://www.springframework.org/schema/beans"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p = "http://www.springframework.org/schema/p"
    xmlns:util = "http://www.springframework.org/schema/util"
    xmlns:security = "http://www.springframework.org/schema/security"
    xmlns:http = "http://www.springframework.org/schema/http"
    xsi:schemaLocation = "http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security.xsd
    http://www.springframework.org/schema/http
    http://www.springframework.org/schema/http/spring-http.xsd
     ">
```

В проекте virtualpets-server-springboot задействована Java-конфигурация, поэтому в файле SecurityConfig.java не описываются пространства имен. В нем импортируются используемые классы из различных пакетов Spring Security (листинг 12.6).

Листинг 12.6. Файл SecurityConfig.java

```
package ru.urvanov.virtualpets.server.config;

import java.util.List;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.ProviderManager;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import org.springframework.security.config.Customizer;
import
org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.context.DelegatingSecurityContextRepository;
import org.springframework.security.web.context.HttpSessionSecurityContextRepository;
import org.springframework.security.web.context.RequestAttributeSecurityContextRepository;
import org.springframework.security.web.context.SecurityContextRepository;

import ru.urvanov.virtualpets.server.auth.CustomAuthenticationEntryPoint;

@Configuration
@EnableMethodSecurity
public class SecurityConfig {
...
```

После ассоциации префиксов с пространствами имен для XML-конфигурации либо импорта пакетов для Java-конфигурации начинается настройка Spring Security.

Сервер виртуальных питомцев имеет различные области, к каждой из которых необходима своя конфигурация безопасности:

1. Общедоступный сайт — это все генерируемые Jakarta Pages и Thymeleaf страницы с информацией об игре, со статистикой сервера, технической информацией о сервере и т. п. Эти страницы должны быть доступны всем пользователям Интернета без необходимости регистрации аккаунта в игре.
2. Страницы сайта, доступные игрокам после ввода логина и пароля.
3. Страницы сайта, доступные администраторам после ввода логина и пароля.
4. API серверной части игры, необходимое для регистрации нового пользователя, точки входа в игру и получения информации о сервере. К этому API должен быть доступ у всех пользователей Интернета без необходимости регистрации аккаунта в игре.

5. API серверной части игры, к которой обращается клиент JavaScript после ввода пользователем логина и пароля. К этому API имеют доступ только зарегистрированные пользователи с незаблокированными аккаунтами.

Область сайта, доступная абсолютно всем, настраивается проще всего. Все страницы расположены по подадресам адреса **currentApplicationContext/site**, а значит, необходимо разрешить всем без исключения доступ к этим адресам.

Пример XML-конфигурации Spring Security, разрешающей доступ всем пользователям к адресу **currentApplicationContext/site** и всем его подстраницам, приведен в листинге 12.7.

Листинг 12.7. Файл security.xml

```
<security:http pattern = "/site/**" security = "none" />
```

В листинге 12.8 приведен пример аналогичной Java-конфигурации Spring Security из проекта virtualpets-server-springboot.

Листинг 12.8. Файл SecurityConfig.java

```
@Bean
public SecurityFilterChain securityFilterChainSite(
        HttpSecurity http) throws Exception {
    return http
            .securityMatcher( "/site/**")
            .authorizeHttpRequests((authorize) ->
                authorize.anyRequest().permitAll()
            )
            .build();
}
```

Код из обоих этих листингов делает одно и то же. Он взаимозаменяем. При использовании Java-конфигурации в проекте на Spring Framework даже без Spring Boot код из листинга 12.8 привел бы к аналогичному результату.

Экземпляр класса `HttpSecurity`, передаваемый в метод `securityFilterChainSite` (см. листинг 12.8), представляет собой аналог тега `security:http` для Java-конфигурации (см. листинг 12.7).

Атрибут `pattern` тега `security:http` используется для указания шаблона URL, к которому применяется вся оставшаяся конфигурация тега. Аналогом атрибута `pattern` в классе `HttpSecurity` является метод `securityMatcher`. В коде приведенных примеров как в XML-конфигурации (см. листинг 12.7), так и в Java-конфигурации (см. листинг 12.8), настройка будет применена для URL, начинающихся с `/site`.

Атрибут `security = "none"` в XML-конфигурации делает доступными адреса с `/site`, указанные в `pattern`, абсолютно всем. Аналогичная конфигурация в Java осуществляется в методе `authorizeHttpRequests`, в который передается лямбда-выражение, разрешающее доступ к любому адресу любым пользователям. Метод `authorizeHttpRequests` отрабатывает только после успешной валидации методом

securityMatcher, поэтому все пользователи получат доступ только к адресам, начинающимся с /site, как указано в securityMatcher.

12.4. Интерфейсы *UserDetails* и *UserDetailsService*

Вся необходимая для аутентификации пользователей информация находится в таблице user:

- поле login содержит логин, задействуемый при аутентификации на сайте или в клиенте игры;
- поле password содержит хеш пароля, используемого при аутентификации на сайте или в клиенте игры;
- поле roles содержит список ролей, разделенных запятыми. В игре виртуальных питомцев только две роли. В более сложных случаях, возможно, может потребоваться создание дополнительных таблиц для хранения прав доступа и ролей;
- поле enabled содержит флаг разрешения на вход в игру и на сайт.

Информацию, хранящуюся в этих полях, необходимо каким-либо образом передать Spring Security для использования в механизмах аутентификации и авторизации. Для этого используются интерфейсы UserDetailsService и UserDetails из пакета org.springframework.security.core.userdetails.

Интерфейс UserDetailsService содержит один метод — loadUserByUsername, возвращающий интерфейс UserDetails, и используется Spring Security для получения заполненной данными структуры с информацией о пользователе по его логину (листинг 12.9).

Листинг 12.9. UserDetailsService.java

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundExeption;
```

Интерфейс UserDetails, реализации которого предоставляют доступ к информации о пользователе, содержит следующие методы (листинг 12.10).

Листинг 12.10. UserDetails.java

```
public interface UserDetails extends Serializable {

    Collection<? extends GrantedAuthority> getAuthorities();

    String getPassword();

    String getUsername();
```

```
    default boolean isAccountNonExpired() {
        return true;
    }

    default boolean isAccountNonLocked() {
        return true;
    }

    default boolean isCredentialsNonExpired() {
        return true;
    }

    default boolean isEnabled() {
    }
}
```

Методов достаточно много. В проекте игры виртуальных питомцев используется лишь часть этих методов. Существует стандартная реализация `User` из того же пакета `org.springframework.security.core.userdetails`, которую можно использовать напрямую для своих проектов, наследоваться от нее либо описать свою реализацию `UserDetails` самостоятельно.

Какой бы способ ни был выбран в вашем проекте, необходимо понимать, что означает тот или иной метод интерфейса `UserDetails`. Значение методов `getUsername` и `getPassword` очевидно, но со смыслом остальных методов может возникнуть путаница.

- Метод `getAuthorities` возвращает коллекцию полномочий пользователя. В случае с игрой виртуальных питомцев коллекция полномочий представляет собой список ролей пользователя.
- Следующие четыре метода с первого взгляда может быть сложно отличить друг от друга. Чем различаются `isAccountNonLocked` и `isEnabled`? И чем различаются `isAccountNonExpired` и `isCredentialsNonExpired`?
  - Метод `isEnabled` отображается на поле `enabled` в таблице `user` в случае с игрой виртуальных питомцев. Он представляет собой флаг, определяющий, что учетная запись не была заблокирована администратором или сервером по какой-либо причине. Пользователь с заблокированным аккаунтом (если `isEnabled` возвращает `false`) не может войти в систему. Для разблокировки пользователя обычно необходимо какое-либо конкретное действие с его стороны, если разблокировка вообще возможна.
  - Метод `isAccountNonLocked` может показаться похожим на `isEnabled`. Он указывает, что учетная запись не была заблокирована, например после неудачных попыток входа. Действие от пользователя для разблокировки обычно не требуется. В примере игры про виртуальных питомцев метод `isAccountNonLocked` не используется.
  - Метод `isAccountNonExpired` указывает, что срок действия учетной записи не истек. Пользователь с истекшим сроком действия аккаунта не может войти

в систему. Метод этот необходим для систем, в которых пользователи создаются на определенный срок, после которого необходимо его продлевать, если они нужны. В примере игры про виртуальных питомцев этот метод не используется.

- Метод `isCredentialsNonExpired` представляет собой флаг, определяющий, что срок действия пароля не истек. Пользователь с истекшим паролем не может войти в систему. В примере игры про виртуальных питомцев этот метод также не используется.

Пример приложения с игрой про виртуальных питомцев использующий реализацию `UserDetailsImpl`, расширяющую класс `User` из Spring Security и добавляющую поля с первичным ключом пользователя и его полным именем, приведен в листинге 12.11.

Листинг 12.11. UserDetailsImpl.java

```
package ru.urvanov.virtualpets.server.auth;

import java.util.Collection;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.User;

/**
 * Реализация интерфейса
 * {@link org.springframework.security.core.userdetails.UserDetails}.
 * Добавляет два поля:
 * <ul>
 *   <li>
 *   userId - первичный ключ пользователя,
 *   </li>
 *   <li>
 *   name - полное отображаемое имя пользователя.
 *   </li>
 * </ul>
 */
public class UserDetailsImpl extends User {

    private static final long serialVersionUID = -3285304553448604871L;

    /**
     * Первичный ключ пользователя.
     */
    private Integer userId;

    /**
     * Полное отображаемое имя пользователя.
     */
    private String name;
```

```
    /**
     * Инициализирует экземпляр UserDetailsImpl.
     * @param userId Первичный ключ пользователя.
     * @param username Логин.
     * @param name Полное отображаемое имя пользователя.
     * @param password Пароль.
     * @param enabled Учетная запись включена.
     * @param authorities Список ролей.
     */
    public UserDetailsImpl(Integer userId, String username, String name,
            String password,
            boolean enabled,
            Collection<? extends GrantedAuthority> authorities) {
        super(username, password, enabled, true, true, true,
                authorities);
        this.userId = userId;
        this.name = name;
    }

    /**
     * @return Первичный ключ пользователя
     */
    public Integer getUserId() {
        return userId;
    }

    /**
     * @return Полное отображаемое имя пользователя.
     */
    public String getName() {
        return name;
    }
}
```

Класс `UserDetailsImpl` выглядит одинаково как для проекта virtualpets-server-springframework, так и для проекта virtualpets-server-springboot. Обратите внимание, что он имеет единственный конструктор, который принимает все необходимые параметры, сохраняет значение первичного ключа и полное имя в своих полях, а все остальные значения передает в конструктор базового класса `User`. Класс `UserDetailsImpl` не имеет методов установки значений — только методы получения значений из его полей, что должно защитить их от случайного изменения.

Сервис `UserDetailsServiceImpl`, реализующий интерфейс `UserDetailsService`, содержит только один публичный метод — `loadByUsername`, реализующий соответствующий метод из интерфейса (листинг 12.12).

Листинг 12.12. UserDetailsServiceImpl.java

```
package ru.urvanov.virtualpets.server.service;

import java.util.Collection;
import java.util.HashSet;
```

```
import java.util.Optional;
import java.util.Set;
import java.util.StringTokenizer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import ru.urvanov.virtualpets.server.auth.UserDetailsImpl;
import ru.urvanov.virtualpets.server.dao.UserDao;
import ru.urvanov.virtualpets.server.dao.domain.User;

/**
 * Сервис получения экземпляра {@link UserDetails} для слоя безопасности.
 */
@Service("userDetailsService")
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserDao userDao;

    /**
     * Основной метод сервиса. Загружает экземпляр
     * {@link UserDetailsImpl} из базы данных.
     * @return Заполненный экземпляр {@link UserDetailsImpl}.
     */
    @Override
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        Optional<User> user = userDao.findByLogin(username);
        return user.map(u -> new UserDetailsImpl(
                u.getId(),
                u.getLogin(),
                u.getName(),
                u.getPassword(),
                u.isEnabled(),
                this.fillAuthorities(u)))
                .orElseThrow(() ->  new UsernameNotFoundException(username));
    }

    /**
     * Заполняет коллекцию полномочий на основе строки
     * {@link User#getRoles()}. Строка содержит список ролей, разделенных запятой.
     * @param user Информация о пользователе из слоя постоянства.
     * @return Заполненную коллекцию полномочий, как того требует
     * Spring Security.
     */
```

```
    private Collection<? extends GrantedAuthority> fillAuthorities(
            User user) {
        Set<GrantedAuthority> granted = new HashSet<GrantedAuthority>();
        StringTokenizer t = new StringTokenizer(user.getRoles(), ",");
        t.asIterator().forEachRemaining((role) ->
                granted.add(new SimpleGrantedAuthority("ROLE_" + role)));
        return granted;
    }
}
```

Приватный метод `fillAuthorities` формирует коллекцию полномочий из строкового представления слоя постоянства. Префикс `ROLE_` — это стандартный префикс ролей Spring Security. Колонка `roles` таблицы `user` хранит список ролей, разделенных запятыми. Spring Security по умолчанию требует наличия префикса `ROLE_` для ролей и использует интерфейс `GrantedAuthority` для их представления.

Интерфейс `GrantedAuthority` в рамках тестового приложения сервера виртуальных питомцев используется как роль, поскольку в сервере виртуальных питомцев реализуется разделение прав доступа на основе ролей. Сам интерфейс `GrantedAuthority` — это не роль, это именно полномочия, он может хранить право доступа на какое-либо действие и т. д.

12.5. Интерфейсы *AuthentiationManager* и *Authentication Provider*

Интерфейс `AuthenticationManager` из пакета `org.springframework.security.authentication` определяет, как фильтры Spring Security выполняют аутентификацию. Наиболее часто используемая реализация интерфейса `AuthenticationManager` — это класс `ProviderManager`.

`ProviderManager` содержит список экземпляров классов, реализующих интерфейс `AuthenticationProvider`.

Каждая реализация интерфейса `AuthenticationProvider` определяет свой способ аутентификации:

- `DaoAuthenticationProvider`;
- `OAuth2AuthenticationProvider`;
- `LdapAuthenticationProvider`;
- `NullAuthenticationProvider`;
- `JwtAuthenticationProvider`;
- `OpenSaml4AuthenticationProvider`;
- `RememberMeAuthenticationProvider`;
- и ряд других.

`ProviderManager` проходит по списку `AuthenticationProvider`, как показано на рис. 12.3, и предоставляет каждому из них возможность осуществить аутентификацию. Если

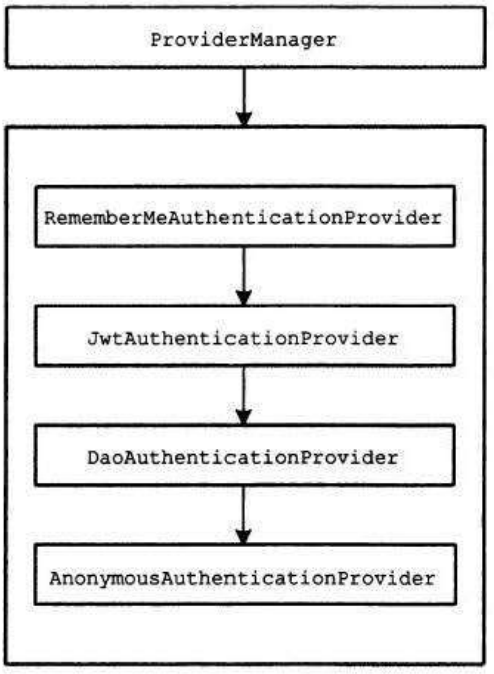


Рис. 12.3. ProviderManager **и список** AuthenticationProvider

ни один из AuthenticationProvider не может аутентифицировать запрос, то процесс аутентификации завершается с исключением ProviderNotFoundException.

В XML-конфигурации (файл security.xml) проекта virtualpets-server-springframework AuthenticationManager объявляется с помощью тега authentication-manager, в котором с помощью вложенных тегов authentication-provider указываются реализации AuthenticationProvider (листинг 12.13).

Листинг 12.13. Файл security.xml

```
<security:authentication-manager
        alias = "authenticationManager">
    <security:authentication-provider
        ref = "authenticationProvider" >
    </security:authentication-provider>
</security:authentication-manager>
```

Аналогичная конфигурация для проекта virtualpets-server-springboot и Java-конфигурации приведена в листинге 12.14.

Листинг 12.14. SecurityConfig.java

```
@Bean
public AuthenticationManager authenticationManager(
        DaoAuthenticationProvider authenticationProvider) {
    return new ProviderManager(List.of(authenticationProvider));
}
```

Бин `authenticationProvider` — это `DaoAuthenticationProvider`, использующий `UserDetailsService` для получения данных о пользователе (листинг 12.15).

Листинг 12.15. Файл security.xml

```
<bean id = "authenticationProvider"
    class = "org.springframework.security.authentication.dao.DaoAuthenticationProvider">
    <property name = "userDetailsService" ref = "userDetailsService" />
    <property name = "passwordEncoder" ref = "passwordEncoder" />
</bean>
```

Бин `passwordEncoder` — это бин, отвечающий за хеширование паролей. Проекты virtualpets-server-springframework и virtualpets-server-springboot не хранят пароли в явном виде — в базе данных в поле `password` таблицы `user` хранятся лишь их хеши, полученные помощью алгоритма bcrypt, специально предназначенного для генерации хешей паролей. Аналогично стоит поступать и в реальных проектах. Хранить пароли в открытом виде в базе данных нельзя.

Код конфигурирования бина `passwordEncoder` в virtualpets-server-springframework приведен в листинге 12.16.

Листинг 12.16. SecurityConfig.java

```
<bean id = "passwordEncoder"
    class = "org.springframework.security.crypto.factory.PasswordEncoderFactories"
    factory-method = "createDelegatingPasswordEncoder" />
```

Аналогичная конфигурация бинов `authenticationProvider` и `passwordEncoder` для Java-конфигурации (проект virtualpets-server-springboot) приведена в листинге 12.17.

Листинг 12.17. SecurityConfig.java

```
@Bean
public DaoAuthenticationProvider authenticationProvider(
        UserDetailsService userService,
        PasswordEncoder passwordEncoder) {
    DaoAuthenticationProvider daoAuthenticationProvider
            = new DaoAuthenticationProvider();
    daoAuthenticationProvider.setUserDetailsService(userService);
    daoAuthenticationProvider.setPasswordEncoder(passwordEncoder);
    return daoAuthenticationProvider;
}

@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories
            .createDelegatingPasswordEncoder();
}
```

Почему в качестве `passwordEncoder` не создается экземпляр `BCryptPasswordEncoder` напрямую? До Spring Security 5.0 именно так и поступали разработчики в большинстве

проектов. Подобный подход вполне работоспособен, и он будет работоспособен и далее. Его недостаток лишь в том, что при смене алгоритма хеширования паролей старые пароли перестают работать. `DelegatingPasswordEncoder`, создаваемый фабричным методом `createDelegatingPasswordEncoder()` класса `PasswordEnoderFactories`, поддерживает различные алгоритмы хеширования, выбирая подходящий алгоритм по записанному в фигурных скобках названию алгоритма.

Например, для хеша пароля:

```
{bcrypt}$2a$10$JT018oNHQuohL8SMLHCBludsjTiJNpG.uDHc3QGkP5V.aMMLSEa7G
```

используется алгоритм `bcrypt`.

Для создания хешей новых паролей `DelegatingPasswordEncoder` применит актуальный на текущий момент алгоритм.

12.6. *SecurityContextRepository*

После аутентификации необходимо где-либо хранить информацию о пользователе — слой сервисов и слой контроллеров должны иметь возможность проверить, прошел ли пользователь, инициировавший запрос, аутентификацию, а также возможность получить заполненный экземпляр `UserDetails`.

`SecurityContextRepository` занимается сохранением информации о пользователе для будущих запросов. Обычно используется `DelegatingSecurityContextRepository`, содержащий список `SecurityContextRepository`, которым он делегирует свою работу. В проекте виртуальных питомцев как в варианте для Spring Framework, так и в варианте для Spring Boot задействуются следующие реализации `SecurityContextRepository`:

- `RequestAttributeSecurityContextRepository` — сохраняет информацию о пользователе в `SecurityContext` как атрибут запроса, что делает ее доступной в рамках одного запроса;
- `HttpSessionSecurityContextRepository` — сохраняет информацию о пользователе в `HttpSession`, что делает ее доступной для последующих запросов в рамках одной HTTP-сессии.

Конфигурирование `SecurityContextRepository` с помощью XML-конфигурации в проекте virtualpets-server-springframework показано в листинге 12.18.

Листинг 12.18. Файл security.xml

```
<bean id = "securityContextRepository"
        class = "org.springframework.security.web.context.DelegatingSecurityContextRepository">
    <constructor-arg ref = "securityContextRepositoryList" />
</bean>

<util:list id = "securityContextRepositoryList">
    <bean class =
       "org.springframework.security.web.context.RequestAttributeSecurityContextRepository" />
    <bean class =
           "org.springframework.security.web.context.HttpSessionSecurityContextRepository" />
</util:list>
```

Аналогичный результат для Java-конфигурации и проекта virtualpets-server-springboot может быть достигнут с помощью следующего кода (листинг 12.19).

Листинг 12.19. SecurityConfig.java

```
@Bean
public SecurityContextRepository securityContextRepository() {
    return new DelegatingSecurityContextRepository(
            new RequestAttributeSecurityContextRepository(),
            new HttpSessionSecurityContextRepository());
}
```

Сохраненный `UserDetails` доступен в процессе обработки запроса в `SecurityContextHolder`:

```
UserDetails userDetails = SecurityContextHolder.getContext()
        .getAuthentication().getPrincipal();
```

Тем не менее в игре виртуальных питомцев описанный здесь способ не применяется. Вместо него используется аннотация `@AuthentiationPrincipal` (листинг 12.20).

Листинг 12.20. PetController.java

```
@RequestMapping(value = "getUserPets", method = RequestMethod.GET)
public PetListResult getUserPets(
        @AuthenticationPrincipal UserDetailsImpl userDetailsImpl)
                throws ServiceException {
    return petService.getUserPets(
            new UserPetDetails(
                userDetailsImpl.getUserId(),
                selectedPet.getPetId()));
}
```

Для работы аннотации `@AuthenticationPrincipal` в Spring Framework необходимо объявление бина `AuthenticationPrincipalArgumentResolver`, как это сделано в файле servlet-context.xml (листинг 12.21).

Листинг 12.21. Файл servlet-context.xml

```
<!-- Поддержка @Controller-->
<annotation-driven>
    <argument-resolvers>
        <!-- Поддержка @AuthenticationPrincipal -->
        <beans:bean class = "org.springframework.security.web.method.annotation.
                                                AuthenticationPrincipalArgumentResolver" />
    </argument-resolvers>
</annotation-driven>
```

Для Spring Boot никакой дополнительной конфигурации не нужно — настройка `AuthenticationPrincipalArgumentResolver` происходит автоматически.

12.7. Раздел сайта *SecurityFilterChain*

12.7.1. Зоны доступа

На сервере виртуальных питомцев у сайта есть не только доступные всем пользователям Интернета разделы, но и разделы, доступные только игрокам и только администраторам.

Детализированное разделение на зоны доступа сайта сервера виртуальных питомцев выглядит так:

- зона **/site/admin** со всеми подстраницами доступна только администраторам;
- зона **/site/user** со всеми подстраницами доступна только пользователям;
- зона **/site** с остальными страницами, кроме указанных в предыдущих пунктах, доступна абсолютно всем.

12.7.2. Тег *security:http* и метод *securityMatcher*

Для описания `SecurityFilterChain` в XML-конфигурации используется тег `security:http`, в атрибуте `pattern` которого указывается шаблон адреса, для которого применяется описываемый `SecurityFilterChain`. При этом обязательно необходимо указать ссылку на `AuthenticationManager` (листинг 12.22).

Листинг 12.22. Файл security.xml

```
<security:http pattern = "/site/**"
        authentication-manager-ref = "authenticationManager">
...
</security:http>
```

Аналогичная конфигурация для проекта virtualpets-server-springboot и Java-конфигурации приведена в листинге 12.23.

Листинг 12.23. SecurityConfig.java

```
@Bean
public SecurityFilterChain securityFilterChainSite(
        HttpSecurity http,
        AuthenticationManager authenticationManager) throws Exception {
    return http
            .securityMatcher( "/site/**")
            .authenticationManager(authenticationManager)
            // ... Остальная часть конфигурации
            .build();
}
```

Здесь:

- `HttpSecurity` — это аналог тега `security:http` из XML-конфигурации;
- `HttpSecurity#securityMatcher` — это аналог атрибута `pattern` тега `security:http`;

- `HttpSecurity#authenticationManager` — это аналог атрибута `authentication-manager-ref` из XML-конфигурации.

Далее нам необходимо описать три зоны сайта внутри шаблона, указанного в атрибуте `pattern` и методе `HttpSecurity#securityMatcher`.

12.7.3. Тег *security:intercept-url* и метод *authorizeHttpRequests*

Тег `security:intercept-url`, помещенный в тег `security:http`, описывает с помощью атрибута `pattern` шаблон адресов, входящий во множество адресов из шаблона `security:http pattern`. Атрибут `access` позволяет указать SpEL (Spring Expression Language), определяющий доступ пользователей к адресам из атрибута `pattern` тега `security:intercept-url` (листинг 12.24).

Листинг 12.24. Файл security.xml

```
<security:http pattern = "/site/**"
        authentication-manager-ref = "authenticationManager">
    <security:intercept-url pattern = "/site/admin/**"
        access = "hasRole('ADMIN')" />
    <security:intercept-url pattern = "/site/user/**"
        access = "hasRole('USER')" />
    <security:intercept-url pattern = "/site/**"
        access = "permitAll" />
...
</security:http>
```

Spring Expression Language (SpEL) часто используется в Spring Security — он еще будет упомянут при описании авторизации на основе методов.

В дополнение к основным возможностям SpEL при использовании в Spring Security добавляются переменные:

- `authentication` — доступ к объекту `Authentication`, возвращаемому в `SecurityContextHolder.getContext().getAuthentication()`;
- `principal` — доступ к `UserDetails`, возвращаемому методом `Authentication#getPrincipal`;

а также следующие дополнительные методы:

- `permitAll` — запрос не нуждается в авторизации, адреса доступны всем. Код из листинга 12.24 настраивает `SecurityFilterChain` таким образом, что к адресам `/site/**`, кроме адресов `/site/admin/**` и `/site/user/**`, будут иметь доступ абсолютно все;
- `denyAll` — запрос запрещен всем и всегда;
- `hasAuthority` — для выполнения запроса пользователь должен иметь `GrantedAuthority` с указанным значением;

- `hasRole` — короткая запись `hasAuthority` с префиксом `ROLE_`. В игре с виртуальными питомцами активно используется именно этот вариант. Например, код `hasRole('ADMIN')` из листинга 12.24 разрешает доступ к адресам, удовлетворяющим шаблону из атрибута `pattern` тега `intercept-url` только пользователям с `GrantedAuthority ROLE_ADMIN`;
- `hasAnyAuthority` — для выполнения запроса пользователь должен иметь `GrantedAuthority` с одним из указанных значений;
- `hasAnyRole` — аналог `hasAnyAuthority`, но с префиксом `ROLE_` для полномочий из `GrantedAuthority`;
- `hasPermission` — более сложный вариант для авторизации на уровне объектов.

Обработка атрибутов `pattern` происходит в следующем порядке:

1. Поступает запрос на какой-нибудь адрес.
2. Адрес сопоставляется с шаблоном из `pattern` в `security:http`. Если сопоставление успешно, то смотрятся теги `security:intercept-url` внутри тега `security:http`, в противном случае обрабатывается следующий тег `security:http`.
3. Обрабатывается первый тег `security:intercept-url` в теге `security:http`. Адрес запроса сопоставляется с его шаблоном в `pattern`. Если сопоставление успешно, то применяется SpEL из атрибута `access` для прохождения процедуры авторизации. При успешной авторизации запрос проходит дальше на обработку и в конечном итоге попадает на обрабатывающий его контроллер. Если авторизация согласно SpEL заканчивается неуспешно, то обработка запроса прерывается и бросается исключение `AccessDeniedException`.
4. Если сопоставление с шаблоном в атрибуте `security:intercept-url` неуспешно, то обрабатывается следующий тег `security:intercept-url`.

Например, запрос на адрес **/site/user/profile** обрабатывается в следующем порядке:

1. Адрес **/site/user/profile** сопоставляется с шаблоном атрибута `pattern` тега `security:http`. Сопоставление успешно.
2. Адрес **/site/user/profile** сопоставляется с шаблоном атрибута `pattern` элемента `<security:intercept-url pattern = "/site/admin/**"`. Сопоставление неуспешно.
3. Адрес **/site/user/profile** сопоставляется с шаблоном атрибута `pattern` элемента `<security:intercept-url pattern = "/site/user/**"`. Сопоставление успешно.
4. Проходит проверка авторизации запроса по SpEL `hasRole('USER')`, и если авторизация завершается успешно, т. е. `SecurityContextHolder` хранит `Authorization` с `GrantedAuthority ROLE_USER`, то запрос передается на обработку. В противном случае возникает исключение `AccessDeniedException`.

Аналогичный результат выдает Java-конфигурация из проекта virtualpets-server-springboot:

```
@Bean
public SecurityFilterChain securityFilterChainSite(
        HttpSecurity http,
        AuthenticationManager authenticationManager) throws Exception {
```

```
    return http
            .securityMatcher( "/site/**")
            .authenticationManager(authenticationManager)
            .authorizeHttpRequests((authorize) ->
                authorize.requestMatchers("/site/admin/**")
                    .hasRole("ADMIN")
                .requestMatchers("/site/user/**")
                    .hasRole("USER")
                .requestMatchers("/site/**")
                    .permitAll()
            )
            ... остальная конфигурация
            .build();
}
```

12.7.4. Защита от CSRF

CSRF (Cross Site Request Forgery, межсайтовая подделка запроса) — эта атака на сайт, использующая недостаток протокола HTTP. Атака осуществляется размещением на сайте, куда заходит посетитель, ссылки или скрипта, осуществляющего отправку запроса на атакуемый сайт. Например, такого вида:

```
A <a href = "http://somebank.ru/transfer?to=9436540643587366amount=100">вот здесь</a> вы
можете посмотреть котиков.
```

Посетитель, зашедший на атакуемый сайт, разумеется, захочет посмотреть котиков и щелкнет на ссылке. В результате браузер отправит запрос на сайт `http://somebank.ru` с указанными в ссылке параметрами запроса. Если посетитель в этот момент был залогинен на сайте своего банка, т. е. у него была активная HTTP-сессия, то браузер подхватит сессионную «куку» и отправит ее в заголовках запроса. В конечном итоге это приведет к выполнению нежелательного запроса на атакуемом сайте от лица посетителя, если, конечно атакуемый сайт не имеет защиты от подобного типа атак.

Реализация защиты от CSRF в проекте virtualpets-server-springframework (XML-конфигурация) приведена в листинге 12.25.

Листинг 12.25. Файл security.xml

```
<security:http pattern = "/site/**"
        authentication-manager-ref = "authenticationManager">
...
    <security:csrf />
...
</security:http>
```

Аналогичная защита от CSRF в проекте virtualpets-server-springboot (Java-конфигурация) настраивается следующим образом (листинг 12.26).

Листинг 12.26. SecurityConfig.java

```
@Bean
public SecurityFilterChain securityFilterChainSite(
        HttpSecurity http,
        AuthenticationManager authenticationManager) throws Exception {

            .csrf(Customizer.withDefaults())
...
}
```

Однако одного только включения защиты недостаточно. Необходимо при формировании запросов вставлять дополнительное поле с генерируемым Spring Security значением, которое в последующем проверяется Spring Security. Теги `form` из библиотеки тегов Spring для Jakarta Pages автоматически добавляют скрытое поле с этим значением.

При создании формы без тега `form` из библиотеки тегов Spring необходимо добавлять на jsp/jspx-страницах скрытое поле самостоятельно:

```
<input type = "hidden"
    name = "${_csrf.parameterName}"
    value = "${_csrf.token}"/>
```

Если этого не сделать, то при проверке значения Spring Security бросит исключение, и вернется страница с отказом в доступе.

Пример добавления такого скрытого поля в файл src/main/webapp/WEB-INF/views/user/profile.jspx из проекта virtualpets-server-springframework приведен в листинге 12.27.

Листинг 12.27. Файл profile.jspx

```
<form method = "POST" action = "${logout_url}">
    <input type = "hidden"
        name = "${_csrf.parameterName}"
        value = "${_csrf.token}"/>
    <input type = "submit" value = "${logout_var}" />
</form>
```

В шаблонах Thymeleaf скрытое поле также необходимо добавлять самостоятельно. Пример добавления такого скрытого поля в файл шаблона src/main/resources/templates/user/profile.html из проекта virtualpets-server-springboot приведен в листинге 12.28.

Листинг 12.28. Файл profile.html

```
<form method = "POST" th:action = "@{/site/logout}">
    <input type = "hidden"
        th:name = "${_csrf.parameterName}"
        th:value = "${_csrf.token}"/>
    <input type = "submit" th:value = "#{virtualpets-server-springboot.site.logout}" />
</form>
```

Значение _csrf.token в обоих этих примерах генерируется бэкендом. Атакующий сайт не может его узнать, а значит, не сможет добавить его в свой запрос или ссылку.

12.7.5. Форма входа

Сайт игры виртуальных питомцев использует свою форму входа и в XML-конфигурации (проект virtualpets-server-springframework) настраивает адреса, как показано в листинге 12.29.

Листинг 12.29. Файл security.xml

```
<security:form-login login-page = "/site/login"
    login-processing-url = "/site/login"
    default-target-url = "/site/user/profile"
    authentication-failure-url = "/site/login?error=1" />
```

В приведенном здесь коде атрибуты тега security:form-login имеют следующие значения:

- атрибут login-page — адрес формы входа. Страница, возвращаемая бэкендом по этому адресу, должна содержать форму входа с полями ввода имени пользователя и пароля;
- атрибут login-processing-url — адрес обработки входа. POST-запросы на этот адрес обрабатываются фильтром Spring Security и осуществляют аутентификацию пользователя. Логин и пароль должны храниться в полях username и password соответственно;
- атрибут default-target-url — страница по умолчанию, отображаемая посетителю после успешного входа;
- атрибут authentication-failure-url — адрес, на который перебрасывает посетителя после неуспешной попытки входа.

JSPX-страница формы входа, возвращаемая при GET-запросах на /site/login, в проекте virtualpets-server-springframework находится по пути src/main/webapp/WEB-INF/views/login.jspx. Код формы входа приведен в листинге 12.30.

Листинг 12.30. Файл login.jspx

```
<form action = "${login_url}" method = "post">
    <div style = "width: 300px;" class = "form">
        <div>
            <input type = "text" name = "username"
                   placeholder = "${name_var}" />
        </div>

        <div>
            <input type = "password" name = "password"
                   placeholder = "${password_var}"/>
        </div>
```

```
        <input type = "hidden"
            name = "${_csrf.parameterName}"
            value = "${_csrf.token}"/>
        <c:if test = "${param.error != null}">
            <div class = "error">
                ${login_error_var}
            </div>
        </c:if>
        <c:if test = "${param.logout != null}">
            <div>
                ${logout_success_var}
            </div>
        </c:if>
        <div>
            <input type = "submit" value = "${login_var}" />
        </div>
    </div>
</form>
```

Однако одной только формы недостаточно — необходимо также организовать связь контроллера с JSPX-страницей (листинг 12.31).

Листинг 12.31. Файл login.jspx

```
<view-controller
    path = "site/login"
    view-name = "login" />
```

Форма входа шаблона Thymeleaf проекта virtualpets-server-springboot описана в файле src/main/resources/templates/login.html (листинг 12.32).

Листинг 12.32. Файл login.html

```
<form th:action = "@{/site/login}" method = "post">
    <div>
        <input type = "text" name = "username" th:placeholder =
            "#{virtualpets-server-springboot.site.username}" />
    </div>

    <div>
        <input type = "password" name = "password"
        th:placeholder = "#{virtualpets-server-springboot.site.password}"/>
    </div>

    <input type = "hidden"
        th:name = "${_csrf.parameterName}"
        th:value = "${_csrf.token}"/>
    <div th:if = "${param.error != null}" class = "error"
        th:text = "#{virtualpets-server-springboot.site.login-error}">
    </div>
```

```
    <div th:if = "${param.logout != null}"
        th:text = "#{virtualpets-server-springboot.site.logout-success}">
    </div>
    <div>
        <input type = "submit" th:value = "#{virtualpets-server-springboot.site.login}" />
    </div>
</form>
```

Связь контроллера с шаблоном Thymeleaf организована следующим образом (листинг 12.33).

Листинг 12.33. WebConfig.java

```
@Configuration
public class WebConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
...
        registry.addViewController("site/login").setViewName("login");
    }
...
}
```

12.7.6. Кнопка выхода

Обработка запросов успешного выхода из сайта виртуальных питомцев реализована в этих же шаблонах: login.jspx и login.html (см *разд. 12.7.5*) — для этого они проверяют параметр запроса `logout` (листинги 12.34 и 12.35).

Листинг 12.34. Файл login.jspx

```
<c:if test = "${param.logout != null}">
    <div>
        ${logout_success_var}
    </div>
</c:if>
```

Листинг 12.35. Файл login.html

```
<div th:if = "${param.logout != null}"
    th:text = "#{virtualpets-server-springboot.site.logout-success}">
</div>
```

Конфигурация Spring Security для virtualpets-server-springframework, настраивающая адрес выхода, приведена в листинге 12.36.

Листинг 12.36. Файл security.xml

```
<security:http pattern = "/site/**"
        authentication-manager-ref = "authenticationManager">
```

```
    <security:logout logout-url = "/site/logout"
        logout-success-url = "/site/login?logout=1" />
</security:http>
```

Атрибуты тега `security:logout` имеют здесь следующие значения:

- атрибут `logout-url` указывает адрес, по которому Spring Security обрабатывает выход посетителя;
- атрибут `logout-success-url` указывает адрес, который возвращает страницу, отображаемую после успешного выхода посетителя из системы.

Конфигурация Spring Security для проекта virtualpets-server-springboot, настраивающая аналогичный адрес выхода, приведена в листинге 12.37.

Листинг 12.37. SecurityConfig.java

```
@Bean
public SecurityFilterChain securityFilterChainSite(
        HttpSecurity http,
        AuthenticationManager authenticationManager) throws Exception {

            .logout((logout) ->
                logout.logoutUrl("/site/logout")
                    .logoutSuccessUrl("/site/login?logout=1")
            )
...
}
```

12.7.7. Листинги

Полный код XML-конфигурации `SecurityFilterChain` зоны сайта проекта virtualpets-server-springframework приведен в листинге 12.38.

Листинг 12.38. Файл security.xml

```
<security:http pattern = "/site/**"
        authentication-manager-ref = "authenticationManager">
    <security:intercept-url pattern = "/site/admin/**"
        access = "hasRole('ADMIN')" />
    <security:intercept-url pattern = "/site/user/**"
        access = "hasRole('USER')" />
    <security:intercept-url pattern = "/site/**"
        access = "permitAll" />
    <security:csrf />
    <security:form-login login-page = "/site/login"
        login-processing-url = "/site/login"
        default-target-url = "/site/user/profile"
        authentication-failure-url = "/site/login?error=1" />
    <security:logout logout-url = "/site/logout"
        logout-success-url = "/site/login?logout=1" />
</security:http>
```

Полный код Java-конфигурации SecurityFilterChain зоны сайта проекта virtualpets-server-springboot приведен в листинге 12.39.

Листинг 12.39. SecurityConfig.java

```
@Bean
public SecurityFilterChain securityFilterChainSite(
        HttpSecurity http,
        AuthenticationManager authenticationManager) throws Exception {
    return http
            .securityMatcher( "/site/**")
            .authenticationManager(authenticationManager)
            .authorizeHttpRequests((authorize) ->
                authorize.requestMatchers("/site/admin/**")
                    .hasRole("ADMIN")
                .requestMatchers("/site/user/**")
                    .hasRole("USER")
                .requestMatchers("/site/**")
                    .permitAll()
            )

            .csrf(Customizer.withDefaults())
            .formLogin((formLogin) ->
                formLogin.loginPage("/site/login")
                    .loginProcessingUrl("/site/login")
                    .defaultSuccessUrl("/site/user/profile")
                    .failureUrl("/site/login?error=1")
            )
            .logout((logout) ->
                logout.logoutUrl("/site/logout")
                    .logoutSuccessUrl("/site/login?logout=1")
            )
            .build();
}
```

12.8. *SecurityFilterChain* зоны API клиента

12.8.1. Тег *security:http* и метод *securityMatcher*

Spring Security поддерживает создание нескольких бинов SecurityFilterChain, что позволяет настроить разные способы доступа к разным адресам.

Пример игры виртуальных питомцев настраивает отдельный SecurityFilterChain для API, используемого клиентом JavaScript. Дополнительные SecurityFilterChain настраиваются аналогично первому:

1. Описывается тег security:http — в случае с XML-конфигурацией.
2. Описывается бин SecurityFilterChain с помощью методов класса HttpSecurity — в случае Java-конфигурации.

Тег `security:http` в проекте virtualpets-server-springframework описывается следующим образом (листинг 12.40).

Листинг 12.40. Файл security.xml

```
<security:http pattern = "/api/**"
        authentication-manager-ref = "authenticationManager"
        entry-point-ref = "authenticationEntryPoint"
        >
...
</security:http>
```

Аналогичная настройка для Java-конфигурации в проекте virtualpets-server-springboot приведена в листинге 12.41.

Листинг 12.41. SecurityConfig.java

```
@Bean
public SecurityFilterChain securityFilterChainApi(
        HttpSecurity http,
        AuthenticationManager authenticationManager,
        AuthenticationEntryPoint authenticationEntryPoint
        ) throws Exception {
    return http
        .securityMatcher("/api/**")
        .authenticationManager(authenticationManager)
        .exceptionHandling(e -> e.authenticationEntryPoint(authenticationEntryPoint))

        .build();
}
```

Код в листингах 12.40 и 12 41 имеет два отличия от аналогичных листингов из `SecurityFilterChain` сайта (см. *разд. 12.7.7*):

- здесь применен шаблон адреса `/api/**` — тогда как для сайта шаблон адреса `/site/**`;
- задействован новый интерфейс `AuthenticationEntryPoint`.

12.8.2. Интерфейс *AuthenticationEntryPoint*

Интерфейс `AuthenticationEntryPoint` отвечает за редирект посетителя на форму входа. При описании `SecurityFilterChain` сайта использовался тег `security:form-login`, который и осуществлял всю настройку `AuthenticationEntryPoint`. В случае API для клиента JavaScript сервер должен вернуть JSON-ответ вместо редиректа на форму входа. Реализация `AuthenticationEntryPoint` для `SecurityFilterChain` клиентского API одинакова и для virtualpets-server-springframework, и для virtualpets-server-springboot (листинг 12.42).

Листинг 12.42. CustomAuthenticationEntryPoint.java

```
package ru.urvanov.virtualpets.server.auth;

import java.io.IOException;

import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectWriter;

import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

/**
 * AuthenticationEntryPoint перебрасывает на форму входа.
 * В приложении сервера виртуальных питомцев только возвращает
 * ответ с 403 Forbidden и JSON с ошибкой.
 */
public class CustomAuthenticationEntryPoint
        implements AuthenticationEntryPoint {

    private ObjectWriter objectWriter = new ObjectMapper().writer();

    @Override
    public void commence(HttpServletRequest request,
            HttpServletResponse response,
            AuthenticationException authException)
            throws IOException, ServletException {
        HttpStatus statusCode = HttpStatus.FORBIDDEN;
        response.setStatus(statusCode.value());
        ProblemDetail problemDetail = ProblemDetail.forStatus(statusCode);
        problemDetail.setDetail(
                authException.getLocalizedMessage() == null
                        ? authException.getMessage()
                        : authException.getLocalizedMessage());
        response.getWriter().write(objectWriter.writeValueAsString(problemDetail));
    }
}
```

Метод `CustomAuthenticationEntryPoint#commence` **здесь просто возвращает клиенту JSON-ответ с описанием ошибки и с HTTP-статусом 403 Forbidden.**

Конфигурирование бина `authenticationEntryPoint` **в проекте virtualpets-server-springframework приведено в листингах 12.43 и 12.44.**

Листинг 12.43. Файл security.xml

```
<bean id = "authenticationEntryPoint"
    class = "ru.urvanov.virtualpets.server.auth.CustomAuthenticationEntryPoint">
</bean>
```

Листинг 12.44. SecurityConfig.java

```
@Bean
public AuthenticationEntryPoint authenticationEntryPoint() {
    return new CustomAuthenticationEntryPoint();
}
```

12.8.3. Тег *security:intercept-url* и метод *authorizeHttpRequests*

SecurityFilterChain для API клиентов JavaScript разрешает доступ к своим методам только пользователям с ролью ROLE_USER, кроме адресов /api/v1/PublicService. Адреса из /api/v1/PublicService содержат методы аутентификации, регистрации нового пользователя, проверки текущей сессии — всё то, что должно быть доступно до момента успешного входа игрока на сервер (листинги 12.45 и 12.46).

Листинг 12.45. Файл security.xml

```
<security:http pattern = "/api/**"
        authentication-manager-ref = "authenticationManager"
        entry-point-ref = "authenticationEntryPoint"
        >
    <security:intercept-url pattern = "/api/v1/PublicService/**"
        access = "permitAll" />
    <security:intercept-url pattern = "/api/**"
        access = "hasRole('USER')" />
 ...
</security:http>
```

Листинг 12.46. SecurityConfig.java

```
@Bean
public SecurityFilterChain securityFilterChainApi(
        HttpSecurity http,
        AuthenticationManager authenticationManager,
        AuthenticationEntryPoint authenticationEntryPoint
        ) throws Exception {
    return http

        .authorizeHttpRequests((authorize) ->
            authorize.requestMatchers("/api/v1/PublicService/**")
                    .permitAll()
                    .requestMatchers("/api/**").hasRole("USER")
        )

        .build();
}
```

Смысл тега security:intercept-url подробно расписан в *разд. 12.7.3*.

12.8.4. Механизм CORS

При локальной разработке игры с виртуальными питомцами, использующейся здесь в качестве примера, серверная часть запускается на порту 8080, в то время как клиентская часть запускается в nginx, доступном на порту 8081. Для того чтобы запросы из JavaScript-кода проходили с **http://localhost:8081** на **http://localhost:8080**, что фактически означает отправку запроса с одного сайта на другой, необходимо настроить CORS.

> ***CORS***
>
> CORS (Cross-Origin Resource Sharing, обмен ресурсами с запросом происхождения) — механизм, позволяющий браузеру получить доступ к выбранным ресурсам сервера на источнике, отличном от того, что сайт использует в текущий момент.

Браузеры ограничивают cross-origin запросы, инициируемые скриптами. Веб-приложения, использующие XMLHttprequest и Fetch API из JavaScript, могут запрашивать HTTP-ресурсы только с того источника, с которого они были загружены.

Настройка CORS может быть выполнена либо средствами Spring MVC, либо средствами Spring Security. Проекты virtualpets-server-springframework и virtualpets-server-springboot настраивают CORS с помощью Spring MVC.

На первом шаге необходимо включить CORS в конфигурации Spring Security с помощью тега `<security:cors />` (листинг 12.47).

Листинг 12.47. Файл security.xml

```
<security:http pattern = "/api/**"
        authentication-manager-ref = "authenticationManager"
        entry-point-ref = "authenticationEntryPoint"
        >
...
    <security:cors />
...
</security:http>
```

Аналогичная настройка для Java-конфигурации приведена в листинге 12.48.

Листинг 12.48. SecurityConfig.java

```
@Bean
public SecurityFilterChain securityFilterChainApi(
        HttpSecurity http,
        AuthenticationManager authenticationManager,
        AuthenticationEntryPoint authenticationEntryPoint
        ) throws Exception {
    return http
...
        // По умолчанию используется настройка из Spring MVC.
        .cors(Customizer.withDefaults())
...
        .build();
}
```

На втором шаге прописывается основная часть настройки в конфигурации Spring MVC (листинг 12.49).

Листинг 12.49. Файл servlet-context.xml

```
<mvc:cors>
    <mvc:mapping path = "/api/**"
        allow-credentials = "true"
        allowed-origins
            = "http://localhost:8081,
            http://localhost:8082,
            http://virtualpets.urvanov.ru,
            https://virtualpets.urvanov.ru"
        allowed-methods = "GET, POST, PUT, DELETE, OPTIONS"/>
</mvc:cors>
```

Приведенный здесь код настраивает CORS таким образом, чтобы запросы с источников **http://localhost:8081**, **http://localhost:8082** и **http://virtualpets.urvanov.ru** разрешались только для HTTP-методов GET, POST, PUT, DELETE, OPTIONS вместе с передачей информации о сессии и аутентификации.

Аналогичная настройка CORS в проекте virtualpets-server-springboot показана в листинге 12.50.

Листинг 12.50. WebConfig.java

```
@Override
public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
        .allowCredentials(true)
        .allowedOrigins("""
                http://localhost:8081, \
                http://localhost:8082, \
                http://virtualpets.urvanov.ru, \
                https://virtualpets.urvanov.ru""")
        .allowedMethods(
                HttpMethod.GET.name(),
                HttpMethod.POST.name(),
                HttpMethod.PUT.name(),
                HttpMethod.OPTIONS.name(),
                HttpMethod.DELETE.name());
}
```

Если запустить проект virtualpets-server-springframework и проект клиента на JavaScript локально и посмотреть на передаваемые HTTP-заголовки `Access-Control-Allow-*`, то картина будет следующая:

1. Клиент отправляет на сервер запрос HTTP `OPTIONS` с указанием `Origin: http://localhost:8081` и с заполненными заголовками `Access-Control-Request-Headers: content-type` и `Access-Control-Request-Method: GET`. Заголовки `Access-Control-Request-Headers` и `Access-Control-Request-Method` содержат информацию о том, что реальный запрос после `OPTIONS` планируется отправить с HTTP-методом `GET` и у запроса планируется заполненный HTTP-заголовок `Content-Type`.

2. Сервер анализирует заголовки `Access-Control-Request-*` и `Origin` запроса `OPTIONS` и принимает решение, разрешен ли такой запрос. В случае успешной проверки возвращается ответ, содержащий заголовки `Access-Control-Allow-Credentials: true`, `Access-Control-Allow-Headers: content-type`, `Access-Control-Allow-Methods: GET,POST,PUT, DELETE,OPTIONS`, `Access-Control-Allow-Origin: http://localhost:8081`, `Access-Control-Max-Age: 1800`. Заголовок `Access-Control-Max-Age` содержит время в секундах, в течение которого допускается кеширование результата запроса `OPTIONS` без отправки другого предварительного запроса на проверку разрешения.
3. Клиент, получив ответ от запроса `OPTIONS`, отправляет реальный запрос с методом HTTP GET и получает на него ответ.

12.8.5. Защита от CSRF

`SecurityFilterChain` для API клиента JavaScript не требует включения защиты от CSRF (Cross Site Request Forgery), т. к. он обрабатывает только AJAX-запросы. Благодаря механизму защиты браузеров от отправки запросов скриптами с других сайтов получение полноценного запроса с телом JSON и заполненным Content-Type: `application/json`, невозможно. Теоретически другие сайты могут формировать ссылки с GET-запросами, но это не приведет к выполнению каких-либо действий, поскольку контроллеры API проверяют HTTP-заголовок `Content-Type`. При переходе по ссылке в браузере запрос отправится с `Content-Type: text/html`, а контроллеры API принимают только запросы с `Content-Type: application/json`, поэтому пользователю отобразится страница с сообщением о неподдерживаемом типе медиа.

Обратите внимание на атрибут `consumes` аннотации `@RequestMapping` — в нем указываются допустимые `Content-Type` запроса (листинг 12.51).

Листинг 12.51. PetController.java

```
@RestController("apiPetController")
@RequestMapping(value = "api/v1/PetService",
        consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_JSON_VALUE)
public class PetController extends ControllerBase {
```

В подобных случаях, когда `Content-Type` запросов проверяется на `application/json` и контроллеры обрабатывают только AJAX-запросы, дополнительная защита от CSRF не нужна и ее можно отключить (листинги 12.52 и 12.53).

Листинг 12.52. Файл security.xml

```
<security:http pattern = "/api/**"
        authentication-manager-ref = "authenticationManager"
        entry-point-ref = "authenticationEntryPoint"
        >
...
    <security:csrf disabled = "true" />
</security:http>
```

Листинг 12.53. SecurityConfig.java

```
@Bean
public SecurityFilterChain securityFilterChainApi(
        HttpSecurity http,
        AuthenticationManager authenticationManager,
        AuthenticationEntryPoint authenticationEntryPoint
        ) throws Exception {
    return http
...
        .csrf(AbstractHttpConfigurer::disable)
        .build();
}
```

12.8.6. Листинги

В конечном результате полные коды настройки `SecurityFilterChain` для API клиента игры на JavaScript выглядят следующим образом (листинги 12.54 и 12.55).

Листинг 12.54. Файл security.xml

```
<security:http pattern = "/api/**"
        authentication-manager-ref = "authenticationManager"
        entry-point-ref = "authenticationEntryPoint"
        >
    <security:intercept-url pattern = "/api/v1/PublicService/**"
        access = "permitAll" />
    <security:intercept-url pattern = "/api/**"
        access = "hasRole('USER')" />
    <security:cors />
    <security:csrf disabled = "true" />
</security:http>
```

Листинг 12.55. SecurityConfig.java

```
@Bean
public SecurityFilterChain securityFilterChainApi(
        HttpSecurity http,
        AuthenticationManager authenticationManager,
        AuthenticationEntryPoint authenticationEntryPoint
        ) throws Exception {
    return http
        .securityMatcher("/api/**")
        .authenticationManager(authenticationManager)
        .exceptionHandling(e ->
                e.authenticationEntryPoint(authenticationEntryPoint))
        .authorizeHttpRequests((authorize) ->
            authorize.requestMatchers("/api/v1/PublicService/**")
                    .permitAll()
                    .requestMatchers("/api/**").hasRole("USER")
        )
```

```
        // По умолчанию используется настройка из Spring MVC.
        .cors(Customizer.withDefaults())
        .csrf(AbstractHttpConfigurer::disable)
        .build();
}
```

12.8.7. Аутентификация из контроллера

API сайта не использует стандартную точку входа, обрабатываемую Spring Security, как это происходило в случае `security:form-login`. Для аутентификации с помощью AJAX-запроса применяется метод контроллера `PublicController` (листинг 12.56).

Листинг 12.56. PublicController.java

```
@RequestMapping(method = RequestMethod.POST, value = "login")
public LoginResult login(
        @RequestBody @Valid LoginArg loginArg,
        HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse)
                throws ServiceException {

    // UsernamePasswordAuthenticationToken с неаутентифицированным пользователем
    Authentication authenticationRequest =
            UsernamePasswordAuthenticationToken.unauthenticated(
                    loginArg.login(), loginArg.password());

    // Попытка аутентифицировать пользователя.
    // При этом будет вызван метод сервиса UserDetailsService
    // для получения информации о пользователе из базы данных по логину
    Authentication authenticationResponse
            = authenticationManager.authenticate(
                    authenticationRequest);
    // Если оказались здесь, значит, аутентификация прошла успешно.
    // В противном случае authenticationManager#authenticate бросил бы исключение.

    // Вызов логики бизнес-слоя.
    publicService.login(loginArg);

    // Текущий контекст безопасности
    SecurityContext securityContext = SecurityContextHolder.getContext();
    // Заполнение текущего контекста аутентифицированным пользователем.
    securityContext.setAuthentication(authenticationResponse);

    // Сохранение контекста безопасности
    // в requestAttributes и в сессии.
    securityContextRepository.saveContext(
            securityContext,
            httpServletRequest,
            httpServletResponse);
```

```
    // Возвращение результата в клиент JavaScript.
    UserDetailsImpl userDetailsImpl
            = (UserDetailsImpl) authenticationResponse
                    .getPrincipal();
    return new LoginResult(
            true,
            null,
            userDetailsImpl.getUserId(),
            userDetailsImpl.getUsername(),
            userDetailsImpl.getName());
}
```

Принимаемый методом JSON-ответ отображается на класс Java (листинг 12.57).

Листинг 12.57. LoginArg.java

```
package ru.urvanov.virtualpets.server.controller.api.domain;

import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;

public record LoginArg(
        @NotNull @Size(min = 3, max = 50) String login,
        @NotNull @Size(min = 1, max = 50) String password,
        @NotNull @Size(min = 1, max = 50) String version) {
};
```

Возвращаемый методом класс приведен в листинге 12.58.

Листинг 12.58. LoginResult.java

```
package ru.urvanov.virtualpets.server.controller.api.domain;

public record LoginResult(boolean success, String message,
        Integer userId, String login, String name) {
};
```

Метод `login` слоя бизнес-логики только проверяет версию из запроса и версию сервера (листинг 12.59).

Листинг 12.59. PublicServiceImpl.java

```
@Override
public void login(LoginArg loginArg)
        throws ServiceException {
    String clientVersion = loginArg.version();
    if (!version.equals(clientVersion)) {
        throw new IncompatibleVersionException("", version,
                clientVersion);
    }
}
```

12.9. Авторизация на основе методов

При описании `SecurityFilterChain` зоны сайта и зоны API клиента на JavaScript использовалась авторизация на основе адресов, т. е. описывались шаблоны адресов с помощью тега `security:intercept-url` или с помощью метода `authorizeHttpRequests` класса `HttpSecurity` и определялись необходимые полномочия для доступа к этим адресам.

Помимо авторизации на основе адресов, Spring Security поддерживает авторизацию на основе методов, при которой для каждого метода бина с помощью специальных аннотаций прописываются правила, определяющие возможность вызова метода пользователем.

По умолчанию авторизация на основе методов отключена.

Проект virtualpets-server-springframework включает авторизацию на основе методов с помощью XML-конфигурации (листинг 12.60).

Листинг 12.60. Файл security.xml

```
<security:method-security />
```

Проект virtualpets-server-springboot включает авторизацию на основе методов с помощью Java-аннотации `@EnableMethodSecurity` (листинг 12.61).

Листинг 12.61. SecurityConfig.java

```
@Configuration
@EnableMethodSecurity
public class SecurityConfig {
```

После включения авторизации на основе методов начинают работать аннотации:

- `@PreAuthorize` — проверка полномочий для запуска метода перед его выполнением. Доступ к параметрам метода осуществляется с помощью `#myParamName`, при этом сам параметр метода должен быть аннотирован;
- `@PostAuthorize` — проверка полномочий для получения значения результата выполнения метода. С помощью этой аннотации проверяется результат выполнения метода и осуществляется проверка на возможность получения данных этого результата аутентифицированным пользователем. Внутри SpEL-выражения доступ к результату выполнения метода осуществляется с помощью переменной `returnObject`;
- `@PreFilter` — фильтрация коллекций, передаваемых в метод. Позволяет проверить передаваемую в метод коллекцию объектов и оставить в ней только те объекты, которые разрешено использовать текущему аутентифицированному пользователю. Внутри SpEL доступ к фильтруемой коллекции осуществляется помощью переменной `filterObject`;
- `@PostFilter` — фильтрация коллекций, возвращаемых из метода. Позволяет проверить возвращаемую методом коллекцию объектов и оставить в ней только те

значения, к которым есть доступ у текущего аутентифицированного пользователя. Внутри SpEL доступ к фильтруемой коллекции осуществляется помощью переменной `filterObject`.

Для формирования правил доступа аннотаций из приведенного здесь списка используется выражение SpEL (Spring Expression Language), в котором доступны объекты `authentication` и `principal`, а также есть доступ к параметрам вызываемого метода, а в случае `@PostAuthorize` и `@PostFilter` — еще и доступ к возвращаемому объекту.

Проекты virtualpets-server-springframework и virtualpets-server-springboot активно используют аннотацию `@PreAuthorize` на методах сервисов. Например, сервис `HiddenObjectsServiceImpl` применяет аннотацию `@PreAuthorize` на всем классе (листинг 12.62).

Листинг 12.62. HiddenObjectsServiceImpl.java

```
@Service
@PreAuthorize("hasRole('USER')")
public class HiddenObjectsServiceImpl implements HiddenObjectsApiService {
```

Вызывать методы бина класса `HiddenObjectsServiceImpl` смогут только аутентифицированные пользователи с ролью `ROLE_USER`.

Аналогичная аннотация используется на уровне методов сервиса `PetServiceImpl` листинг 12.63).

Листинг 12.63. PetServiceImpl.java

```
@Override
@PreAuthorize("hasRole('USER')")
public GetPetFoodsResult getPetFoods(UserPetDetails userPetDetails)
        throws ServiceException {
```

Несколько более сложный вариант `@PreAuthorize`, использующий параметры вызываемого метода внутри своего SpEL, находится в сервисе `UserServiceImpl` (листинг 12.64).

Листинг 12.64. UserServiceImpl.java

```
...
import org.springframework.security.core.parameters.P;
...
@Override
@PreAuthorize("""
        hasRole('ADMIN') \
        && (#userAccessRights.id ne principal.userId)
        """)
@Transactional(rollbackFor = ServiceException.class)
public UserAccessRights saveUserAccessRights(
        @P("userAccessRights") UserAccessRights userAccessRights)
        throws UserNotFoundException {
```

SpEL проверяет здесь наличие роли ROLE_ADMIN у аутентифицированного пользователя. Дополнительно SpEL убеждается, что поле id в параметре userAccessRights метода не равно первичному ключу текущего аутентифицированного пользователя — т. е. что пользователь не пытается изменить права доступа у самого себя.

12.10. Библиотека Spring Security JSP Taglib

Spring Security включает библиотеку тегов Jakarta Pages, находящуюся в артефакте spring-security-taglibs.

Подключение соответствующей зависимости для проекта на Spring Framework показано в листинге 12.65.

Листинг 12.65. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>
    <version>${org.springframework-security-version}</version>
</dependency>
```

Здесь ${org.springframework-security-version} — это используемая версия Spring Security, описанная в элементе properties файла pom.xml.

Библиотека тегов Spring Security JSP Taglib содержит теги в пространстве имен http://www.springframework.org/tags. Для этого пространства используется префикс sec:

```
xmlns:sec = "http://www.springframework.org/security/tags"
```

В библиотеке тегов Spring Security JSP Taglib содержатся теги для доступа текущего аутентифицированного пользователя UserDetails к полям, теги для проверки доступа текущего пользователя к определенному адресу, а также некоторые другие теги, которые применяются гораздо реже.

Тег sec:authorize служит для проверки полномочий пользователя. В самом простом случае он задействуется для ограничения отображения своего содержимого только тем пользователям, которые имеют доступ к определенному адресу.

Пример использования тега sec:authorize из файла src/main/webapp/WEB-INF/views/user/profile.jspx проекта virtualpets-server-springframework приведен в листинге 12.66.

Листинг 12.66. Файл profile.jspx

```
<sec:authorize url = "/site/admin">
    <div>
        <spring:url value = "/site/admin"
            var = "admin_panel_url" />
        <a href = "${admin_panel_url}">${admin_panel_var}</a>
    </div>
</sec:authorize>
```

Здесь ссылка на панель администратора по адресу `/site/admin` отображается только тем пользователям, которые имеют доступ к адресу `/site/admin`. Согласно `SecurityFilterChain` сайта — это пользователи с ролью `ROLE_ADMIN`.

Обычные пользователи с ролью `ROLE_USER` и без роли `ROLE_ADMIN` не увидят этой ссылки. Впрочем, они в любом случае не смогли бы по ней перейти из-за недостатка полномочий.

Попробуйте в качестве эксперимента запустить virtualpets-server-springframework локально, затем зайдите под пользователями `admin` и `user` поочередно (пароль по умолчанию: `123`), используя форму входа **http://localhost:8080/virtualpets-server-springframework/site/login**. При заходе под пользователем `admin` вы увидите ссылку на страницу **Панель администрирования** (рис. 12.4), а при заходе под пользователем `user` ссылки на панель администрирования на странице не будет (рис. 12.5).

Рис. 12.4. Профиль пользователя со ссылкой на панель администратора при заходе под пользователем `admin`

Рис. 12.5. Профиль пользователя без ссылки на панель администратора при заходе под пользователем `user`

В том же файле src/main/webapp/WEB-INF/views/user/profile.jspx находятся примеры использования тега `sec:authentication` для отображения первичного ключа, логина и полного имени текущего пользователя (листинг 12.67). Объект `principal` здесь — это реализация `UserDetails`, используемая в проекте.

Листинг 12.67. Файл profile.jspx

```
<div>
    ID: <sec:authentication property = "principal.userId" />
</div>
<div>
    ${username_var}: <sec:authentication property = "principal.username" />
</div>
<div>
    ${name_var}: <sec:authentication property = "principal.name" />
</div>
```

12.11. Интеграция с Thymeleaf

Проект virtualpets-server-springboot полностью воспроизводит область сайта, которую проект virtualpets-server-springframework генерирует с помощью Jakarta Pages, но в случае virtualpets-server-springboot используется уже Thymeleaf, который имеет аналогичную интеграцию со Spring Security.

Перед использованием интеграции Thymeleaf и Spring Security необходимо подключить дополнительную зависимость `thymeleaf-extras-springsecurity6` (листинг 12.68).

Листинг 12.68. Файл pom.xml

```
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity6</artifactId>
</dependency>
```

Интеграция Thymeleaf и Spring Security похожа на интеграцию Jakarta Pages и Spring Security. В обоих случаях появляется возможность использовать дополнительные теги, и в обоих случаях эти теги делают примерно то же самое.

Шаблон src/main/resources/templates/user/profile.html содержит примеры использования артефакта `thymeleaf-extras-springsecurity6`. В первом теге HTML-шаблона пространство имен `http://www.thymeleaf.org/extras/spring-security` ассоциируется с префиксом `sec` — это необходимо, чтобы подсветка синтаксиса в IDE не подсвечивала новый префикс как ошибку.

Ассоциация пространства имен с префиксом `sec` в файле profile.html показана в листинге 12.69.

Листинг 12.69. Файл profile.html

```
<!DOCTYPE html>
<html xmlns:th = "http://www.thymeleaf.org"
    xmlns:sec = "http://www.thymeleaf.org/extras/spring-security"
```

Пример использования тега `sec:authorize-url` для отображения содержимого элемента только тем пользователям, которые имеют полномочия для доступа к указанному адресу, приведен в листинге 12.70.

Листинг 12.70. Файл profile.html

```
<div sec:authorize-url = "@{/site/admin}">
    <a th:href = "@{/site/admin}"
    th:text =
    "#{virtualpets-server-springboot.site.admin_panel}">
    </a>
</div>
```

Пример использования тега `sec:authentication` для отображения первичного ключа текущего аутентифицированного пользователя, логина и полного имени приведен в листинге 12.71.

Листинг 12.71. Файл profile.html

```
<div>
    ID: <span sec:authentication  = "principal.userId"></span>
</div>
```

```
<div>
    <th:block th:text =
    "#{virtualpets-server-springboot.site.username}">
    </th:block>: <span sec:authentication = "principal.username"></span>
</div>
<div>
    <th:block th:text =
    "#{virtualpets-server-springboot.site.name}">
    </th:block>: <span sec:authentication = "principal.name">
    </span>
</div>
```

Здесь объект `principal`, к полям которого происходит обращение в `sec:authentication`, — это текущий аутентифицированный пользователь, сохраненный в `SecurityContextHolder`, т. е. реализация `UserDetails`, используемая в проекте.

12.12. Резюме

В этой главе рассмотрена архитектура Spring Security, ее интеграция с контекстом Spring и с фильтрами Jakarta EE. Spring Security позволяет реализовать процесс аутентификации и авторизации. Интеграция с Jakarta Pages и Thymeleaf обеспечивает отображение на генерируемых страницах только доступных пользователю элементов, а также отображение информации о текущем аутентифицированном пользователе.

Spring Security тесно взаимодействует со Spring Web MVC и обеспечивает защиту от атак Cross Site Request Forgery.

ГЛАВА 13

Документирование REST-сервисов

13.1. Введение

Каким бы простым ни был API сервиса, какие бы протоколы и стандарты он ни использовал, этого будет недостаточно для понимания способов взаимодействия с сервисом. Чтобы разработчики клиентских приложений могли получить представление о том, как использовать сервис, необходимо иметь дополнительное текстовое описание методов сервиса, его параметров, логики работы с ними, возможных возвращаемых значений и ошибок.

В самом простейшем случае описание может быть выполнено в виде HTML-страницы, документа Word, PDF или ODF. Недостаток подобного подхода в том, что документация сервиса при этом живет отдельно от самого кода сервиса, может существенно от него отличаться, отставать, терять актуальность и т. д.

Помочь нам в решении этой проблемы способен OpenAPI — машиночитаемый язык для описания интерфейсов веб-сервисов. Можно его рассматривать как некий аналог WSDL из мира SOAP-сервисов.

Стартер `springdoc-openapi-starter-webmvc-ui` позволяет генерировать файлы спецификации OpenAPI, а также веб-интерфейс для просмотра этой спецификации в виде, удобном для человека.

Достаточно долгие годы для генерации спецификации OpenAPI использовался проект SpringFox, но последние изменения в его репозитории на GitHub были произведены еще 14 октября 2020 года. Прошло более четырех лет, а значит, проект заброшен и не актуален.

13.2. Подключение зависимостей

На момент подготовки книги для генерации спецификаций OpenAPI в проектах на Spring Boot используется springdoc-openapi. Генерацию спецификации OpenAPI в этой книге мы рассмотрим на примере проекта virtualpets-server-springboot.

Начнем с подключения зависимости (листинг 13.1).

Листинг 13.1. Файл pom.xml

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.6.0</version>
</dependency>
```

Подключения стартера уже достаточно — стартер `springdoc-openapi-starter-webmvc-ui` выполнит всю необходимую настройку.

13.3. Просмотр сгенерированной документации

После запуска проекта virtualpets-server-springboot сгенерированная спецификация OpenAPI (рис. 13.1) доступна по адресу: **http://localhost:8080/v3/api-docs**.

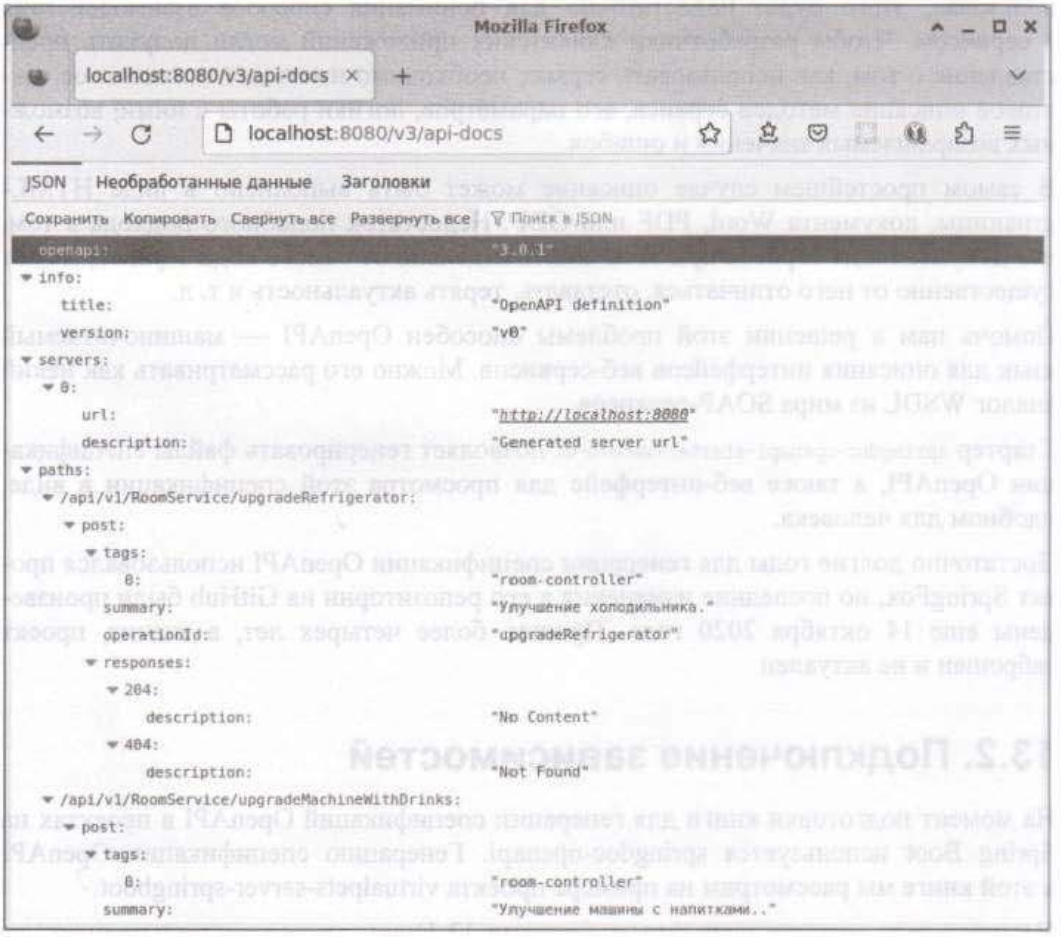


Рис. 13.1. Сгенерированная спецификация OpenAPI

Веб-интерфейс Swagger UI, предоставляющий возможность просмотреть документацию на API в виде, понятном человеку (рис. 13.2), доступен по адресу: **http://localhost:8080/swagger-ui.html**.

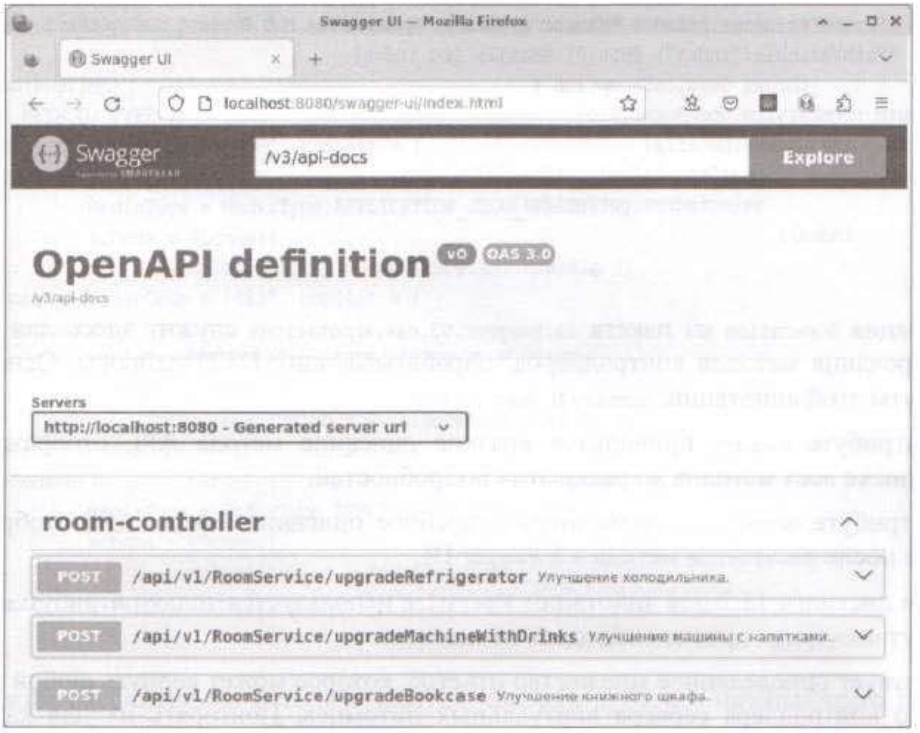


Рис. 13.2. Интерфейс Swagger UI

13.4. Документирование API

При генерации спецификации в Swagger UI используются:

- аннотации Spring MVC;
- аннотации Jakarta Validation;
- могут также использоваться комментарии JavaDoc, но для этого потребуется дополнительная настройка;
- аннотации Swagger.

В качестве примера рассмотрим метод открытия коробки лутбокса начинающего пользователя (листинг 13.2).

Листинг 13.2. RoomController.java

```
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.Parameter;
```

```
@PostMapping(value = "openBoxNewbie/{index}/")
@Operation(summary = "Открытие лутбокса.")
@SwaggerCommonResponses
public OpenBoxNewbieResult openBoxNewbie(
        @AuthenticationPrincipal UserDetailsImpl userDetailsImpl,
        @Parameter(description = "Индекс лутбокса. Начинается с 0.")
        @PathVariable("index") @Min(0) @Max(1) int index)
                throws ServiceException {
    return roomService.openBoxNewbie(
            new UserPetDetails(
                    userDetailsImpl.getUserId(),
                    selectedPet.getPetId()),
            index);
}
```

Аннотация `@Operation` из пакета `io.swagger.v3.oas.annotations` служит здесь для документирования методов контроллеров, обрабатывающих HTTP-запросы. Основные атрибуты этой аннотации: `summary` и `description`:

- в атрибуте `summary` приводится краткое описание метода API, отображаемое в списке всех методов до раскрытия подробностей;
- в атрибуте `description` приводится подробное описание метода API, отображаемое после раскрытия метода в Swagger UI.

В коде листинга 13.2 для аннотации `@Operation` используется только атрибут `summary`, атрибут `description` применения здесь не находит.

Существует определенное множество ответов, которое может вернуть любой метод любого контроллера сервера виртуальных питомцев. Повторять их для каждого метода — это не самая лучшая идея, поскольку это приведет к загромождению кода и повысит вероятность ошибок. Аннотация `@SwaggerCommonResponses` (листинг 13.3) — это специальная аннотация проекта virtualpets-server-springboot, в которой собраны все стандартные ответы API, чтобы не повторять их для каждой операции. Без аннотации `@SwaggerCommonResponses` все аннотации `@ApiResponse` приходилось бы повторять для каждого метода контроллера, обрабатывающего запросы API клиента на JavaScript.

Листинг 13.3. SwaggerCommonResponses.java

```
package ru.urvanov.virtualpets.server.controller.api.swagger;

import java.lang.annotation.Target;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.TYPE;

import java.lang.annotation.Retention;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import org.springframework.http.MediaType;
import org.springframework.http.ProblemDetail;
```

```
import io.swagger.v3.oas.annotations.media.Content;
import io.swagger.v3.oas.annotations.media.Schema;
import io.swagger.v3.oas.annotations.responses.ApiResponse;

/**
 * Описание стандартных ответов API, которые может вернуть любой метод.
 */
@Retention(RUNTIME)
@Target({ METHOD, TYPE })
@ApiResponse(responseCode = "400", content = {
        @Content(
                mediaType = MediaType.APPLICATION_JSON_VALUE,
                schema = @Schema(
                        implementation = ProblemDetail.class)) })
@ApiResponse(responseCode = "403", content = {
        @Content(
                mediaType = MediaType.APPLICATION_JSON_VALUE,
                schema = @Schema(
                        implementation = ProblemDetail.class)) })
@ApiResponse(responseCode = "404", content = {
        @Content(
                mediaType = MediaType.APPLICATION_JSON_VALUE,
                schema = @Schema(
                        implementation = ProblemDetail.class)) })
public @interface SwaggerCommonResponses {
}
```

Продолжим рассмотрение метода открытия коробки лутбокса начинающего пользователя (см. листинг 13.2). Аннотация `@Parameter` из того же пакета `io.swagger.v3.oas.annotations` служит для документирования параметров методов. С помощью этой аннотации здесь описывается параметр `index` запроса. Как можно видеть, с использованием `@Parameter` документируются не только параметры пути, но и параметры запроса, передаваемые после знака вопроса в адресе.

Метод `openBoxNewbie` возвращает сущность `OpenBoxNewbieResult` (листинг 13.4). Поля этой сущности помечены аннотацией `@Schema`, в атрибуте `description` которой находится их описание.

Листинг 13.4. OpenBoxNewbieResult.java

```
package ru.urvanov.virtualpets.server.controller.api.domain;

import java.util.Map;

import io.swagger.v3.oas.annotations.media.Schema;
import ru.urvanov.virtualpets.server.dao.domain.BuildingMaterialId;

@Schema(description = "Добыча из лутбокса")
public record OpenBoxNewbieResult(
```

```
    @Schema(
        description = "Индекс лутбокса. Начинается с 0.",
        example = "0")
    int index,

    @Schema(
        description = "Полученные ресурсы",
        example = """
            {"TIMBER": 2, "STONE": 3}
            """)
    Map<BuildingMaterialId, Integer> buildingMaterials) {
};
```

Представление метода `openBoxNewbie` в Swagger UI показано на рис. 13.3.

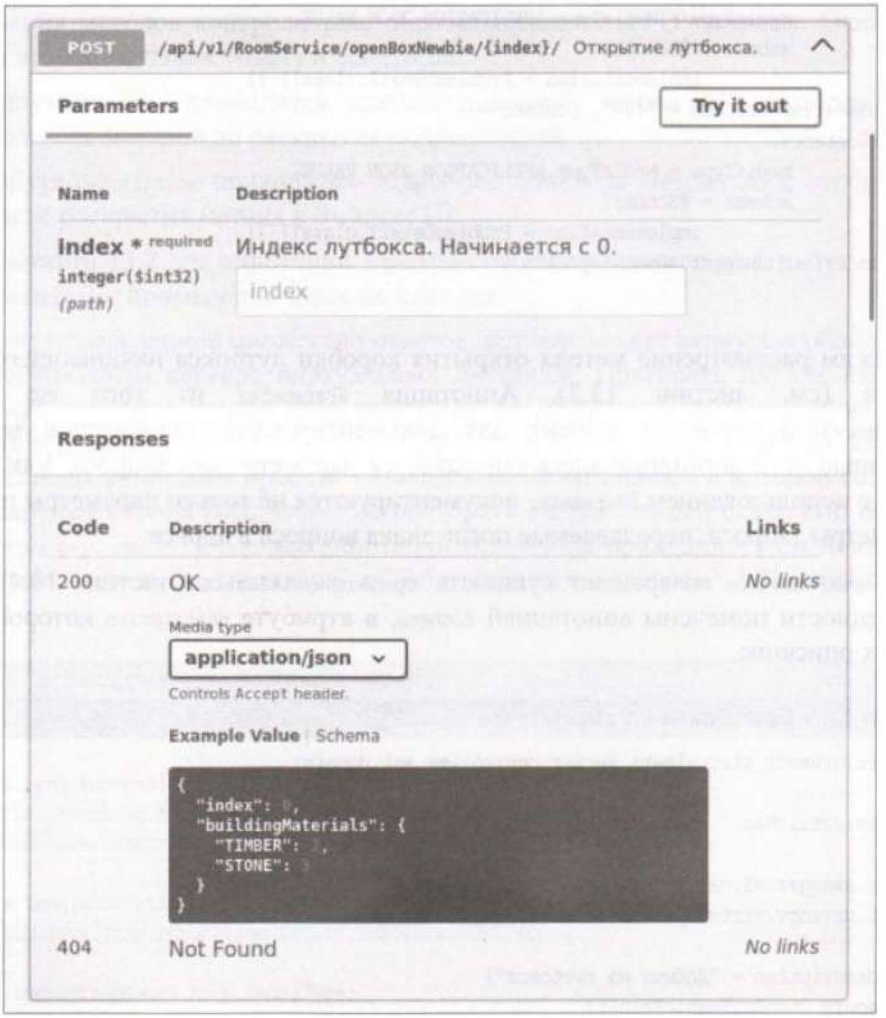


Рис. 13.3. Сгенерированная страница Swagger UI с описанием метода открытия лутбокса

Обратите внимание, что над примером ответа, приведенном на рис. 13.3, находятся надписи **Example Value** и **Schema**. Они работают по логике вкладок — т. е. позволяют переключаться между отображением примера и иерархического описания. Если щелкнуть на опции **Shema**, то будет отображено описание из атрибутов `description` аннотаций `@Schema` из класса `OpenBoxNewbieResult` (рис. 13.4).

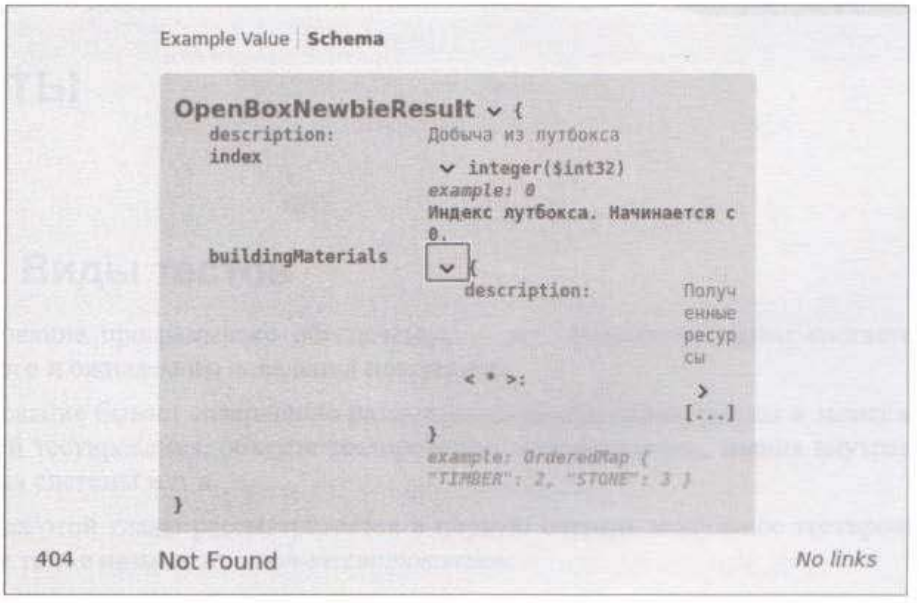


Рис. 13.4. Описание схемы `OpenBoxNewbieResult` в Swagger UI

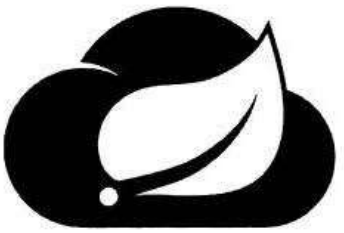
13.5. Резюме

Проект springdoc-openapi позволяет генерировать документацию OpenAPI для проектов на Spring.

OpenAPI и Swagger — это не единственные инструменты документирования API веб-сервисов. Существуют и другие спецификации и языки моделирования со своими нишами и особенностями:

- RAML (RESTful API Modeling Language) — основанный на YAML язык описания статических API;
- API Blueprint — высокоуровневый язык описания веб-API;
- WADL (Web Application Description Language) — XML-описания веб-сервисов, основанных на HTTP.

На сегодняшний день при разработке сервисов на Java и Spring в основном используется OpenAPI и Swagger, а для интеграции со Spring — стартер `springdoc-openapi-starter-webmvc-ui`, как это сделано в проекте virtualpets-server-springboot.



ГЛАВА 14

Тесты

14.1. Виды тестов

Тестирование программного обеспечения — это процесс проверки соответствия реального и ожидаемого поведения программы.

Тестирование бывает совершенно разных видов и классифицируется в зависимости от целей тестирования, объекта тестирования, автоматизации, знания внутреннего строения системы и т. д.

В рамках этой главы рассматривается в первую очередь модульное тестирование, которое также называют *юнит-тестированием*.

> ***Модульное тестирование***
>
> Модульное тестирование — это разновидность тестирования, при котором осуществляется проверка работоспособности отдельных модулей исходного кода программы, а также набора из одного или более программных модулей вместе с соответствующими данными.

14.2. Фреймворк JUnit

JUnit — фреймворк модульного тестирования программ на Java. И проект virtualpets-server-springframework, и проект virtualpets-server-springboot используют JUnit в своих тестах.

14.2.1. Подключение зависимостей

Для проекта на Spring Framework (virtualpets-server-springframework) необходимо в файле pom.xml подключить артефакт `junit-jupiter` (листинг 14.1).

Листинг 14.1. Файл pom.xml

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
```

```
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
```

Здесь `${junit.version}` — это версия JUnit, объявленная в секции `properties` файла pom.xml (листинг 14.2).

Листинг 14.2. Файл pom.xml

```
<properties>
        <junit.version>5.11.0</junit.version>
</properties>
```

Для проекта на Spring Boot (virtualpets-server-springboot) необходимо в файле pom.xml подключить стартер `spring-boot-starter-test` (листинг 14.3).

Листинг 14.3. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

14.2.2. Простейший тест

Сами тесты работают и выглядят абсолютно идентично как в проектах на Spring Framework, так и в проектах на Spring Boot.

В качестве начального примера хорошо подходят небольшие классы с одним-двумя методами, не имеющие состояния. В проектах virtualpets-server-springframework и virtualpets-server-springboot в пакете `ru.urvanov.virtualpets.server.convserv` как раз содержатся несколько классов, в которых есть только один метод конвертации. Рассмотрим для примера класс `BookToApiConverter` (листинг 14.4).

Листинг 14.4. BookToApiConverter.java

```
package ru.urvanov.virtualpets.server.convserv;

import org.springframework.core.convert.converter.Converter;

import ru.urvanov.virtualpets.server.dao.domain.Book;

/**
 * На основе экземпляра класса Book предметной области создает
 * экземпляр Book из API для JavaScript-клиента.
 */
public class BookToApiConverter implements Converter<Book,
        ru.urvanov.virtualpets.server.controller.api.domain.Book> {
```

```
    /**
     * @param source Экземпляр {@link Book} предметной области.
     * @return Экземпляр {@link
     * ru.urvanov.virtualpets.server.controller.api.domain.Book} API
     */
    @Override
    public ru.urvanov.virtualpets.server.controller.api.domain.Book
            convert(
                    Book source) {
        return new ru.urvanov.virtualpets.server.controller.api.domain
                .Book(
                        source.getId(),
                        source.getBookcaseLevel(),
                        source.getBookcaseOrder());
    }
}
```

Класс `BookToApiConverter` имеет только один метод — `convert`, возвращающий экземпляр Book API на основе экземпляра `Book` предметной области. Тест метода `BookToApiConverter#convert` должен соответственно вызывать метод `convert`, а затем сверять возвращаемое им значение с эталонным.

Классы тестов располагаются в каталоге **src/test/java**. В простейшем случае тесты JUnit представляют собой обычные классы, один или несколько методов которых помечены аннотацией `@Test` из пакета `org.junit.jupiter.api`. Пример JUnit-теста `BookToApiConverterTest`, тестирующего класс `BookToApiConverter`, приведен в листинге 14.5.

Листинг 14.5. BookToApiConverterJUnitTest.java

```
package ru.urvanov.virtualpets.server.convserv;

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

import ru.urvanov.virtualpets.server.dao.domain.Book;


/**
 * Тесты для {@link BookToApiConverter}.
 */
class BookToApiConverterJUnitTest {

    private static final float HIDDEN_OBJECTS_GAME_DROP_RATE = 0;
    private static final int BOOKCASE_ORDER = 0;
    private static final int BOOKCASE_LEVEL = 1;
    private static final String BOOK_ID = "DESTINY_BOOK";
```

```
    /**
     * Простейший пример теста JUnit.
     */
    @Test
    void convert() {
        // Экземпляр тестируемого класса
        BookToApiConverter converter = new BookToApiConverter();

        // Подготовка тестовых данных
        Book source = new Book(
                BOOK_ID,
                BOOKCASE_LEVEL,
                BOOKCASE_ORDER,
                HIDDEN_OBJECTS_GAME_DROP_RATE);

        // Подготовка ожидаемого результата
        var expected = new ru.urvanov.virtualpets.server.controller.api
                .domain.Book(
                        BOOK_ID,
                        BOOKCASE_LEVEL,
                        BOOKCASE_ORDER);

        // Вызов тестируемого метода
        var actual = converter.convert(source);

        // Проверка результата
        assertEquals(expected,  actual);
    }
}
```

Каждый тест должен содержать хотя бы один вызов `assert*` из класса `org.junit.jupiter.api.Assertions`. В коде листинга 14.5 для сравнения ожидаемого экземпляра объекта (`expected`) и реального экземпляра объекта (`actual`) используется метод `Assertions#assertEquals`. Если `expected` и `actual` окажутся не равны, то тест завершится неудачей. Сравнение объектов производится с помощью метода `Object#equals`.

Другие полезные методы из класса `Assertions`:

- `assertNotEquals` — объекты/ значения не равны;
- `assertNull` — объект ссылается на `null`;
- `assertFalse` — равно `false`;
- `assertTrue` — равно `true`;
- `assertArrayEquals` — поэлементное сравнение массивов;
- `assertDoesNotThrow` — лямбда-выражение НЕ бросает исключение;
- `assertInstanceOf` — является экземпляром объекта.

Тесты не обязательнодолжны быть public

Тесты могут и не иметь модификатора доступа. Класс теста и его методы с аннотацией `@Test` не нуждаются ни в каких модификаторах доступа — ни в `public`, ни в `private`, ни в `protected`.

14.2.3. Запуск тестов

При сборке Maven тесты запускаются на фазе `test`:

```
mvn clean test
```

Тесты также проходят при запуске любой фазы, идущей после `test`, — например: `package`, `verify`, `install`, `deploy`.

Интеграционные тесты

Жизненный цикл сборки Maven содержит также фазу запуска интеграционных тестов. Если вы хорошо знакомы с Maven и фазами сборки — это хорошо. Если же ваше представление о Maven лишь поверхностное, то пока лучше слишком в это не погружаться.

Достаточно часто — чтобы убедиться в правильности принятого подхода — приходится запускать тесты из IDE, а не из сборки Maven. При этом обычно требуется запустить какой-нибудь один специфичный тест, а не все сразу. Все популярные IDE имеют интеграцию с JUnit для запуска тестов и отображения результатов их выполнения.

Для запуска тестов JUnit в IntelliJ IDEA необходимо щелкнуть мышью на зеленом значке, расположенном в области с номерами строк кода на том же уровне, что и заголовок метода с тестом (поз. 1 на рис. 14.1), а затем в появившемся меню (поз. 2) выбрать пункт **Run *<имя метода>***. Например, для запуска теста `BookToApiConverterJUnitTest#convert` щелкните на указанном стрелкой (поз. 1) зеленом значке левее заголовка метода в области с номерами строк, а затем выберите пункт **Run 'convert()'**, как показано на рис. 14.1.

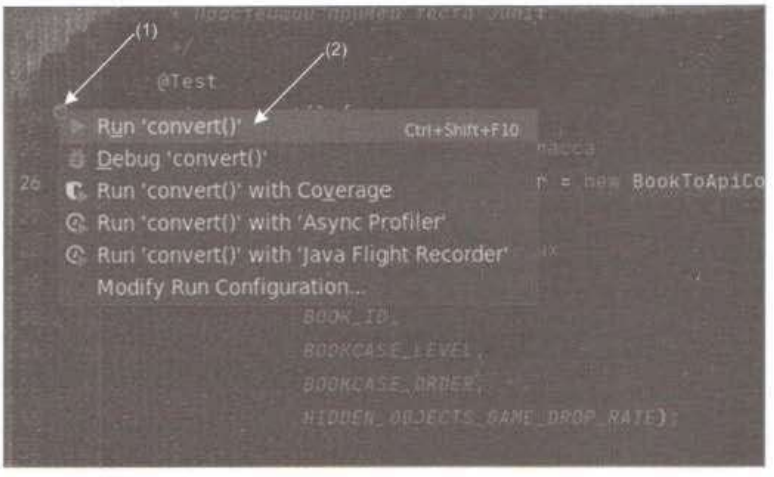


Рис. 14.1. Запуск теста JUnit из IntelliJ IDEA

После выполнения теста отобразится окно с результатом. При успешном прохождения теста это будет одна строка **Test Results** с зеленой галочкой (рис. 14.2). Если какие-либо из запущенных тестов закончились неудачей, то для этого провалившегося теста в окне результатов отобразится строка с перечеркнутым красным кружком, при щелчке на которой будет выведена детальная информация о произошедшем.

Рис. 14.2. Результаты теста `BookToApiConverterJUnitTest#convert` **в IntelliJ IDEA**

IntelliJ IDEA позволяет запускать не только одиночные тесты. Если щелкнуть правой кнопкой мыши в окне **Projects** на всем проекте, либо на каталоге с классами тестов, либо на классе с тестами, то в контекстном меню отобразится аналогичный пункт запуска всех тестов из выбранного элемента.

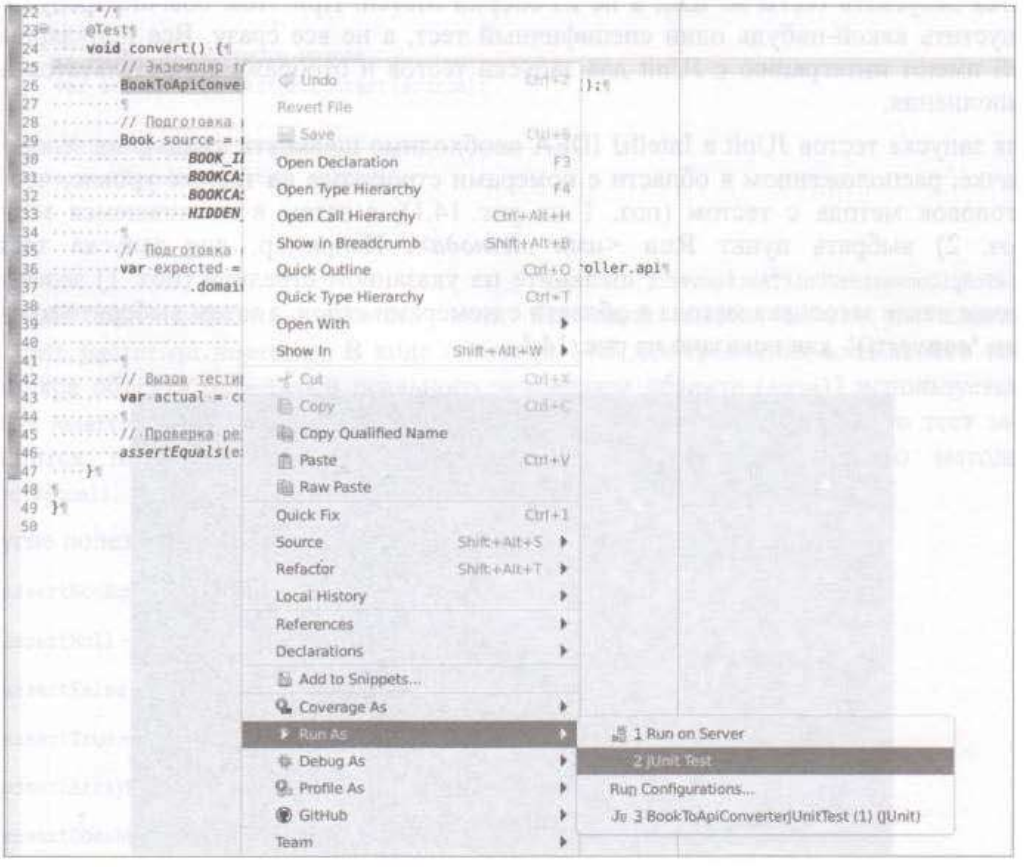


Рис. 14.3. Запуск теста JUnit в Eclipse

Запуск тестов JUnit в Eclipse выглядит несколько иначе. Там необходимо щелкнуть мышью на методе с тестом и в открывшемся контекстном меню выбрать пункт **Run As | JUnit Test** (рис. 14.3). После выполнения теста отобразится окно с результатом (рис. 15.4).

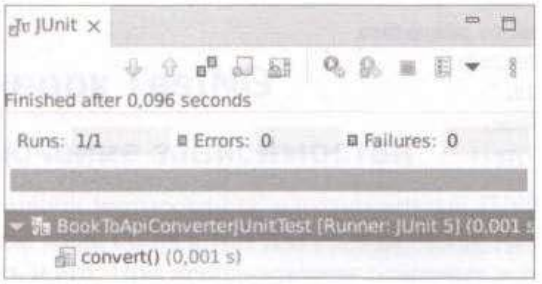


Рис. 14.4. Результаты теста `BookToApiConverterJUnitTest#convert` **в Eclipse**

14.2.4. Параметризованные тесты

Зачастую один и тот же метод необходимо проверить на разных тестовых данных. Как вариант, в этом случае можно создать несколько тестов, которые подготавливают исходные данные и вызывают тестируемый метод с ними. В большинстве случаев такой подход приведет к разрастанию кода тестов и дублированию его фрагментов, что в дальнейшем усложнит добавление новых тестовых данных.

Для упрощения создания тестов с разными тестовыми данными используются параметризованные тесты. В JUnit для этого служит аннотация `@ParameterizedTest` из пакета `org.junit.jupiter.params`, которая ставится на параметризованном методе. Существуют разные способы предоставить тестовые данные в такой метод. Наиболее гибкий вариант — это применение аннотации `@MethodSource` из пакета `org.junit.jupiter.params.provider`, в единственном атрибуте `value` которого указывается наименование метода, предоставляющего тестовые данные.

Простейший пример использования аннотаций `@ParameterizedTest` и `@MethodSource` приведен в листинге 14.6.

Листинг 14.6. PetToApiConverterJUnitTest.java

```
/**
 * Простейший пример параметризованного теста JUnit.
 * @param petId Первичный ключ питомца.
 * @param petName Имя питомца.
 * @param petType Тип питомца.
 */
@ParameterizedTest
@MethodSource("convertMethodSource")
void convert(Integer petId, String petName, PetType petType) {
    // Экземпляр тестируемого класса
    PetToApiConverter converter = new PetToApiConverter();
```

```
    // Подготовка исходных данных
    Pet source = new Pet();
    source.setId(petId);
    source.setName(petName);
    source.setPetType(petType);

    // Подготовка ожидаемого результата
    var expected = new PetInfo(
                    petId,
                    petName,
                    petType);

    // Вызов тестируемого метода
    var actual = converter.convert(source);

    // Проверка результата
    assertEquals(expected,  actual);
}
```

Метод `convert` здесь принимает параметры `petId`, `petName` и `petType`, которые предоставляет ему метод `convertMethodSource`, указанный в `@MethodSource`.

Статический метод `convertMethodSource`, имя которого указано в аннотации `@MethodSource`, должен быть объявлен в том же классе, что и тест (листинг 14.7).

Листинг 14.7. PetToApiConverterJUnitTest.java

```
private static final Integer PET1_ID = 1;
private static final String PET1_NAME = "Vasya";
private static final PetType PET1_TYPE = PetType.CAT;

private static final Integer PET2_ID = 2;
private static final String PET2_NAME = "Котик";
private static final PetType PET2_TYPE = PetType.CAT;

/**
 * Подготовка данных параметризованного теста.
 * @return Параметры теста.
 */
static Stream<Arguments> convertMethodSource() {
    return Stream.of(
            arguments(PET1_ID, PET1_NAME, PET1_TYPE),
            arguments(PET2_ID, PET2_NAME, PET2_TYPE));
}
```

Здесь метод `convertMethodSource` формирует исходные данные для двух запусков теста `PetToApiConverterJUnitTest#convert`:

1. Запуск с питомцем `Vasya` с `id = 1`.
2. Запуск с питомцем `Котик` с `id = 2`.

Параметризованные тесты стоит использовать далеко не всегда. Если тестируемый метод линеен, то дополнительные тестовые данные не принесут никакой пользы. Параметризованные тесты лучше всего раскрывают свой потенциал, когда необходимо проверять какие-либо ответвления, граничные условия, отрицательные и положительные значения и т. д.

14.3. Фреймворк TestNG

14.3.1. Подключение зависимостей

TestNG — это фреймворк тестирования, вдохновленный JUnit и NUnit. Несмотря на то что в современном мире стандартом для Java является JUnit, в некоторых проектах можно встретить и его. При использовании основных возможностей фреймворков TestNG и JUnit разницу между ними заметить практически невозможно.

Подключается TestNG одинаково и в случае проекта на Spring Framework, и в случае проекта на Spring Boot (листинг 14.8).

Листинг 14.8. Файл pom.xml

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>${testng.version}</version>
    <scope>test</scope>
</dependency>
```

Здесь `testng.version` — это версия TestNG, объявленная в `properties` того же файла pom.xml (листинг 14.9).

Листинг 14.9. Файл pom.xml

```
<properties>
    <testng.version>7.9.0</testng.version>
</properties>
```

14.3.2. Простейший тест

Тесты с помощью TestNG пишутся аналогично тестам на JUnit. В простейшем случае разница будет лишь в пакетах с аннотациями и порядке параметров `expected` и `actual` в методах `assert*`. Например, аннотация `@Test` в JUnit находится в пакете `org.junit.jupiter.api`, а в TestNG аналогичная аннотация находится в пакете `org.testng`. А методы `assert*` в JUnit принимают в качестве первого параметра ожидаемое значение (`expected`), а в качестве второго параметра принимают реальное значение (`actual`). Методы `assert*` TestNG, наоборот, принимают в качестве первого параметра `actual`, а в качестве второго — `expected`.

Пример простейшего теста на TestNG приведен в листинге 14.10.

Листинг 14.10. BookToApiConverterTestNgTest.java

```
package ru.urvanov.virtualpets.server.convserv;

import static org.testng.Assert.assertEquals;

import org.testng.annotations.Test;

import ru.urvanov.virtualpets.server.dao.domain.Book;

/**
 * Тесты для {@link BookToApiConverter}.
 */
public class BookToApiConverterTestNgTest {

    private static final float HIDDEN_OBJECTS_GAME_DROP_RATE = 0;
    private static final int BOOKCASE_ORDER = 0;
    private static final int BOOKCASE_LEVEL = 1;
    private static final String BOOK_ID = "DESTINY_BOOK";

    /**
     * Простейший пример теста TestNG.
     */
    @Test
    void convert() {
        // Экземпляр тестируемого класса
        BookToApiConverter converter = new BookToApiConverter();

        // Подготовка исходных данных
        Book source = new Book(
                BOOK_ID,
                BOOKCASE_LEVEL,
                BOOKCASE_ORDER,
                HIDDEN_OBJECTS_GAME_DROP_RATE);

        // Подготовка ожидаемого результата
        var expected = new ru.urvanov.virtualpets.server.controller.api
                .domain.Book(
                        BOOK_ID,
                        BOOKCASE_LEVEL,
                        BOOKCASE_ORDER);

        // Вызов тестируемого метода
        var actual = converter.convert(source);

        // Проверка результата
        assertEquals(actual, expected);
    }
}
```

14.3.3. Запуск тестов

Тесты TestNG в IntelliJ IDEA запускаются аналогично тестам JUnit. Для запуска теста необходимо щелкнуть мышью на зеленом значке, расположенном в области с номерами строк кода на том же уровне, что и заголовок метода с тестом (поз. 1 на рис. 14.5), а затем в появившемся меню (поз. 2) выбрать пункт **Run *<имя метода>***. Например, для запуска теста `BookToApiConverterTestNgTest#convert` щелкните на указанном стрелкой (поз. 1) зеленом значке левее заголовка метода в области с номерами строк, а затем выберите пункт **Run 'convert()'**, как показано на рис. 14.5.

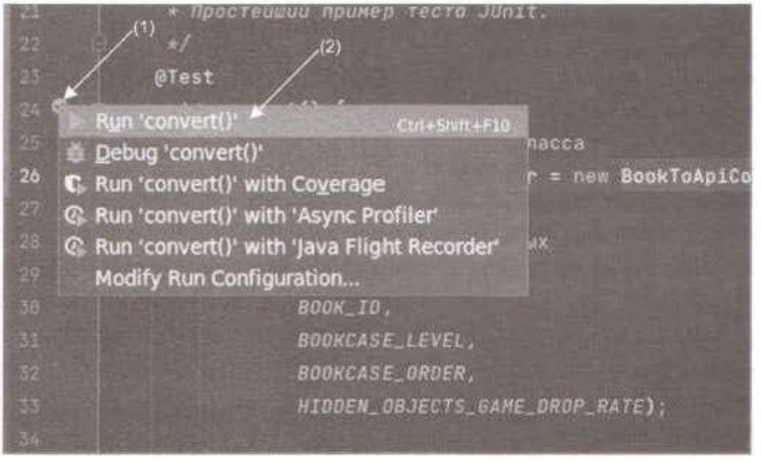


Рис. 14.5. Запуск тестов TestNG в IntelliJ IDEA

Результат выполнения тестов отобразится в точно таком же окне, как и результат выполнения тестов JUnit (см. рис. 14.2).

Для запуска тестов TestNG в Eclipse следует воспользоваться опциями **Run** и **Debug**, расположенными над методами тестов (рис. 14.6).

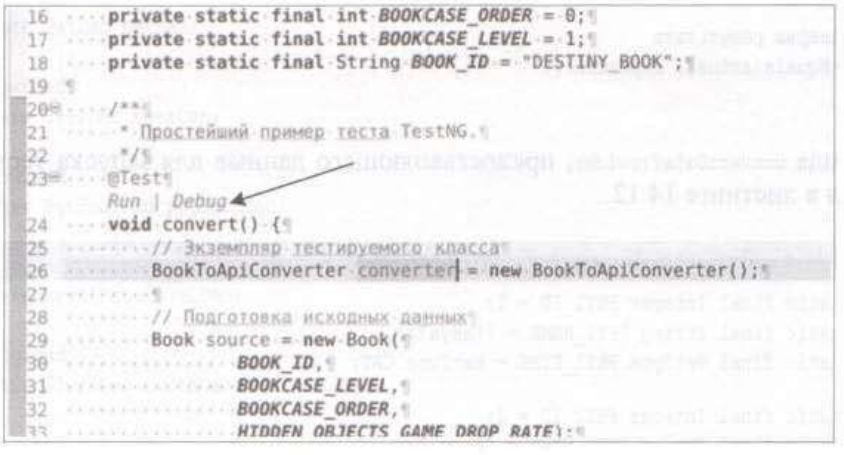


Рис. 14.6. Запуск тестов TestNG в Eclipse

14.3.4. Параметризованные тесты

Аналогично фреймворку JUnit фреймворк TestNG также предоставляет возможность писать параметризованные методы. Синтаксис параметризованных тестов TestNG несколько отличается от синтаксиса параметризованных тестов JUnit. Аннотация `@Test` из TestNG имеет атрибут `dataProvider`, в котором указывается имя метода, предоставляющего данные для запуска тестов (листинг 14.11).

Листинг 14.11. PetToApiConverterTestNgTest.java

```
/**
 * Простейший пример параметризованного теста TestNg.
 * @param petId Первичный ключ питомца.
 * @param petName Имя питомца.
 * @param petType Тип питомца.
 */
@Test(dataProvider = "convertDataProvider")
void convert(Integer petId, String petName, PetType petType) {
    // Экземпляр тестируемого класса
    PetToApiConverter converter = new PetToApiConverter();

    // Подготовка исходных данных
    Pet source = new Pet();
    source.setId(petId);
    source.setName(petName);
    source.setPetType(petType);

    // Подготовка ожидаемого результата
    var expected = new PetInfo(
                    petId,
                    petName,
                    petType);

    // Вызов тестируемого метода
    var actual = converter.convert(source);

    // Проверка результата
    assertEquals(actual, expected);
}
```

Код метода `convertDataProvider`, предоставляющего данные для запуска теста `convert`, приведен в листинге 14.12.

Листинг 14.12. PetToApiConverterTestNgTest.java

```
private static final Integer PET1_ID = 1;
private static final String PET1_NAME = "Vasya";
private static final PetType PET1_TYPE = PetType.CAT;

private static final Integer PET2_ID = 2;
private static final String PET2_NAME = "Котик";
private static final PetType PET2_TYPE = PetType.CAT;
```

```
/**
 * Подготовка данных параметризованного теста.
 * @return Параметры теста.
 */
@DataProvider
static Object[][] convertDataProvider() {
    return new Object[][] {
        {PET1_ID, PET1_NAME, PET1_TYPE},
        {PET2_ID, PET2_NAME, PET2_TYPE}
    };
}
```

14.4. Фреймворк Mockito

14.4.1. Введение

При рассмотрении фреймворков JUnit и TestNG в предыдущих разделах главы тестируемые классы были простейшими и не имели зависимостей. В реальных приложениях классы вызывают методы интерфейсов и других классов, а значит, в тестах необходимо каким-либо образом инициализировать их экземпляры. При этом реализации классов, которые используются при запуске приложения, имеют свои зависимости.

Рассмотрим для примера класс `PetServiceImpl` (листинг 14.13).

Листинг 14.13. PetServiceImpl.java

```
@Service("petService")
public class PetServiceImpl implements PetService, PetApiService {

    @Autowired
    private RoomDao roomDao;

    @Autowired
    private PetDao petDao;

    @Autowired
    private UserDao userDao;

    @Autowired
    private PetFoodDao petFoodDao;

    @Autowired
    private LevelDao levelDao;

    @Autowired
    private ClothDao clothDao;

    @Autowired
    private PetJournalEntryDao petJournalEntryDao;
```

```
    @Autowired
    private Clock clock;

    @Autowired
    private ConversionService conversionService;
```

Огромное количество зависимостей класса `PetServiceImpl` в приведенном здесь коде берется из контекста Spring (аннотация `@Autowired`). В модульном тестировании контекст Spring не поднимается, но класс `PetServiceImpl` не может работать без своих зависимостей.

Поскольку `PetServiceImpl` использует в качестве зависимостей только интерфейсы, хорошим вариантом решения проблемы станет создание отдельных тестовых реализаций интерфейсов, которые возвращают фиксированные значения из своих методов, а затем использование этих зависимостей в тестах. Подобные тестовые реализации достаточно легко создаются без дополнительных фреймворков, но зависимостей много, а их создание и использование сильно загромоздит код тестов.

В современной разработке для создания тестовых реализаций используется фреймворк Mockito, позволяющий создавать mock-объекты для автоматических тестов. Mockito поддерживает как тесты JUnit, так и тесты TestNG.

14.4.2. Зависимости Mockito + JUnit

Для использования Mockito в проектах на JUnit необходимо подключить соответствующую зависимость (листинг 14.14).

Листинг 14.14. Файл pom.xml

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-junit-jupiter</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
```

Здесь `${mockito.version}` — версия Mockito, объявленная в разделе `properties` файла pom.xml (листинг 14.15).

Листинг 14.15. Файл pom.xml

```
<properties>
    <mockito.version>5.12.0</mockito.version>
</properties>
```

В проектах на Spring Boot стартер `spring-boot-starter-test` уже включает в себя зависимость от Mockito (листинг 14.16).

Листинг 14.16. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

14.4.3. Примеры в тестовом приложении

В проектах virtualpets-server-springframework и virtualpets-server-springboot пример использования Mockito находится в тестах:

- `PetServiceImplJUnitTest` — пример теста с использованием JUnit и Mockito;
- `UserServiceTestNgTest` — пример теста с использованием TestNG и Mockito.

Использование Mockito в проектах virtualpets-server-springframework и virtualpets-server-springboot ничем не различается.

14.4.4. Интеграция с тестом JUnit

Тест `PetServiceImplJUnitTest` использует аннотацию `@ExtendWith`, которая применяется в JUnit для добавления расширений. Поддержка Mockito добавляется расширением `MockitoExtension` из пакета `org.mockito.junit.jupiter` (листинг 14.17).

Листинг 14.17. PetServiceImplJUnitTest.java

```
@ExtendWith(MockitoExtension.class)
public class PetServiceImplJUnitTest {
```

Расширение `MockitoExtension` позволяет использовать аннотации `@Mock`, `@Spy` и `@InjectMocks`.

14.4.5. Mock-объекты

Аннотация `@Mock` создает mock-объекты, методы которых возвращают значения, задаваемые с помощью методов `when` из класса `org.mockito`, а также позволяют отслеживать вызовы своих методов.

Аннотация `@Spy` создает spy-объекты, которые работают аналогично исходным реальным объектам, но позволяют отслеживать вызовы методов, как и mock-объекты.

Например, класс `PetServiceImpl` содержит метод `getPetNewJournalEntriesCount` (листинг 14.18).

Листинг 14.18. PetServiceImpl.java

```
/**
* Возвращает количество новых, еще непрочитанных записей в дневнике питомца.
* @param petId Первичный ключ питомца.
* @return Количество новых записей.
*/
@Override
public Long getPetNewJournalEntriesCount(Integer petId) {
 return petDao.getPetNewJournalEntriesCount(petId);
}
```

Логики в этом методе фактически никакой нет — он содержит только строку вызова метода из слоя DAO. Следовательно, для проверки того, что метод отработал правильно, необходимо проверить, что он вызвал `petDao` с параметром `petId` и вернул его результат в качестве своего результата.

При модульном тестировании не используется реальная реализация интерфейса `PetDao` — необходимо задействовать тестовую реализацию, которая при вызове с заранее определенными для теста параметрами вернет заранее определенный результат.

Объявление mock-объекта `petDao` с помощью Mockito приведено в листинге 14.19.

Листинг 14.19. PetServiceImplJUnitTest.java

```
@Mock
private PetDao petDao;
```

Внедрение объявленных mock-объектов в тестируемый объект PetServiceImpl показано в листинге 14.20.

Листинг 14.20. PetServiceImplJUnitTest.java

```
@InjectMocks
private PetServiceImpl service;
```

Аннотация `@InjectMocks` создает экземпляр объекта `PetServiceImpl` и внедряет в него mock-объекты из текущего теста.

14.4.6. Настройка возвращаемых значений

В самом методе теста с помощью статического метода `org.mockito.Mockito#when` задается возвращаемое mock-объектом значение (листинг 14.21).

Листинг 14.21. PetServiceImplJUnitTest.java

```
when(petDao.getPetNewJournalEntriesCount(eq(10))).thenReturn(2L);
```

Метод `when` здесь — это статический метод `org.mockito.Mockito#when`. Обычно он импортируется через `import static` — чтобы не загромождать лишними упоминаниями класса Mockito в самих тестах (листинг 14.22).

Листинг 14.22. PetServiceImplJUnitTest.java

```
import static org.mockito.Mockito.when;
```

В качестве параметра метода `when` в коде листинга 14.21 передается специально оформленный объект, описывающий ожидаемый вызов метода mock-объекта. Но вызов `petDao.getPetNewJournalEntriesCount` — это не вызов реального метода, он не возвращает какого-либо осмысленного результата. Код `petDao.getPetNewJournalEntriesCount(eq(10))` в листинге 14.21 следует читать как вызов метода `getPetNewJournalEntriesCount` с параметром, равным `10`.

Метод `eq` — это статический метод класса `ArgumentMatchers`, который чаще всего импортируется через `import static`, чтобы не загромождать тестовые методы (листинг 14.23).

Листинг 14.23. PetServiceImplJUnitTest.java

```
import static org.mockito.ArgumentMatchers.eq;
```

С помощью метода `thenReturn` (см. листинг 14.21) указывается значение, которое должен вернуть mock-объект при вызове метода, описанного в статическом методе `when`.

В итоге после подобной настройки mock-объект `petDao` при вызове метода `getPetNewJournalEntriesCount` с параметром `10` вернет значение `2`. Полный код этого теста приведен в листинге 14.24.

Листинг 14.24. PetServiceImplJUnitTest.java

```
@Test
void getPetJournalEntriesCount_ok() {
    // Подготовка mock-объектов
    when(petDao.getPetNewJournalEntriesCount(eq(10))).thenReturn(2L);

    // Вызов тестируемого метода
    Long actual = service.getPetNewJournalEntriesCount(10);

    // Проверка результата
    assertEquals(Long.valueOf(2L), actual);
}
```

Приведенный здесь тест — это самый легкий пример использования Mockito. Метод `getPetNewjournalEntriesCount` крайне простой, и единственное, чем он занимается, — это делегирует свое выполнение аналогичному методу из `petDao`. Однако классы в слое бизнес-логики в большинстве случаев имеют какое-либо ветвление, производят вычисления и т. д.

14.4.7. Дополнительные примеры

В качестве примера метода, который не только использует mock-объект, но и содержит свою логику, хорошо подойдет метод addExperience из класса PetServiceImpl (листинг 14.25).

Листинг 14.25. PetServiceImpl.java

```
/**
 * Добавление опыта питомцу. Обрабатывает переход питомца на следующий уровень
 * при достижении необходимого порога опыта.
 * @param pet Питомец.
 * @param exp Добавляемый опыт.
 */
@Override
public void addExperience(Pet pet, Integer exp) {
    int nextExperience = pet.getExperience() + exp;
    Optional<Level> nextLevelOpt = levelDao.findById(
            pet.getLevel().getId() + 1);
    nextLevelOpt.ifPresentOrElse((nextLevel) -> {
        pet.setExperience(nextExperience);
        if (nextExperience >= nextLevel.getExperience()) {
            pet.setLevel(nextLevel);
        }
    }, () -> {
        Level lastLevel = levelDao.findById(pet.getLevel().getId()).orElseThrow();
        pet.setExperience(Math.min(nextExperience, lastLevel.getExperience()));
    });
}
```

Метод addExperience содержит ветвление:

- если из слоя DAO вернулось непустое значение (т. е. существует следующий уровень, которого питомец может достичь), то увеличение опыта происходит с проверкой достижения следующего уровня — а это еще одно ветвление:
  - если питомец после получения опыта достигает порогового значения следующего уровня, то он получает новый уровень;
  - в противном случае увеличение опыта происходит без получения питомцем нового уровня;
- если из слоя DAO вернулось пустое значение (т. е. питомец достиг максимального уровня), то увеличение опыта происходит с проверкой на достижение максимального опыта текущего уровня. Питомец не может получить опыт выше максимального значения, допустимого для текущего достигнутого уровня.
- При тестировании метода addExperience необходимо проверить все ответвления (листинг 14.26).

Листинг 14.26. PetServiceImplJUnitTest.java

```
/**
 * Тестирование основной ветки повышения опыта.
 */
@Test
void testAddExperience_ok() {
    // Подготовка тестовых данных
    Pet pet = new Pet();
    pet.setExperience(0);
    Level level1 = new Level(LEVEL1_ID, LEVEL1_EXPERIENCE);
    Level level2 = new Level(LEVEL2_ID, LEVEL2_EXPERIENCE);
    pet.setLevel(level1);

    // Настройка mock-объектов
    when(levelDao.findById(LEVEL2_ID))
            .thenReturn(Optional.of(level2));

    // Вызов тестируемого метода
    service.addExperience(pet, 1);

    // Проверка результата
    assertEquals(1, pet.getExperience());
}

/**
 * Тестирование ветки повышения опыта, в которой питомец получает
 * новый уровень.
 */
@Test
void testAddExperience_levelUp() {
    // Подготовка тестовых данных
    Pet pet = new Pet();
    pet.setExperience(9);
    Level level1 = new Level(LEVEL1_ID, LEVEL1_EXPERIENCE);
    Level level2 = new Level(LEVEL2_ID, LEVEL2_EXPERIENCE);
    pet.setLevel(level1);

    // Настройка mock-объектов
    when(levelDao.findById(LEVEL2_ID))
            .thenReturn(Optional.of(level2));

    // Вызов тестируемого метода
    service.addExperience(pet, 1);

    // Проверка результата
    assertEquals(10, pet.getExperience());
    assertEquals(level2, pet.getLevel());
}
```

```
/**
 * Тестирование ветки повышения опыта, в которой питомец уже достиг максимального уровня.
 */
@Test
void testAddExperience_lastLevel() {
    // Подготовка тестовых данных
    Pet pet = new Pet();
    pet.setExperience(10);
    Level level2 = new Level(LEVEL2_ID, LEVEL2_EXPERIENCE);
    pet.setLevel(level2);

    // Настройка mock-объектов
    when(levelDao.findById(LEVEL2_ID))
            .thenReturn(Optional.of(level2));
    when(levelDao.findById(LEVEL2_ID + 1))
            .thenReturn(Optional.empty());

    // Вызов тестируемого метода
    service.addExperience(pet, 1);

    // Проверка результата
    assertEquals(10, pet.getExperience());
    assertEquals(level2, pet.getLevel());
}
```

14.4.8. Подсчет вызова методов

Следующая возможность mock-объектов — проверка вызовов методов. Примером этому может служить метод `updatePetsTask` класса `PetServiceImpl` из проекта virtualpets-server-springframework, приведенный в листинге 14.27 (в проекте virtualpets-server-springboot метод выглядит иначе, поэтому необходимо смотреть код именно в virtualpets-server-springframework).

Листинг 14.27. PetServiceImpl.java

```
@Override
public void updatePetsTask() {
    petDao.updatePetsTask();
}
```

Метод `updatePetsTask` не принимает никаких параметров и ничего не возвращает — соответственно в тесте не получится проверить возвращаемое значение. Единственное, что действительно необходимо проверить, — был ли вызван метод `updatePetsTask` из `petDao`. Для этого используется статический метод `verify` из `org.mockito.Mockito`, который принято импортировать через `import static` (листинг 14.28).

Листинг 14.28. PetServiceImplJUnitTest.java

```
import static org.mockito.Mockito.verify;
```

Метод `verify` проверяет, что требуемый метод был вызван определенное количество раз. Количество ожидаемых вызовов метода передается вторым параметром. Например, так производится проверка того, что метод `updatePetsTask` mock-объекта `petDao` был вызван один раз:

```
verify(petDao, times(1)).updatePetsTask();
```

Метод `times` — статический метод класса `org.mockito.Mockito`, который также импортируется чаще всего через `import static`:

```
import static org.mockito.Mockito.times;
```

Если в тесте ожидаемого в `verify` вызова метода не произошло либо вызовов было не столько, сколько ожидалось, то тест закончится неудачей.

Метод `verify` перегружен. Вариант без параметров используется в том случае, когда ожидается ровно один вызов метода, т. е. вариант:

```
verify(petDao, times(1)).updatePetsTask();
```

по своему действию идентичен варианту:

```
verify(petDao).updatePetsTask();
```

Полный код теста `updatePetsTask` приведен в листинге 14.29.

Листинг 14.29. PetServiceImplJUnitTest.java

```
@Test
void updatePetTask() {
    // Вызов тестируемого метода
    service.updatePetsTask();

    // Проверка, что метод слоя DAO действительно вызывался.
    verify(petDao).updatePetsTask();
}
```

14.4.9. Перехват параметров

В некоторых случаях при написании тестов методов, не возвращающих значение, необходимо не только убедиться, что определенный метод вызывался, но и проверить, с какими параметрами он вызывался. Рассмотрим, например, метод создания нового питомца, приведенный в листинге 14.30.

Листинг 14.30. PetServiceImpl.java

```
@Override
@PreAuthorize("hasRole('USER')")
@Transactional(rollbackFor = ServiceException.class)
public void create(UserPetDetails userPetDetails,
   CreatePetArg createPetArg) throws ServiceException {
 Pet pet = new Pet();
 pet.setName(createPetArg.name());
 pet.setCreatedDate(OffsetDateTime.now(clock));
```

```
 pet.setUser(userDao.getReference(userPetDetails.userId()));
 pet.setComment(createPetArg.comment());
 pet.setPetType(createPetArg.petType());
 Level level = levelDao.findById(1).orElseThrow();
 pet.setLevel(level);
 petDao.save(pet);
}
```

Метод `PetServiceImpl#create` не возвращает никаких значений. Он подготавливает данные для слоя DAO и вызывает метод `petDao.save` с заполненным параметром `pet`. Модульный тест должен проверить, что `petDao.save` вызывается с правильно заполненным экземпляром сущности `Pet`. Передаваемый в DAO экземпляр `Pet` создается и заполняется внутри метода. Для того чтобы проверить правильность его заполнения, необходимо перехватить фактическое переданное в метод `petDao.save` значение.

Перехватывание параметров методов mock-объектов осуществляется с помощью `ArgumentCaptor`:

```
ArgumentCaptor<Pet> captor = ArgumentCaptor.forClass(Pet.class);
```

Созданный подобный образом `ArgumentCaptor` используется при проверке вызова метода mock-объекта после вызова тестируемого метода:

```
verify(petDao).save(captor.capture());
```

После вызова метода `capture` фактическое захваченное значение возвращается методом `getValue` экземпляра `ArgumentCaptor`:

```
Pet actual = captor.getValue();
```

Полученное из `ArgumentCaptor` значение проверяется методами `assert*`:

```
assertEquals(PET_NAME, actual.getName());
assertEquals(PET_TYPE, actual.getPetType());
assertEquals(PET_COMMENT, actual.getComment());
assertEquals(user, actual.getUser());
```

Полный код тестирования метода `PetServiceImpl#create` приведен в листинге 14.31.

Листинг 14.31. PetServiceImplJUnitTest.java

```
private static final Integer PET_ID = 1;

private static final String PET_NAME = "Котик";

private static final PetType PET_TYPE = PetType.CAT;

private static final String PET_COMMENT = "Просто комментарий";

private static final Integer USER_ID = 1;

/**
 * Пример использования ArgumentCaptor
 * @throws ServiceException
 */
```

```
@Test
void create() throws ServiceException {
    // Подготовка тестовых данных
    UserPetDetails userPetDetails = new UserPetDetails(
            USER_ID, PET_ID);
    CreatePetArg createPetArg = new CreatePetArg(
            PET_NAME, PetType.CAT, PET_COMMENT);

    // Подготовка mock-объектов
    User user = new User();
    when(userDao.getReference(userPetDetails.userId()))
        .thenReturn(user);
    Level level = new Level(1, 0);
    Optional<Level> levelOpt = Optional.of(level);
    when(levelDao.findById(eq(1))).thenReturn(levelOpt);

    // Вызов тестируемого метода
    service.create(userPetDetails, createPetArg);

    // Проверка результата
    ArgumentCaptor<Pet> captor = ArgumentCaptor.forClass(Pet.class);
    // Перехват значения, переданного в petDao.save
    verify(petDao).save(captor.capture());
    // Проверка перехваченного значения
    Pet actual = captor.getValue();
    assertEquals(PET_NAME, actual.getName());
    assertEquals(PET_TYPE, actual.getPetType());
    assertEquals(PET_COMMENT, actual.getComment());
    assertEquals(user, actual.getUser());
}
```

14.4.10. Зависимости Mockito + TestNG

Зависимости, необходимые для подключения Mockito и TestNG в проекте на Spring Framework, приведены в листинге 14.32.

Листинг 14.32. Файл pom.xml

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-testng</artifactId>
    <version>0.5.2</version>
    <scope>test</scope>
</dependency>
```

Зависимости, необходимые для подключения Mockito и TestNG в проекте на Spring Boot, приведены в листинге 14.33.

Листинг 14.33. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-testng</artifactId>
    <version>0.5.2</version>
    <scope>test</scope>
</dependency>
```

Использование Mockito в проектах virtualpets-server-springframework и virtualpets-server-springboot ничем не различается.

14.4.11. Интеграция с тестом TestNG

Интеграция Mockito и TestNG происходит несколько отличным от интеграции JUnit способом, но сама работа с аннотациями `@Mock`, `@Spy`, `@InjectMocks` реализуется точно так же, как и в случае JUnit.

Подключение Mockito к тесту осуществляется с помощью добавления `MockitoTestNGListener` к списку листенеров:

```
@Listeners(MockitoTestNGListener.class)
public class UserServiceTestNgTest {
```

После добавления листенера в тесте станут работать аннотации `@Mock`, `@Spy` и `@InjectMocks`. Пример для теста `UserServiceTestNgTest` из проекта virtualpets-server-springboot приведен в листинге 14.34.

Листинг 14.34. UserServiceTestNgTest.java

```
@Mock
private UserDao userDao;

private ZoneId zoneId = ZoneId.of("Europe/Moscow");

@Spy
private Clock clock = Clock.fixed(ZonedDateTime
        .of(LocalDate.of(2024, 8, 14), LocalTime.of(16, 58), zoneId)
        .toInstant(), zoneId);

@InjectMocks
private UserServiceImpl userService;
```

Статические методы `when` и `verify` здесь работают аналогично тестам с JUnit.

Полный текст теста UserServiceTestNgTest из проекта virtualpets-server-springboot приведен в листинге 14.35.

Листинг 14.35. UserServiceTestNgTest.java

```
package ru.urvanov.virtualpets.server.service;

import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertNotNull;
import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.Mockito.when;

import java.time.Clock;
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.OffsetDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.util.ArrayList;
import java.util.List;

import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.Spy;
import org.mockito.testng.MockitoTestNGListener;
import org.testng.annotations.Listeners;
import org.testng.annotations.Test;

import ru.urvanov.virtualpets.server.controller.api.domain.RefreshUsersOnlineResult;
import ru.urvanov.virtualpets.server.controller.api.domain.UserInfo;
import ru.urvanov.virtualpets.server.dao.UserDao;
import ru.urvanov.virtualpets.server.dao.domain.User;
import ru.urvanov.virtualpets.server.service.domain.UserPetDetails;
import ru.urvanov.virtualpets.server.service.exception.ServiceException;

@Listeners(MockitoTestNGListener.class)
public class UserServiceTestNgTest {

    private static final Integer USER_ID = 1;

    private static final Integer PET_ID = 2;

    private static final String USER_FULL_NAME = "Иванов Иван";

    @Mock
    private UserDao userDao;

    private ZoneId zoneId = ZoneId.of("Europe/Moscow");

    @Spy
    private Clock clock = Clock.fixed(ZonedDateTime
            .of(LocalDate.of(2024, 8, 14), LocalTime.of(16, 58), zoneId)
            .toInstant(), zoneId);
```

```
    @InjectMocks
    private UserServiceImpl userService;

    @Test
    void getUsersOnline() throws ServiceException {
        // Подготовка тестовых данных
        UserPetDetails userPetDetails = new UserPetDetails(USER_ID, PET_ID);
        List<User> usersOnline = new ArrayList<>();
        User user1 = new User();
        user1.setId(USER_ID);
        user1.setName(USER_FULL_NAME);
        usersOnline.add(user1);

        // Настройка mock-объектов
        OffsetDateTime activeAfterDate = OffsetDateTime.now(clock).minusMinutes(5);
        when(userDao.findByActiveDateAfter(eq(activeAfterDate))).thenReturn(usersOnline);

        // Вызов тестируемого метода
        RefreshUsersOnlineResult actual = userService.getUsersOnline(userPetDetails);

        // Проверка результата
        assertNotNull(actual.users());
        assertEquals(actual.users().size(), 1);
        UserInfo actualUser0 = actual.users().get(0);
        assertEquals((Integer) actualUser0.id(), (Integer) USER_ID);
        assertEquals(actualUser0.name(), USER_FULL_NAME);
    }
}
```

14.5. Фреймворк Spring

14.5.1. Введение

Модульные тесты, проверяющие логику одного конкретного метода в изоляции от окружения, быстро выполняются, и их легко создавать. Основной их недостаток заключается в том, что они не поднимают контекст Spring, а значит, до запуска приложения разработчик не узнает об ошибках, которые он допустил в его конфигурировании. Ими могут быть:

- отсутствующие зависимости;
- неправильные названия бинов;
- проблемы внедрения бинов;
- ошибки конфигурации модулей Spring;
- ошибки описания сущностей JPA;
- а также все остальные ошибки, которые могут быть связаны с некорректной настройкой контекста Spring и работой приложения в целом.

Интеграция Spring и JUnit достаточно похожа как в Spring Framework, так и в Spring Boot, но Spring Boot предоставляет дополнительные аннотации, облегчающие создание тестов.

14.5.2. Подключение зависимостей

Подключение зависимостей для проекта на Spring Framework приведено в листинге 14.36.

Листинг 14.36. Файл pom.xml

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${org.springframework-version}</version>
    <scope>test</scope>
</dependency>
```

Здесь `${org.springframework-version}` — это версия Spring Framework, а `scope = test` нужен для подключения зависимости только для фазы тестирования.

Для проекта на Spring Boot все необходимые зависимости подключаются с помощью стартера `spring-boot-starter-test` (листинг 14.37).

Листинг 14.37. Файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

14.5.3. Интеграция JUnit и Spring

Интеграция JUnit со Spring осуществляется с использованием расширения `SpringExtension` из пакета `org.springframework.test.context.junit.jupiter` (листинг 14.38).

Листинг 14.38. BaseDaoImplTest.java

```
@ExtendWith(SpringExtension.class)
...
class BaseDaoImplTest {
...
}
```

Расширение `SpringExtension` поднимает контекст Spring. Конфигурация задается с помощью атрибутов `classes` и `locations` аннотации `@ContextConfiguration`: в атрибуте `locations` записываются XML-файлы конфигураций, а в атрибуте `classes` указываются классы с Java-конфигурацией (листинг 14.39).

Листинг 14.39. BaseDaoImplTest.java

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {DaoTestConfig.class, ClockConfig.class,
        DataSourceConfig.class})
...
class BaseDaoImplTest {
...
}
```

Здесь `DaoTestConfig` — класс с аннотацией `@Configuration`, который инициализирует бины приложения virtualpets-server-springframework, необходимые для тестирования слоя DAO (листинг 14.40).

Листинг 14.40. DaoTestConfig.java

```
package ru.urvanov.virtualpets.server.controller.config;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
import org.springframework.context.annotation.Profile;

/**
 * Конфигурация Spring для тестирования слоя DAO отдельно от всех остальных слоев приложения.
 */
@Configuration
@ImportResource("file:src/main/webapp/WEB-INF/spring/servlet-tx.xml")
@ComponentScan(basePackages = {"ru.urvanov.virtualpets.server.dao"})
@Profile("test-dao")
public class DaoTestConfig {
}
```

Аннотация `@ImportResource` подключает здесь файл конфигурации servlet-tx.xml, в котором настраиваются бины библиотеки Liquibase, работы с транзакциями и `EntityManagerFactory`, которые создают схему базы данных и осуществляют работу с сущностями.

С помощью `@ComponentScan` указывается пакет, в котором находятся бины слоя DAO. Бины из этого пакета используют бины, создаваемые в файле servlet-tx.xml.

14.5.4. Библиотека Testcontainers

Остается сконфигурировать источник данных. Одним из вариантов источника данных для тестов может стать реальная база данных, развернутая на компьютере разработчика. У подобного подхода есть серьезный недостаток — каждому разработчику придется настраивать и запускать тестовую базу данных самостоятельно. Для тестов необходима также база данных PostgreSQL, которая запускается автоматически перед запуском тестов, а затем — после выполнения тестов — подчищает всё за собой и самостоятельно останавливается.

Проекты virtualpets-server-springframework и virtualpets-server-springboot используют Testcontainers for Java — библиотеку Java, поддерживающую JUnit и предоставляющую простой способ доступа к экземплярам популярных библиотек в Docker-контейнерах.

Testcontainers поддерживает не только язык Java, но и другие языки: Go, .NET, Node.js, Python, Rust, Haskell, Ruby, Clojure, Elixir. У каждого из этих языков программирования имеются свои особенности использования Testcontainers. В любом случае Testcontainers — это фактически стандарт при написании тестов.

Подключение Testcontainers к проекту virtualpets-server-springframework показано в листинге 14.41.

Листинг 14.41. Файл pom.xml

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <version>${testcontainers.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>${testcontainers.version}</version>
    <scope>test</scope>
</dependency>
```

Здесь `${testcontainers.version}` — это версия Testcontainers, объявленная в секции `properties` файла pom.xml (листинг 14.42).

Листинг 14.42. Файл pom.xml

```
<properties>
    <testcontainers.version>1.20.1</testcontainers.version>
</properties>
```

Подключение Testcontainers к проекту virtualpets-server-springboot реализуется аналогично. Единственное различие состоит в том, что в этом случае нет необходимости указывать версию Testcontainers — она подтягивается из зависимостей Spring Boot (листинг 14.43).

Листинг 14.43. Файл pom.xml

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>postgresql</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
```

```
    <artifactId>junit-jupiter</artifactId>
    <scope>test</scope>
</dependency>
```

Управление контейнером осуществляется аннотацией `@Container` совместно с аннотацией `@Testcontainers` (листинг 14.44).

Листинг 14.44. BaseDaoImplTest.java

```
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

...
@Testcontainers
class BaseDaoImplTest {

    /**
     * Управляет запуском и остановкой контейнера PostgreSQL.
     * При запуске нескольких тестов контейнер создается один раз
     * и переиспользуется последующими тестами.
     */
    @Container
    public static PostgreSQLContainer<?> postgreSQLContainer
            = new PostgreSQLContainer<>("postgres:16.1");
}
```

База данных PostgreSQL в запущенном контейнере доступна по адресу `jdbc:tc:postgresql:16.1:///databasename`. При этом поддерживается передача скрипта инициализации в параметре `TC_INITSCRIPT`. Вот пример адреса подключения со скриптом инициализации:

```
jdbc:tc:postgresql:16.1:///databasename?TC_INITSCRIPT=init.sql
```

14.5.5. Класс *SingleConnectionDataSource*

Тестам DAO необходим источник данных `DataSource`, настроенный на работающую базу PostgreSQL. Поскольку в тестах проектов virtualpets-server-springframework и virtualpets-server-springboot используется Testcontainers, то для получения требуемых данных следует подключаться к PostgreSQL, запускаемому в поднимаемом им контейнере PostgreSQL. Поэтому перед подключением к базе данных PostgreSQL необходимо создать экземпляр `DataSource`.

Проект virtualpets-server-springframework при реальном запуске работает в сервере приложений Apache Tomcat и использует пулы подключений Commons DPCP2 и Commons Pool 2, предоставляемые сервером приложений. Проект virtualpets-server-springboot при реальном запуске использует пул подключений HikariCP. Для тестов пулы подключений не нужны — достаточно одного подключения, которое можно использовать во всех тестах.

Класс `SingleConnectionDataSource` из пакета `org.springframework.jdbc.datasource`, входящий в состав артефакта spring-jdbc, — это специальная реализация `DataSource` для тестов, метод `getConnection` которой всегда возвращает одно и то же подключение.

Экземпляр `SingleConnectionDataSource` необходимо создать как бин Spring. Создание бина вынесено в отдельный файл конфигурации, чтобы его было легко задействовать при других типах тестов, а не только слоя DAO (листинг 14.45).

Листинг 14.45. DataSourceConfig.java

```
package ru.urvanov.virtualpets.server.controller.config;

import java.io.IOException;
import java.sql.SQLException;

import javax.sql.DataSource;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.SingleConnectionDataSource;

@Configuration
@Profile({"test-dao", "test-mock-mvc"})
public class DataSourceConfig {

    /**
     * Настройка источника данных на базу данных
     * PostgreSQL в контейнере Testcontainers.
     * @return Источник данных для тестов.
     * @throws SQLException
     */
    @Bean(destroyMethod = "close")
    public DataSource dataSource() throws IOException, SQLException {
        // DataSource с одним всегда работающим подключением.
        // Фактически это не пул подключений, он всегда возвращает
        // одно и то же подключение к базе данных, не создавая новое.
        SingleConnectionDataSource result = new SingleConnectionDataSource();

        // jdbc:tc ... - это адрес к базе данных в контейнере
        // Адрес взят из документации Testcontainers.
        result.setUrl("""
                jdbc:tc:postgresql:16.1:///databasename?\
                TC_INITSCRIPT=init.sql""");

        // ContainerDatabaseDriver работает с подключениями вида
        // jdbc:tc
        result.setDriverClassName(
                "org.testcontainers.jdbc.ContainerDatabaseDriver");

        // Рабочая схема с таблицами проекта.
        result.setSchema("virtualpets_server_springframework");
```

```
            // Заглушаем вызовы close в возвращаемом подключении,
            // чтобы никто не закрывал наше единственное подключение.
            // В реальности это оборачивает возвращаемый Connection
            // в прокси, в котором вызовы close игнорируются.
            result.setSuppressClose(true);
            return result;
        }
}
```

Java-конфигурация `DataSourceConfig` подключается к базовому классу тестов `BaseDaoImplTest` с помощью аннотации `@ContextConfiguration`, как и остальные классы Java-конфигурации (листинг 14.46).

Листинг 14.46. BaseDaoImplTest.java

```
@ContextConfiguration(classes = {DaoTestConfig.class, ClockConfig.class,
        DataSourceConfig.class})
```

14.5.6. Класс *ClockConfig*

Коды `DaoTestConfig` и `DataSourceConfig` уже приводились ранее (см. листинги 14.40 и 14.45 соответственно). Класс Java-конфигурации `ClockConfig` создает экземпляр `Clock`, всегда возвращающий фиксированное время для повторяемости тестов (листинг 14.47).

Листинг 14.47. ClockConfig.java

```
package ru.urvanov.virtualpets.server.controller.config;

import java.time.Clock;
import java.time.Instant;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.temporal.TemporalAccessor;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Profile;

@Profile({"test-dao", "test-mock-mvc"})
public class ClockConfig {
    /**
     * Настраивает экземпляр Clock, возвращающий всегда
     * одну и ту же дату и одно и то же время для предсказуемости  тестов.
     * @return Экземпляр Clock с фиксированной датой и временем.
     */
    @Bean
    public Clock clock() {
        ZoneId zoneId = ZoneId.of("Europe/Moscow");
        TemporalAccessor offsetDateTime = ZonedDateTime.of(
                2024, 3, 15, 18, 52, 0, 0, zoneId);
```

```
        Instant instant = Instant.from(offsetDateTime );
        return Clock.fixed(instant, zoneId);
    }
}
```

14.5.7. Класс *BaseDaoImplTest*

В листинге 14.48 представлен конечный код класса `BaseDaoImplTest` проекта virtualpets-server-springframework.

Листинг 14.48. BaseDaoImplTest.java

```
package ru.urvanov.virtualpets.server.dao;

import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.transaction.annotation.Transactional;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import ru.urvanov.virtualpets.server.controller.config.ClockConfig;
import ru.urvanov.virtualpets.server.controller.config.DaoTestConfig;
import ru.urvanov.virtualpets.server.controller.config.DataSourceConfig;

/**
 * Базовый класс для тестов слоя DAO
 */
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = {DaoTestConfig.class, ClockConfig.class,
        DataSourceConfig.class})
@Testcontainers
@ActiveProfiles({"test", "test-dao"})
@Transactional
class BaseDaoImplTest {

    /**
     * Управляет запуском и остановкой контейнера PostgreSQL.
     * При запуске нескольких тестов контейнер создается один раз
     * и переиспользуется последующими тестами.
     */
    @Container
    public static PostgreSQLContainer<?> postgreSQLContainer
            = new PostgreSQLContainer<>("postgres:16.1");
}
```

Обратите внимание, что с помощью аннотации `@ActiveProfiles` здесь включены профили `test` и `test-dao`. Классы Java-конфигурации `DataSourceConfig` и `ClockConfig` работа-

ют только для профилей `test-dao` и `test-mock-mvc`, т. к. именно эти профили указаны в аннотации `@Profile` для них (листинг 14.49).

Листинг 14.49. DataSourceConfig.java

```
@Profile({"test-dao", "test-mock-mvc"})
```

14.5.8. Аннотация *@DataJpaTest*

Класс `BaseDaoImplTest` в проекте virtualpets-server-springboot отличается от класса `BaseDaoImplTest` в проекте virtualpets-server-springframework. Различие вызвано тем, что Spring Boot добавляет специальные аннотации для тестирования отдельных слоев приложения:

- `@DataJpaTest` — для тестирования слоя DAO. По умолчанию аннотация `@DataJpaTest` сканирует классы сущностей и репозитории Spring Data JPA;
- `@JdbcTest` — для тестов, которым необходимы только `DataSource` и `JdbcTemplate`;
- `@DataJdbcTest` — работает аналогично `@JdbcTest`, но дополнительно инициализирует репозитории Spring Data JDBC;
- `@DataR2dbcTest` — работает аналогично `@DataJdbcTest`, но инициализирует репозитории Spring Data R2DBC вместо репозитория Spring Data JDBC;
- `@DataMongoTest` — для тестов слоя, работающего с MongoDB;
- `@DataRedisTest` — для тестов слоя, использующих Redis;
- `@DataLdapTest` — для тестов слоя, использующих LDAP;

и другие.

При использовании аннотации `@DataJpaTest` подключать расширение `SpringExtension` через аннотацию `@ExtendWith` не нужно.

Полный код класса BaseDaoImplTest для проекта virtualpets-server-springboot приведен в листинге 14.50.

Листинг 14.50. BaseDaoImplTest.java

```
package ru.urvanov.virtualpets.server.dao;

import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase;
import org.springframework.boot.test.autoconfigure.jdbc.AutoConfigureTestDatabase.Replace;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import ru.urvanov.virtualpets.server.Application;
import ru.urvanov.virtualpets.server.config.ClockConfig;
```

```
import ru.urvanov.virtualpets.server.config.DaoTestConfig;
import ru.urvanov.virtualpets.server.config.DataSourceConfig;

/**
 * Базовый класс для тестов слоя DAO
 */
@Testcontainers
@ContextConfiguration(classes = {
        Application.class, DaoTestConfig.class, ClockConfig.class,
        DataSourceConfig.class})
@ActiveProfiles({"test", "test-dao"})
@DataJpaTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
class BaseDaoImplTest {

    /**
     * Управляет запуском и остановкой контейнера PostgreSQL.
     * При запуске нескольких тестов контейнер создается один раз
     * и переиспользуется последующими тестами.
     */
    @Container
    public static PostgreSQLContainer<?> postgreSQLContainer
            = new PostgreSQLContainer<>("postgres:16.1");
}
```

14.5.9. Класс *BookcaseDaoImplTest*

Тесты DAO-слоя наследуются от класса `BaseDaoImplTest` и тестируют методы классов как самые обычные JUnit-тесты. При этом им доступны аннотации внедрения бинов из контейнера Spring.

Пример теста класса `BookcaseDaoImpl` из проекта virtualpets-server-springframework приведен в листинге 14.51.

Листинг 14.51. BookcaseDaoImplTest.java

```
package ru.urvanov.virtualpets.server.dao;

import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;

import java.util.Optional;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.jdbc.Sql;

import ru.urvanov.virtualpets.server.dao.domain.Bookcase;

@Sql("/ru/urvanov/virtualpets/server/clean.sql")
```

```
class BookcaseDaoImplTest extends BaseDaoImplTest {

    @Autowired
    BookcaseDao bookcaseDao;

    @Test
    void testFind1() {
        Optional<Bookcase> bookcase = bookcaseDao.findById(1);
        assertTrue(bookcase.isPresent());
    }


    @Test
    void testFind2() {
        Optional<Bookcase> bookcase = bookcaseDao.findFullById(1);
        assertTrue(bookcase.isPresent());
        assertNotNull(bookcase.map(Bookcase::getBookcaseCosts).orElse(null));
    }

    @Test
    void testFind3() {
        Optional<Bookcase> bookcase = bookcaseDao.findById(-1);
        assertTrue(bookcase.isEmpty());
    }

    @Test
    void testFind4() {
        assertFalse(bookcaseDao.findFullById(-1).isPresent());
    }
}
```

14.5.10. Аннотация @Sql

Для очистки таблиц перед выполнением каждого теста служит аннотация `@Sql`. Она позволяет выполнять файлы скриптов SQL, указанные в атрибуте `scripts` или `value`, либо отдельные SQL-команды, прописанные в атрибуте `statements`.

Код класса `BookcaseDaoImplTest` (см. листинг 14.51) включает аннотацию `@Sql` над всем классом с указанием SQL-файла clean.sql, содержащего SQL-команду, очищающую таблицы перед тестом (листинг 14.53).

Листинг 14.53. Файл clean.sql

```
TRUNCATE TABLE
     "pet_book",
     "pet_achievement",
     "pet",
     "pet_cloth",
     "pet_journal_entry",
     "user",
     "chat",
```

```
    "pet_food",
    "pet_building_material",
    "pet_drink",
    "room"
    CASCADE;
```

Очистка используемых таблиц перед запускам каждого теста необходима, т. к. тесты должны быть независимы от результатов предыдущих тестов. Все тесты используют одно подключение к одной и той же базе PostgreSQL, запущенной в одном и том же контейнере Docker. Если не очищать таблицы перед запуском следующего теста, то данные, попавшие в таблицы в предыдущих тестах, приведут к искажению результата этого теста.

14.5.11. Библиотека assertj

Тесты DAO-слоя проекта virtualpets-server-springboot вместо методов `assert*` из JUnit используют библиотеку assertj, предоставляющую более выразительные методы проверки результатов тестов. Например, assertj содержит специальные методы для работы с коллекциями, с массивами, с итераторами, файлами, объектами, потоками, с Map, с исключениями, с Optional (листинг 14.54).

Листинг 14.54. BookcaseDaoImplTest.java

```
package ru.urvanov.virtualpets.server.dao;

import static org.assertj.core.api.Assertions.assertThat;

import java.util.Optional;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.jdbc.Sql;

import ru.urvanov.virtualpets.server.dao.domain.Bookcase;

@Sql("/ru/urvanov/virtualpets/server/clean.sql")
class BookcaseDaoImplTest extends BaseDaoImplTest {

    @Autowired
    BookcaseDao bookcaseDao;

    @Test
    void testFind1() {
        Optional<Bookcase> bookcase = bookcaseDao.findById(1);
        assertThat(bookcase).isPresent();
    }

    @Test
    void testFind2() {
        Optional<Bookcase> bookcase = bookcaseDao.findFullById(1);
```

```
        assertThat(bookcase).isPresent();
        assertThat(bookcase).map(Bookcase::getBookcaseCosts).isPresent();
    }

    @Test
    void testFind3() {
        Optional<Bookcase> bookcase = bookcaseDao.findById(-1);
        assertThat(bookcase).isEmpty();
    }

    @Test
    void testFind4() {
        Optional<Bookcase> bookcase = bookcaseDao.findFullById(-1);
        assertThat(bookcase).isEmpty();
    }
}
```

Примечание

Библиотеку assert можно использовать без Spring.

14.5.12. Фреймворк MockMvc

Тесты слоя DAO и модульные тесты слоя бизнес-логики проверяют лишь часть работы приложения, но они не дают никакой информации о том, что веб-приложение хоть как-то готово обрабатывать запросы.

Spring предоставляет возможность как полностью запускать приложение на фазе тестов и отправлять к нему HTTP-запросы, а затем сверять полученный результат с ожидаемым, так и использовать более облегченный вариант тестирования веб-приложения с помощью MockMvc.

Фреймворк Spring MVC Test, или MockMvc, предоставляет возможность тестирования приложения Spring MVC, эмулируя отправку запросов в приложения вместо запуска полноценного сервера.

14.5.13. Аннотация *@SpringBootTest*

Настройка тестов `MockMvc` в проектах virtualpets-server-springframework и virtualpets-server-springboot различается. Различие это связано с тем, что Spring Boot предоставляет аннотацию `@SpringBootTest`, осуществляющую практически всю необходимую настройку (листинг 14.55).

Листинг 14.55. BaseControllerTest.java

```
package ru.urvanov.virtualpets.server.controller.api;

import static
org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.springSecurity;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.extension.ExtendWith;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

/**
 * Базовый класс для тестов MockMvc слоя контроллеров.
 */
@ExtendWith(SpringExtension.class)
@Testcontainers
@SpringBootTest
@ActiveProfiles({"test", "test-mock-mvc"})
abstract class BaseControllerTest {

    /**
     * Конфигурация веб-приложения. С помощью
     * {@code wac.getServletContext()} осуществляется доступ к экземпляру
     * {@link jakarta.servlet.ServletContext}
     */
    @Autowired
    protected WebApplicationContext wac;

    /**
     * Методы наследников используют это поле для выполнения запросов:
     *  <pre>{@code
     *     mockMvc.perform(...
     * }</pre>
     */
    protected MockMvc mockMvc;

    /**
     * Управляет запуском и остановкой контейнера PostgreSQL.
     * При запуске нескольких тестов контейнер создается один раз
     * и переиспользуется последующими тестами.
     */
    @Container
    public static PostgreSQLContainer<?> postgreSQLContainer
            = new PostgreSQLContainer<>("postgres:16.1");

    @BeforeEach
    void setUp() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(wac)
                .apply(springSecurity())
                .build();
```

14.5.14. Настройка *MockMvc* для Spring Framework

Аналогичная настройка тестов MockMvc для проекта virtualpets-server-springframework выполняется несколько сложнее, т. к. необходимо указывать файлы XML-конфигураций самостоятельно, прописывая их в аннотации @ContextHierarchy, но результат будет точно таким же (листинг 14.56).

Листинг 14.56. BaseControllerTest.java

```
package ru.urvanov.virtualpets.server.controller.api;

import static org.springframework.security.test.web.servlet.setup.
                                                    SecurityMockMvcConfigurers.springSecurity;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ActiveProfiles;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.ContextHierarchy;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.context.web.WebAppConfiguration;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.context.WebApplicationContext;
import org.testcontainers.containers.PostgreSQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;

import ru.urvanov.virtualpets.server.controller.config.ClockConfig;
import ru.urvanov.virtualpets.server.controller.config.DataSourceConfig;

/**
 * Базовый класс для тестов MockMvc слоя контроллеров.
 */
@ExtendWith(SpringExtension.class)
@Testcontainers
@ActiveProfiles({"test", "test-mock-mvc"})
@WebAppConfiguration
@ContextHierarchy({
        @ContextConfiguration(classes = { ClockConfig.class, DataSourceConfig.class }),
        @ContextConfiguration(locations = {
                "file:src/main/webapp/WEB-INF/spring/root-context.xml",
                "file:src/main/webapp/WEB-INF/spring/security.xml",
                """
                file:src/main/webapp/WEB-INF/spring/appServlet\
                /servlet-context.xml""" }) })
abstract class BaseControllerTest {

    /**
     * Конфигурация веб-приложения. С помощью
     * {@code wac.getServletContext()} осуществляется доступ к экземпляру
     * {@link jakarta.servlet.ServletContext}
     */
```

```
    @Autowired
    protected WebApplicationContext wac;

    /**
     * Методы наследников используют это поле для выполнения запросов:
     *  <pre>{@code
     *     mockMvc.perform(...
     * }</pre>
     */
    protected MockMvc mockMvc;

    /**
     * Управляет запуском и остановкой контейнера PostgreSQL.
     * При запуске нескольких тестов контейнер создается один раз
     * и переиспользуется последующими тестами.
     */
    @Container
    public static PostgreSQLContainer<?> postgreSQLContainer
            = new PostgreSQLContainer<>("postgres:16.1");

    @BeforeEach
    void setUp() {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(wac)
                .apply(springSecurity())
                .build();
    }
}
```

Реальные тесты `MockMvc` **наследуются от класса** `BaseControllerTest` **и используют поле** `mockMvc` **для выполнения запросов. Экземпляр** `MockMvc` **создается и настраивается в методе** `setUp`**, помеченном аннотацией** `@BeforeEach`**, а значит, выполняющемся перед запуском каждого теста. Метод** `MockMvcBuilders#webAppContextSetup` **создает экземпляр** `MockMvc`**, а метод** `apply(springSecurity())` **настраивает в** `MockMvc` **поддержку Spring Security.**

14.5.15. Простой тест *MockMvc*

Запросы `MockMvc` **осуществляются с помощью метода** `perform`**, в котором приводится адрес запроса, затем указываются параметры запроса, тело запроса, заголовки и т. д., а с помощью методов** `andExpect` **проверяется результат выполнения запроса (листинг 14.57).**

Листинг 14.57. PublicControllerTest.java

```
package ru.urvanov.virtualpets.server.controller.api;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
```

```
import org.junit.jupiter.api.Test;
import org.springframework.http.MediaType;

/**
 * Базовый пример теста MockMvc
 */
class PublicControllerTest extends BaseControllerTest {

    @Test
    void getServerTechnicalInfo() throws Exception {
        mockMvc.perform(get(
                "/api/v1/PublicService/serverTechnicalInfo")
                .contentType(MediaType.APPLICATION_JSON)
                .accept(MediaType.APPLICATION_JSON))
                .andExpectAll(status().isOk(), content()
                        .contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.info").isMap())
                .andExpect(jsonPath("$.info['java.version']")
                        .isString());
        ;
    }
}
```

14.5.16. Аннотация @MockBean

Spring Boot дополнительно добавляет аннотацию `@MockBean`, позволяющую создавать mock-бины, а затем настраивать возвращаемые ими значения без выполнения реальных вызовов. Пример из проекта virtualpets-server-springboot приведен в листинге 14.58.

Листинг 14.58. PublicControllerTest.java

```
package ru.urvanov.virtualpets.server.controller.api;
import static org.mockito.Mockito.when;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import java.util.Map;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.mock.mockito.MockBean;

import org.springframework.http.MediaType;

import ru.urvanov.virtualpets.server.controller.api.domain.ServerTechnicalInfo;
import ru.urvanov.virtualpets.server.service.PublicApiService;
```

```
/**
 * Базовый пример теста MockMvc
 */
class PublicControllerTest extends BaseControllerTest {

    @MockBean
    private PublicApiService publicApiService;

    @Test
    void getServerTechnicalInfo() throws Exception {
        // Подготовка данных
        Map<String, String> info
                = Map.of("java.version", "some_version");
        ServerTechnicalInfo expected = new ServerTechnicalInfo(info );

        // Настройка моков
        when(publicApiService.getServerTechnicalInfo())
                .thenReturn(expected);

        // Выполнение запроса и проверка результата
        mockMvc.perform(get(
                "/api/v1/PublicService/serverTechnicalInfo")
                .contentType(MediaType.APPLICATION_JSON)
                .accept(MediaType.APPLICATION_JSON))
                .andExpectAll(status().isOk(), content()
                        .contentType(MediaType.APPLICATION_JSON))
                .andExpect(jsonPath("$.info").isMap())
                .andExpect(jsonPath("$.info['java.version']")
                        .isString());
    }
}
```

14.5.17. Интеграция *MockMvc* и Spring Security

Тест `PublicControllerTest` тестирует публичные методы API, доступные всем пользователям, включая анонимных. Тест `MockMvc` позволяет тестировать не только подобные методы, но и методы, доступные лишь после успешной аутентификации. Для этого используется метод `with(user(authentication_principal))`, где

- `with` — это статический метод класса `MockHttpServletRequestBuilder`, передаваемого в `perform`;
- `user` — это статический метод класса `SecurityMockMvcRequestPostProcessors`.

Перед использованием метода `with(user(authentication_principal))` необходимо настроить в `MockMvc` поддержку Spring Security с помощью `apply(springSecurity())` (листинг 14.59).

Листинг 14.59. BaseControllerTest.java

```
@BeforeEach
void setUp() {
    this.mockMvc = MockMvcBuilders.webAppContextSetup(wac)
            .apply(springSecurity())
            .build();
}
```

Пример тестов, осуществляющих запросы MockMvc со специально созданным для этого пользователем, приведен в листинге 14.60.

Листинг 14.60. PetControllerTest.java

```
package ru.urvanov.virtualpets.server.controller.api;

import static
org.springframework.security.test.web.servlet.request.SecurityMockMvcRequestPostProcessors.user;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

import java.util.List;

import org.junit.jupiter.api.Test;
import org.springframework.http.MediaType;
import org.springframework.mock.web.MockHttpSession;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.test.context.jdbc.Sql;

import ru.urvanov.virtualpets.server.auth.UserDetailsImpl;

/**
 * <p>
 * Тест для {@link PetController}.
 * </p>
 * <p>
 * Пример теста с использованием сессии и подстановкой пользователя,
 * от лица которого выполняется запрос.
 * </p>
 */
class PetControllerTest extends BaseControllerTest {

    private UserDetailsImpl userDetailsImpl = new UserDetailsImpl(1, "test", "Tester",
            "123",
            true,
            List.of(new SimpleGrantedAuthority("ROLE_USER")));

    @Test
    @Sql({"/ru/urvanov/virtualpets/server/clean.sql",
            "PetControllerTest.sql"})
```

```
void getUserPets() throws Exception {
    mockMvc.perform(get("/api/v1/PetService/getUserPets")
            .contentType(MediaType.APPLICATION_JSON)
            .accept(MediaType.APPLICATION_JSON)
            .with(user(userDetailsImpl)))
            .andExpectAll(status().isOk(), content()
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(jsonPath("$.petsInfo").isArray());
}

@Test
@Sql(
        scripts = {"/ru/urvanov/virtualpets/server/clean.sql",
                "PetControllerTest.sql"},
        statements = """
        insert into\
            virtualpets_server_springframework.pet_food(\
                pet_id, food_id, food_count)\
            values (1, 'DRY_FOOD', 10)""")
void eat() throws Exception {
    MockHttpSession session = new MockHttpSession(
            wac.getServletContext(),
            "test");
    selectPet(session);
    mockMvc.perform(post("/api/v1/PetService/satiety")
            .contentType(MediaType.APPLICATION_JSON)
            .accept(MediaType.APPLICATION_JSON)
            .content("""
                    {
                        "foodId": "DRY_FOOD"
                    }""")
            .with(user(userDetailsImpl))
            .session(session)
            )
            .andExpectAll(status().isNoContent());
    ;
}

private void selectPet(MockHttpSession session) throws Exception {
    mockMvc.perform(post("/api/v1/PetService/select")
            .contentType(MediaType.APPLICATION_JSON)
            .accept(MediaType.APPLICATION_JSON)
            .content("""
                    {
                        "petId": 1
                    }""")
            .with(user(userDetailsImpl))
            .session(session)
            )
            .andExpectAll(status().isNoContent());
```

14.6. Резюме

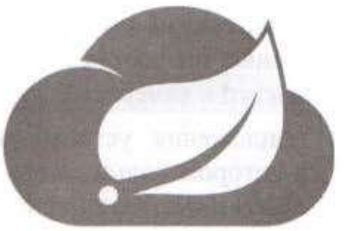
JUnit и TestNG — два наиболее популярных фреймворка для создания модульных тестов. Новые проекты в основном используют JUnit. Spring интегрируется как с JUnit, так и с TestNG.

Библиотека Testcontainers позволяет поднимать контейнер Docker с требуемой базой данных перед запуском тестов и останавливать его по окончании теста.

Фреймворк MockMvc предоставляет возможность тестировать не только внутренние методы приложения, но и полную обработку запросов контроллерами Spring MVC.

Создание тестов в Spring Framework и Spring Boot выполняется похоже, но Spring Boot предоставляет дополнительные аннотации, значительно облегчающие настройку контекста под конкретный тест.

ГЛАВА 15



Клиентское приложение

15.1. Проект Spring Mobile

Spring Mobile — это древний проект Spring, созданный для облегчения разработки мобильных веб-приложений.

Spring Mobile предоставлял следующие возможности:

- определение устройства на сервере;
- управление предпочитаемым типом сайта: мобильный, десктопный или PDA;
- переключение сайтов между мобильным, десктопным и PDA;
- управление представлениями для различных устройств.

Spring Mobile больше не поддерживается, поэтому изучать и описывать его не имеет смысла.

15.2. Приложение Progressive Web Application

Progressive Web Application (PWA) — это приложение, построенное на веб-технологиях, но предоставляющее возможности работы, аналогичные нативному приложению.

Отличия нативных приложений от веб-сайтов:

- нативные приложения (Android, iOS и т. п.) обычно создаются с помощью набора инструментов SDK (Software Development Kit), предоставляемого разработчиком платформы, и распространяются с помощью магазина приложений платформы, в котором пользователи могут найти приложение и установить его;
- нативные приложения запускаются с помощью значка на устройстве — в отличие от веб-сайта, который пользователь должен посетить с помощью браузера;
- приложения могут работать без доступа к сети — в отличие от сайтов, которые можно посещать только при наличии доступа в Интернет;
- приложения могут выполнять фоновые действия, даже когда они не запущены;

- приложения интегрируются с операционной системой, могут отправлять сообщения, предоставляют возможность выбора фотографии из фотоальбома, имеют доступ к камере, GPS и прочим устройствам;
- приложения устанавливаются из централизованного магазина приложений, в котором пользователь может найти их и установить, а также получают обновления из магазина приложений.

PWA представляют собой некоторую комбинацию возможностей обычного веб-сайта и нативных приложений:

- PWA разрабатываются с помощью обычных веб-технологий. Фактически они представляют собой веб-сайт, поэтому могут запускаться на множестве операционных систем и множестве устройств без перекомпиляции;
- PWA может быть установлено на устройство, при этом оно получает значок для его запуска на нем;
- PWA может работать в фоне и без доступа к Интернету;
- PWA может использовать полноэкранный режим, а не только запускаться во вкладке браузера;
- PWA может интегрироваться с устройством и получать доступ к его возможностям;
- PWA может распространяться через магазины приложений.

Клиент игры с виртуальными питомцами, рассматриваемой в качестве примера в этой книге, представляет собой PWA. Исходники клиента доступны:

- в GitHub[1];
- в специальном разделе на сайте автора книги[2].

Далеко не каждый веб-сайт является Progressive Web Application. Чтобы он им стал, необходимо выполнение следующих условий:

- в блоке `HEAD` страницы должен быть указан файл манифеста;
- в файле манифеста должны быть указаны как минимум: `name`, `short_name`, `display`, `description`, значок;
- в коде приложения должен быть описан `Service Worker`, который обрабатывает событие `install` и кеширует все ресурсы приложения;
- приложение должно заинтересовать посетителя — ему предоставляется возможность провести на странице браузера с приложением какое-то время, «потыкать» экранные элементы и т. д.;
- сайт должен работать по HTTPS.

При выполнении этих условий в объект `window` придет событие `beforeinstallprompt`.

---

[1] См. **https://github.com/urvanov-ru/virtualpets-client-js**.

[2] См. **https://urvanov.ru/книги/spring-book-2024/**.

Файл манифеста клиента виртуальных питомцев приведен в листинге 15.1.

Листинг 15.1. Файл manifest.json

```
{
  "background_color": "purple",
  "description": "Urvanov's Pretty Virtual Pets JavaScript Client",
  "display" : "fullscreen",
  "orientation": "landscape",
  "icons": [
    {
      "src": "data/images/cat/cat1.png",
      "sizes": "150x150"
    }
  ],
  "name": "Urvanov's Pretty Virtual Pets",
  "short_name": "VirtualPets",
  "start_url": "index.html"
}
```

15.3. Работающая игра

HTML-страница со ссылками, изображенная на рис. 15.1, расположена по основному адресу приложения[1]. Ссылки эти ведут соответственно на клиентскую часть игры[2], написанную на JavaScript, и на публичный сайт серверной части игры[3].

Клиент на JavaScript начинается с экрана выбора языка (рис. 15.2) — всего в игре поддерживаются два языка: английский и русский.

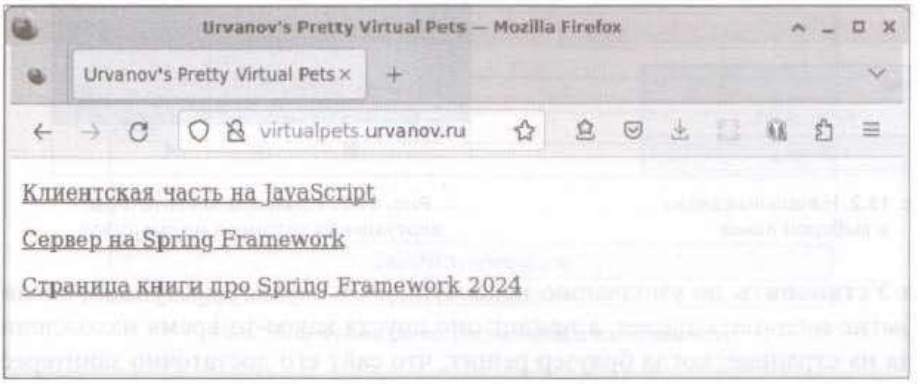


Рис. 15.1. Страница, расположенная по адресу **http://virtualpets.urvanov.ru**

[1] См. **http://virtualpets.urvanov.ru**.

[2] См. **http://virtualpets.urvanov.ru/virtualpets-client-js/**.

[3] См. **http://virtualpets.urvanov.ru/virtualpets-server-springframework/site/home**.

Язык по умолчанию определяется на основе предпочитаемого языка посетителя, установленного в браузере. Выбранный посетителем язык сохраняется в localStorage браузера, так что при следующем посещении игры выбранный язык будет принят по умолчанию.

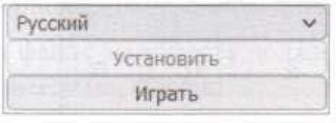


Рис. 15.2. Начальный экран с выбором языка

Рис. 15.3. Установка клиента игры виртуальных питомцев на смартфон

Кнопка **Установить** по умолчанию недоступна. Она станет доступной, когда придет событие `beforeinstallprompt`, а придет оно спустя какое-то время нахождения посетителя на странице, когда браузер решит, что сайт его достаточно заинтересовал. Возможно, ускорить принятие браузером такого решения может помочь выбор несколько раз языка из выпадающего списка. Пример диалогового окна, открывающегося по нажатию на кнопку **Установить**, показан на рис. 15.3.

Не все браузеры поддерживают установку приложений, но браузеры современных смартфонов обычно генерируют событие `beforeinstallprompt`, а значит, позволяют установить приложение на устройство.

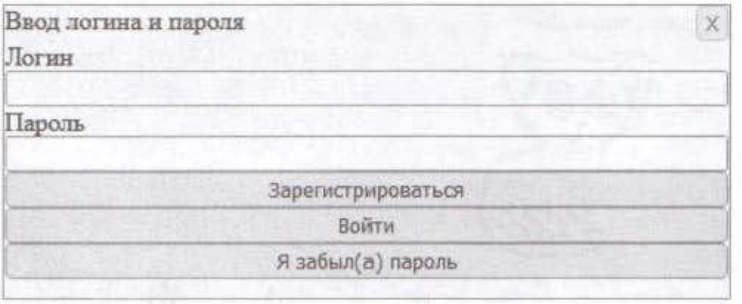


Рис. 15.4. Форма ввода логина и пароля

По нажатию кнопки **Играть** отображается форма ввода логина и пароля, показанная на рис. 15.4.

Экранная форма ввода логина и пароля работает аналогично подобным формам на всех сайтах и во всех играх:

1. Посетитель вводит логин.
2. Посетитель вводит пароль.
3. Посетитель щелкает на кнопке **Войти**.

По щелчку на кнопке **Войти** JavaScript-клиент отправляет в бэкенд запрос, обрабатываемым методом `PublicController#login`.

Если посетитель еще не имеет учетной записи, то он может создать новую с помощью экранной формы, открывающейся по щелчку на кнопке **Зарегистрироваться**. Внешний вид формы регистрации показан на рис. 15.5.

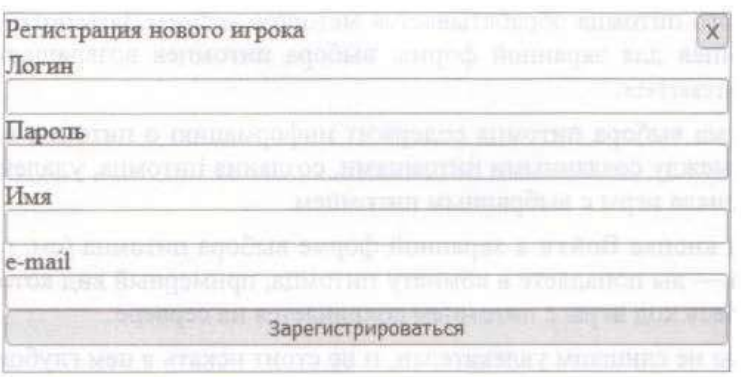


Рис. 15.5. Форма регистрации нового пользователя

Запросы регистрации нового пользователя обрабатываются методом `PublicController#register`.

При успешном прохождении процедуры аутентификации на экранной форме, показанной на рис. 15.4, игрок попадает на экранную форму выбора питомца (рис. 15.6) либо на экранную форму создания питомца (рис. 15.7), если у игрока нет ни одного питомца.

Рис. 15.6. Экранная форма выбора питомца

Рис. 15.7. Экранная форма создания питомца

Создание нового питомца обрабатывается методом `PetController#create`. Список доступных питомцев для экранной формы выбора питомцев возвращается методом `PetController#getUserPets`.

Экранная форма выбора питомца содержит информацию о питомце и ряд кнопок для перехода между созданными питомцами, создания питомца, удаления питомца, а также для начала игры с выбранным питомцем.

По щелчку на кнопке **Войти** в экранной форме выбора питомца (см. рис. 15.6) начинается игра — вы попадаете в комнату питомца, примерный вид которой показан на рис. 15.8. Весь ход игры с питомцем сохраняется на сервере.

Геймплей игры не слишком увлекателен, и не стоит искать в нем глубокого смысла. Основная цель игры: показать пример приложения, разработанного на Spring, и создать полигон для демонстрации основных технологий и базовых возможностей.

Исходный код клиента на JavaScript в рамках этой книги не рассматривается, т. к. книга посвящена Spring. Для особо заинтересовавшихся отмечу, что он представляет собой переписанный вариант старого клиента на Swing + Spring Framework + Spring HttpInvoker.

Spring Framework — это не только технология для бэкенда. Как уже отмечалось ранее, вполне допускается разработка с его помощью и десктопных приложений.

Рис. 15.8. Комната питомца

Для знакомых с языком программирования JavaScript стиль и архитектура клиентской части могут показаться очень странными. Всё это из-за того, что архитектура осталась прежней — той, которая была в древнем клиенте на Java.

Заключение

Вот и закончилась книга. Содержимое ее достаточно сложное. Вряд ли можно осознать его за одно прочтение. Если вы только изучаете Spring, то вам, вероятно, стоит переосмыслить прочитанное, сделать небольшую паузу, а затем прочитать книгу еще раз. Такое повторное прочтение поможет не просто освежить материал книги в памяти, но и понять его на более глубоком уровне.

При создании книги и подготовке материала для нее было урезано и упрощено множество моментов. Не стоит читать ее разделы как справочный материал. Экосистема Spring очень огромная и сложная, поэтому даже простая книга о ней для начинающего будет сложной. В книге сделана попытка провести читателя через основные этапы изучения Spring и сделать упор на базовые знания. Информация, которая слишком сложна для начального освоения, в большинстве случаев просто убрана как ненужная и запутывающая. Базовая же информация представлена в максимальном объеме.

Несмотря на то что книга представляет собой именно учебник, понимание ее содержимого требует знания определенных технологий: HTTP, JSON, XML, YAML, хорошего знания Java, HTML, CSS, JavaScript, SQL, необходимо также иметь навыки работы с выбранной IDE и с Git и много чего знать из мира информационных технологий вообще.

Внутри каждой главы разделы и подразделы продолжают мысли друг друга, т. к. описывают использование фреймворка Spring на специальном тестовом приложении. Для понимания каждого раздела в большинстве случаев необходимо предварительно прочитать предыдущие.

Если вы в поиске работы, то ваши дальнейшие шаги после изучения материала книги могут быть следующими:

1. Попытаться попасть на два-три собеседования.
2. Запомнить вопросы, которые задавались на собеседовании.
3. Найти материал по этим вопросам.
4. Изучить материал.

5. Повторять предыдущие пункты до тех пор, пока поиск работы не увенчается успехом.

Если вы уже работаете, а книгу читали для углубления знаний, то:

- посмотрите, какие технологии используются на вашем текущем месте работы. Имеет смысл почитать книги и документацию по ним в первую очередь;
- подумайте о том, что может быть полезно при следующем поиске работы. Изучите вакансии на рынке труда, посмотрите видео с тематических конференций и т. п. Попробуйте изучить что-нибудь из того, что найдете и что покажется вам интересным.

Предметный указатель

В

Г, З

И

К

Л

М

Н

О

П

Р

С

Т

Ф

Ш

Ю, Я

Spring и Spring Boot

Разработка облачных приложений на Java

Внедрение зависимостей на Java и взаимодействие с облачным сервером

Перед вами книга о фреймворке Spring и его популярном подпроекте Spring Boot для работы с облачными приложениями. Spring активно использует внедрение зависимостей и аннотации Java, применяется при управлении крупными серверными приложениями. В Spring и Spring Boot реализован необходимый минимум команд для настройки и обновления конфигурации приложений на Java, предоставляется встроенный веб-сервер и экосистема для внедрения зависимостей.

Основное назначение описываемых технологий — обслуживание микросервисов на Java и координация их взаимодействий.

Все примеры в книге рассмотрены на материале сквозного проекта, представляющего собой простую игру и иллюстрирующего основные приемы взаимодействия с сервером, обновления данных и поддержания их актуальности.

Федор Урванов — Java-программист с более чем 15-летним стажем, обладатель сертификатов 1Z0-803 (Oracle Certified Associate, Java SE 7 Programmer) и 1Z0-804 (Oracle Certified Professional, Java SE 7 Programmer). Автор блога https://urvanov.ru и книги «Java. Состояние языка и его перспективы». Участвовал в разработке программного обеспечения для нескольких крупнейших банков России.

ISBN 978-5-9775-2049-2

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru