[Spring in Action Covers Spring 5.0 перевод на русский. Глава 1](https://beginner-mercenary.blogspot.com/2019/04/spring-in-action-covers-spring-50-1.html)

**Начало работы со Spring**

Хотя греческий философ Гераклит не был хорошо известен как разработчик программного обеспечения, он, казалось, хорошо разбирался в этом вопросе. Его цитируют так: "единственное неизменное-это перемены.” Это заявление отражает основополагающую истину разработки программного обеспечения.

То, как мы разрабатываем приложения сегодня, отличается от того, что было год назад, 5 лет назад, 10 лет назад, и, конечно, 15 лет назад, когда первоначальная структура Spring Framework была представлена в книге рода Джонсона, Expert One-on-One J2EE Design and Development (Wrox, 2002, http://mng.bz/oVjy).

Тогда наиболее распространенными типами разработанных приложений были браузерные веб-приложения, поддерживаемые реляционными базами данных. Хотя этот тип разработки по-прежнему актуален, и Spring хорошо подходит для таких приложений, мы также заинтересованы в разработке приложений, состоящих из микросервисов, предназначенных для облака, которые сохраняют данные в различных базах данных. И новый интерес к реактивному программированию направлен на обеспечение большей масштабируемости и повышение производительности с неблокирующими операциями.

По мере развития разработки программного обеспечения структура Spring также претерпевала изменения, направленные на решение современных задач в области разработки, включая микрослужбы и реактивное программирование. Spring также намеревался упростить свою собственную модель разработки, введя Spring Boot.

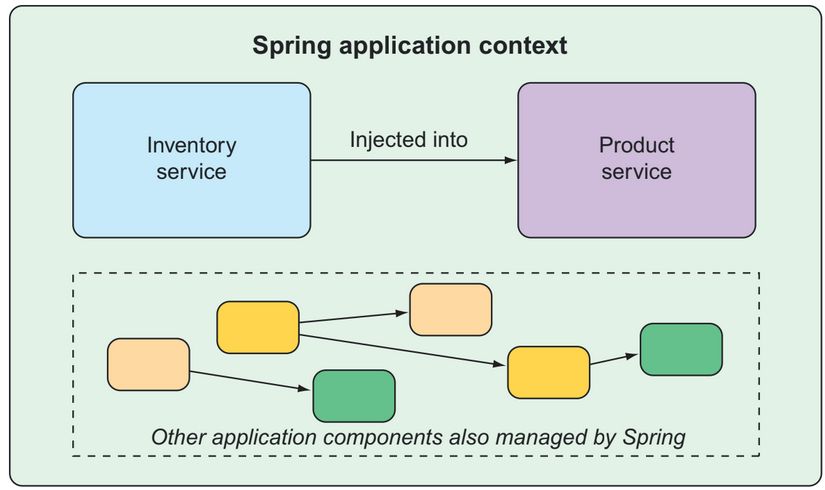
Независимо от того, разрабатываете ли вы простое веб-приложение с поддержкой баз данных или создаете современное приложение, основанное на микросервисах, Spring-это платформа, которая поможет вам достичь ваших целей. Эта глава - ваш первый шаг в путешествии по современной разработке приложений с Spring.

**1.1 Что такое Spring?**

Я знаю, что вам, вероятно, не терпится начать писать приложение Spring, и я уверяю вас, что до конца этой главы вы разработаете простое приложение. Но сначала, позвольте мне подготовить почву с несколькими основными понятиями Spring, которые помогут вам понять, что делает Spring.

Любое нетривиальное приложение состоит из множества компонентов, каждый из которых отвечает за свою часть общей функциональности приложения, координируя работу с другими элементами приложения. При запуске приложения эти компоненты каким-то образом должны быть созданы и представлены друг другу.

По своей сути Spring предлагает *контейнер*, часто называемый контекстом приложения Spring (Spring application context), который создает компоненты приложения и управляет ими. Эти компоненты, или *beans,*тесно связанные друг с другом внутри Spring application context, как кирпичи, ступени, трубопровод, проводка все вместе составляя дом.  
Способ соединения bean-ов вместе основан на шаблоне, известном как dependency injection (DI). Вместо того, чтобы компоненты создавали и поддерживали жизненный цикл других компонентов, от которых они зависят, приложение с зависимостями полагается на отдельную сущность (контейнер) для создания и обслуживания всех компонентов и внедрения их в компоненты, которые в них нуждаются. Обычно это делается с помощью аргументов конструктора или методов доступа к свойствам.  
Например, предположим, что среди множества компонентов приложения есть два, к которым вы будете обращаться: служба запасов (для получения уровней запасов) и Служба продуктов (для предоставления базовой информации о продукте). Служба продуктов зависит от службы запасов, чтобы иметь возможность предоставить полный набор сведений о продуктах. На рисунке 1.1 показаны связи между этими компонентами и контекстом приложения Spring.



Помимо основного контейнера, Spring и полный портфель связанных библиотек предлагают веб-платформу, различные варианты сохранения данных, инфраструктуру безопасности, интеграцию с другими системами, мониторинг времени выполнения, поддержку микрослужб, модель реактивного программирования и многие другие функции, необходимые для современной разработки приложений.  
Исторически сложилось так, что контекст приложения Spring для связывания компонентов был связан с одним или несколькими XML-файлами, описывающими компоненты и их связь с другими компонентами. Например, следующий XML-код объявляет два bean-а InventoryService bean и ProductService bean, и привязку *InventoryService*bean в  *ProductService* через аргументы конструктора:

<bean id="inventoryService" class="com.example.InventoryService" />   
<bean id="productService" class="com.example.ProductService" />  
            <constructor-arg ref="inventoryService" />  
</bean>  
  
Однако в последних версиях Spring чаще используется конфигурация на основе Java. Следующий класс конфигурации на основе Java эквивалентен конфигурации XML:  
  
@Configuration  
public class ServiceConfiguration {  
   @Bean  
   public InventoryService inventoryService() {  
      return new InventoryService();  
   }  
  
   @Bean  
   public ProductService productService() {  
      return new ProductService(inventoryService());  
   }  
  
}

Аннотация *@Configuration* указывает Spring, что это класс конфигурации, который будет предоставлять bean-ы контексту приложения Spring. Методы класса конфигурации аннотируются *@Bean*, указывая, что объекты, которые они возвращают, должны быть добавлены как bean в контексте приложения (по умолчанию, их bean идентификаторы будут соответствовать именам методов, которые определяют их).  
Конфигурация Java-based предлагает несколько преимуществ по сравнению с xml-конфигурацией, включая повышение безопасности и улучшение refactorability. Но даже при всем при этом явная настройка с помощью Java или XML необходима только в том случае, если Spring не может автоматически настроить компоненты.  
Автоматическая конфигурация имеет свои корни в методах Spring известных как *autowiring*и *component scanning*. С помощью component scanning Spring может автоматически обнаруживать компоненты из classpath приложения и создавать их как bean-ы в контексте приложения Spring. С autowiring Spring автоматически объединяет компоненты с другими компонентами, от которых они зависят.  
Совсем недавно, с введением Spring Boot, автоматическая настройка вышла далеко за рамки сканирования компонентов и *autowiring*. Spring Boot-это расширение Spring Framework, которое предлагает несколько улучшений производительности. Наиболее известным из этих усовершенствований является автоконфигурация, где Spring Boot может сделать разумные предположения о том, какие компоненты должны быть настроены и подключены друг к другу, на основе записей в classpath, переменных среды и других факторов.  
Я хотел бы показать вам пример кода, демонстрирующего автоконфигурацию. Но я не могу. Видите ли, автоконфигурация очень похожа на ветер. Вы можете видеть последствия этого, но нет никакого кода, который я могу показать вам и сказать: "Смотрите! Вот пример автоконфигурации!” Такое случается, компоненты и функциональность без написания кода. Именно этот отсутствие кода имеет важное значение для автоконфигурации и делает ее такой замечательной.  
Автоконфигурация Spring Boot значительно сократила объем явной конфигурации (будь то XML или Java), необходимой для построения приложения. Фактически, к тому времени, когда вы закончите пример в этой главе, у вас будет работающее приложение Spring, которое имеет только одну строку кода конфигурации Spring!  
Spring Boot настолько улучшает Spring-разработку, что без него сложно представить разработку Spring-приложений. По этой причине в этой книге Spring и Spring Boot рассматриваются как одно и то же. Мы будем использовать Spring Boot, насколько это возможно, и явную конфигурацию только при необходимости. И, поскольку Spring XML configuration-это олдскульный способ работы с Spring, мы сосредоточимся в первую очередь на Java-конфигурации Spring.  
Но хватит болтать, yakety-yak, и флудить. Название этой книги включает в себя фразу в действии, так что давайте двигаться, и вы можете начать писать свое первое приложение с Spring.

**1.2  Инициализация приложения Spring**

Благодаря ходу этой книги вы создадите Taco Cloud, онлайн-приложение для заказа самой замечательной еды, созданной человеком-такосом. Конечно, для достижения этой цели вы будете использовать Spring, Spring Boot и множество связанных библиотек и платформ.  
Вы найдете несколько вариантов инициализации приложения Spring. Хотя я мог бы провести вас через шаги ручного создания структуры каталогов проекта и определения спецификации сборки, это потерянное время—время, которое лучше потратить на написание кода приложения. Таким образом вы собираетесь опереться на Spring Initializr для начальной загрузки приложения.  
Spring Initializr-это веб-приложение на основе браузера и REST API, которое может создавать структуру проекта skeleton Spring, которую вы можете реализовать с любой функциональностью, которую вы хотите. Несколько способов использования Spring Initializr:

Из веб-приложения на http://start.spring.io

Из командной строки с помощью команды curl

Из командной строки с помощью интерфейса командной строки Spring Boot

При создании нового проекта с Spring Tool Suite

При создании нового проекта с IntelliJ IDEA

При создании нового проекта в NetBeans

Вместо того, чтобы тратить несколько страниц этой главы, рассказывая о каждом из этих вариантов, я собрал эти детали в приложении. В этой главе и на протяжении всей этой книги я покажу вам, как создать новый проект, используя мой любимый вариант: поддержка Spring Initializr в Spring Tool Suite.  
Как следует из названия, Spring Tool Suite-это фантастическая среда разработки Spring. Но он также предлагает удобную функцию Spring Boot Dashboard, которая (по крайней мере, на момент написания этой статьи) недоступна ни в одном из других вариантов IDE.  
Если вы не являетесь пользователем Spring Tool Suite, это нормально; мы все еще можем быть друзьями. Перейдите к приложению и замените наиболее подходящий вариант инициализации инструкциями в следующих разделах. Но знайте, что в этой книге я могу время от времени ссылаться на функции, специфичные для Spring Tool Suite, такие как Spring Boot Dashboard. Если вы не используете Spring Tool Suite, вам нужно будет адаптировать эти инструкции чтобы соответствовать вашей идее.

**1.2.1 Инициализация Spring проекта с Spring Tool Suite**

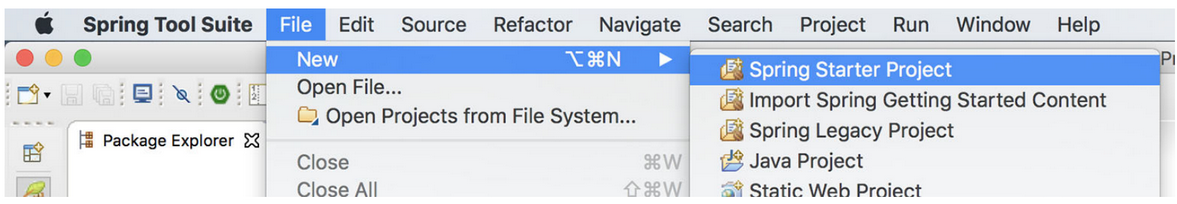
Чтобы начать работу с новым проектом Spring в Spring Tool Suite, перейдите в меню File и выберите New, а затем Spring Starter Project. На рисунке 1.2 показана структура меню.  


Рисунок 1.2 запуск нового проекта с Инициализатором в Spring Tool Suite

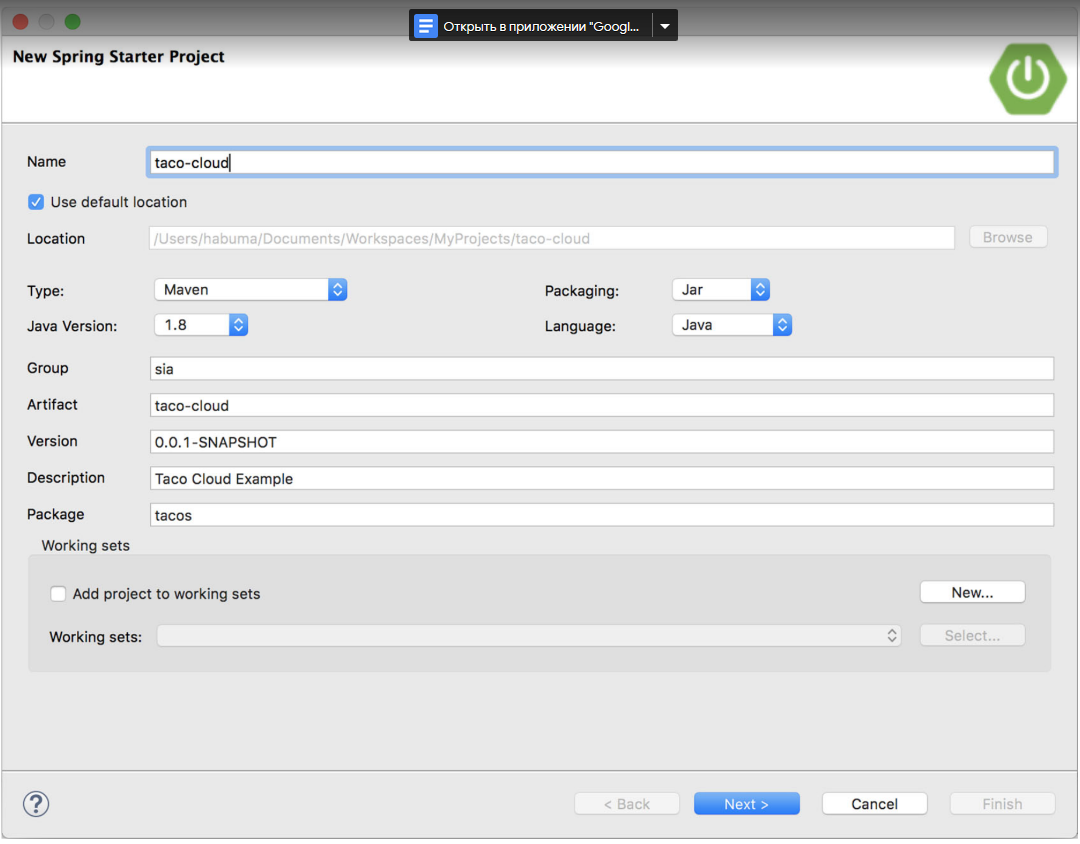
После выбора Spring Starter Project появится диалоговое окно мастера создания проекта (рис. 1.3). На первой странице мастера запрашиваются общие сведения о проекте, такие как имя проекта, описание и другие важные сведения. Если вы знакомы с содержанием Maven pom.xml-файл, вы будете распознавать большинство полей как элементы, которые в конечном итоге в спецификации сборки Maven. Для приложения Taco Cloud заполните диалоговое окно, как показано на рисунке 1.3, и нажмите Next.  


Рис. 1.3 указания общей информации о проекте для приложения Taco Cloud

Следующая страница мастера позволяет выбрать зависимости для добавления в проект (см. рисунок 1.4). Обратите внимание, что в верхней части диалогового окна можно выбрать версию Spring Boot, на которой будет основываться проект. По умолчанию используется самая последняя доступная версия. Это, как правило, хороший вариант, если вам не нужно ориентироваться на другую версию.  
Что касается самих зависимостей, вы можете либо развернуть различные разделы и искать нужные зависимости вручную, либо искать их в поле поиска в верхней части списка доступных. Для приложения Taco Cloud установите зависимости, показанных на рисунке 1.4.

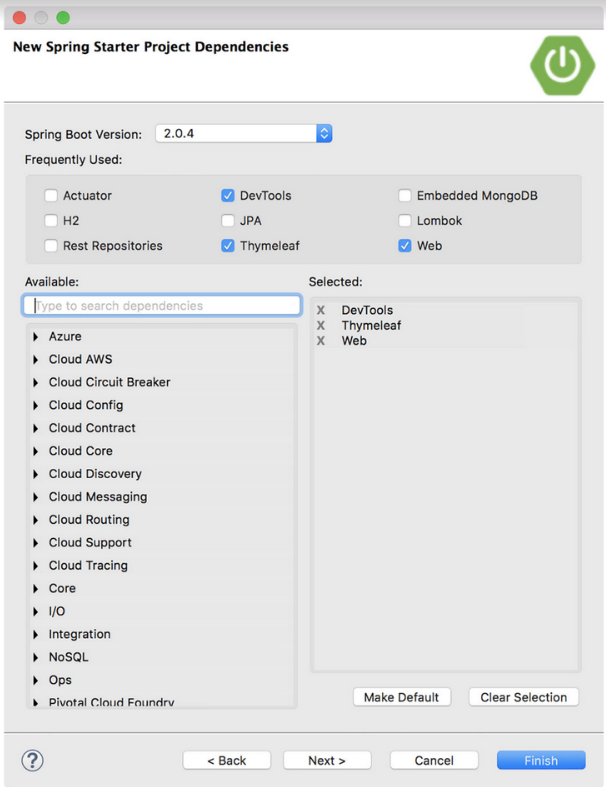


Рисунок 1.4 выбор первоначальных зависимостей

На этом этапе можно нажать кнопку Finish, чтобы создать проект и добавить его в рабочую область. Но если вы чувствуете себя более уверенным, нажмите кнопку Next еще раз, чтобы увидеть последнюю страницу мастера создания начального проекта, как показано на рисунке 1.5.

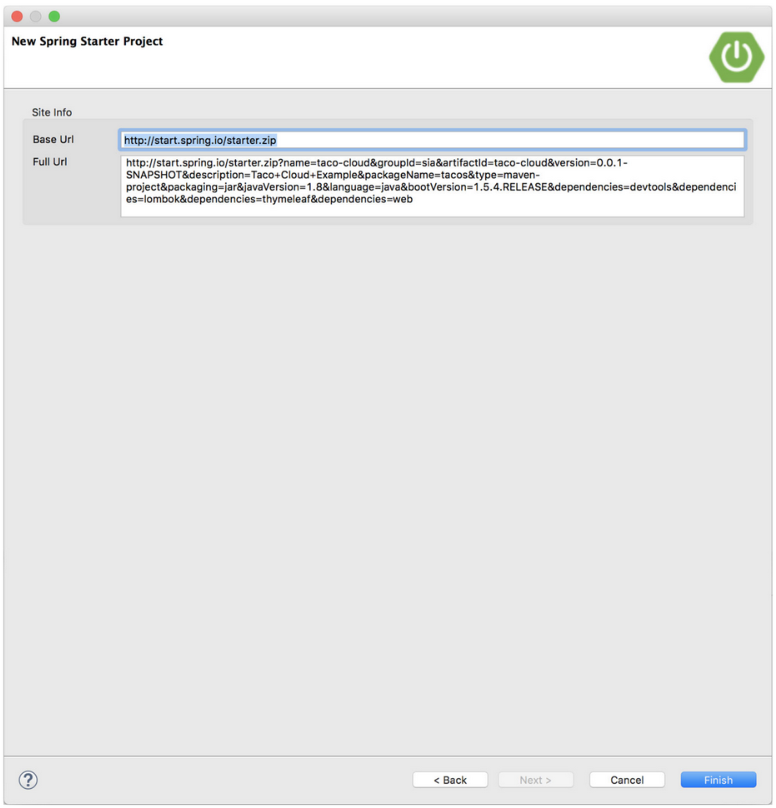
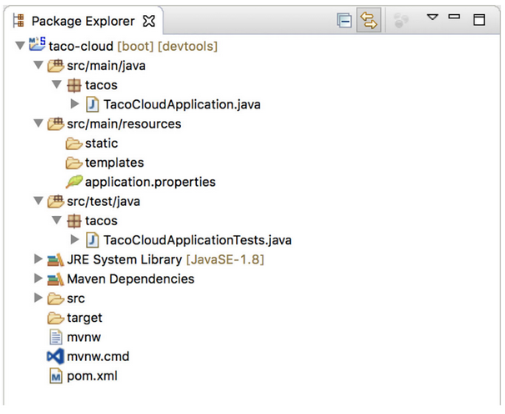


Рисунок 1.5 Дополнительные настройки Инициализации

По умолчанию мастер создания проекта вызывает Spring Initializr по адресу http://start.spring.io для генерации проекта. Как правило, нет необходимости переопределять это значение по умолчанию, поэтому можно нажать кнопку Finish на второй странице мастера. Но если по какой-то причине вы размещаете свой собственный клон Initializr (возможно, локальная копия на вашем собственном компьютере или настроенный клон, работающий внутри брандмауэра вашей компании), то вы можете изменить поле базовый Url, чтобы указать на ваш экземпляр Initializr, прежде чем нажать кнопку Finish.  
После нажатия кнопки Finish проект загружается из Initializr и загружается в рабочую область. Подождите несколько минут, пока он загрузится и соберется, а затем вы сможете начать разработку функциональности приложения. Но сначала давайте взглянем на то, что создал инициализатор.

**1.2.2 Изучение структуры проекта Spring**

После загрузки проекта в IDE откройте его, чтобы увидеть, что он содержит. На рисунке 1.6 показан открытый проект Taco Cloud в Spring Tool Suite.  
  
Рисунок 1.6 начальная структура проекта Spring, как показано в Spring Tool Suite

Вы можете увидеть, что это типичная Maven или Gradle структура проекта, где исходники приложения находится в src/main/java, код -тестовом находится под в src/test/java, а не в JAVA-ресурсах, которые расположены в src/main/resources. В рамках этой структуры проекта, вы должны знать элементы:

* mvnw и mvnw.cmd-это скрипты обертки Maven. Эти сценарии можно использовать для построения проекта, даже если на компьютере не установлен Maven.
* pom.xml-это спецификация сборки Maven. Мы рассмотрим это глубже через минуту.
* TacoCloudApplication.java-это основной класс Spring Boot, который запускает проект. Мы рассмотрим этот класс поближе через минуту.
* -application.properties - этот файл изначально пуст, но предлагает место, где можно указать свойства конфигурации. Мы немного повозимся с этим файлом в этой главе, но я отложу подробное объяснение свойств конфигурации до главы 5.
* static - эта папка, где вы можете разместить любой статический контент (изображения, таблицы стилей, JavaScript, и так далее), который вы хотите использовать в браузере. Изначально она пуста.
* templates - в этой папке вы разместите файлы шаблонов, которые будут использоваться для отображения содержимого в браузере. Изначально она пуста, но вскоре вы добавите шаблон Thymeleaf.
* TacoCloudApplicationTests.java - -это простой тестовый класс, который обеспечивает успешную загрузку контекста приложения Spring. По мере разработки приложения в него будут добавляться дополнительные тесты.

По мере развития проекта Taco Cloud, вы будите наполнять эту структуру проекта -Java-кодом, изображениями, таблицами стилей, тестами и другими сопутствующими материалами, которые сделают ваш проект более полным. Но давайте копнем немного глубже в некоторые из пунктов, которые предоставил Spring Initializr.

Изучение спецификации сборки

При заполнении формы -Initializr вы указали, что ваш проект должен быть построен с помощью Maven. Поэтому Spring Initializr построил для вас pom.xml-файл уже заполненный выбранными параметрами. Следующий листинг показывает весь pom.xml - файл, предоставленный инициализатором.

Листинг 1.1 первоначальная спецификация сборки Maven

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0       http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>sia</groupId>

<artifactId>taco-cloud</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>       <!------JAR packaging -->

<name>taco-cloud</name>

<description>Taco Cloud Example</description>

 <parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>2.0.4.RELEASE</version> <!------- Spring Boot version -->

<relativePath/> <!-- lookup parent from repository -->

</parent>

 <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>   
  <artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>  
<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.seleniumhq.selenium</groupId>  
<artifactId>htmlunit-driver</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

 <build>

<plugins>

  <plugin> <!-- Spring Boot plugin -->

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

      </plugin>

    </plugins>

  </build>

</project>

Первый примечательный пункт в pom.xml-файле является элемент <packaging>. Вы выбрали сборку приложения в виде исполняемого файла JAR, а не файла WAR. Это, вероятно, один из самых любопытных вариантов, которые вы сделаете, особенно для веб-приложения. В конце концов, традиционные веб-приложения Java упаковываются в виде WAR-файлов, оставляя JAR-файлы предпочтительной упаковкой для библиотек и экзотического настольного UI-приложения.  
Выбор упаковывать JAR для облако-ориентированных приложений правильный выбор. В то время как файлы WAR идеально подходят для развертывания на традиционном сервере приложений Java, они не подходят для большинства облачных платформ. Хотя некоторые облачные платформы (например, Cloud Foundry) способны развертывать и запускать файлы WAR, все облачные платформы Java способны запускать исполняемый файл JAR. Поэтому Spring Initializr по умолчанию использует jar-упаковку, если не указано иное.  
Если вы планируете развернуть приложение на традиционном сервере приложений Java, вам потребуется выбрать War packaging и включить класс веб-инициализатора. Более подробно мы рассмотрим создание файлов WAR в главе 2.  
Затем обратите внимание на элемент <parent> и, более конкретно, его дочерний элемент <version>. Это указывает, что родительским POM проекта является spring-boot-starter-parent. Помимо прочего, Родительский POM обеспечивает управление зависимостями для нескольких библиотек, обычно используемых в проектах Spring. Для библиотек, охватываемых родительским POM, указывать версию не нужно, так как она наследуется от родительской. В версии 2.0.4.RELEASE, указывается, что вы используете Spring Boot 2.0.4 и, таким образом, унаследуете управление зависимостями, как определено этой версией Spring Boot.  
Хотя мы говорим о зависимостях, обратите внимание, что есть три зависимости, объявленные в элементе <dependencies>. Первые две должны выглядеть несколько знакомыми. Они напрямую соответствуют зависимостям Web и Thymeleaf, выбранным до нажатия кнопки Готово в мастере создания проекта Spring Tool Suite. Третья зависимость предоставляет множество полезных возможностей тестирования. Вам не нужно было устанавливать флажок, чтобы он был включен, потому что Spring Initializr предполагает (надеюсь, правильно), что вы будете писать тесты.  
Можно также заметить, что все три зависимости имеют слово *starter* в идентификаторе артефакта. Зависимости Spring Boot starter отличаются тем, что обычно сами по себе не имеют кода библиотеки, а вместо этого временно извлекают другие библиотеки. Эти стартовые зависимости предлагают три основных преимущества:

* Файл сборки будет значительно меньше и проще в управлении, так как вам не нужно будет объявлять зависимость от каждой библиотеки, которая вам может понадобиться.
* Вы можете думать о своих зависимостях с точки зрения возможностей, которые они предоставляют, а не с точки зрения имен библиотек. Если вы разрабатываете веб-приложение, вы добавите зависимость web starter, а не список отдельных библиотек, которые позволяют писать веб-приложение.
* Вы освобождены от бремени беспокоиться о версиях библиотеки. Вы можете быть уверены в том, что для данной версии Spring Boot версии библиотеки, введенные транзитивно, будут совместимы. Вам нужно только беспокоиться о том, какую версию Spring Boot вы используете.

Наконец, спецификация сборки заканчивается плагином Spring Boot. Этот плагин выполняет несколько важных функций:

* Это является заданием Maven, которое позволяет запускать приложение с помощью Maven. Вы попробуете это в разделе 1.3.4.
* Это гарантирует, что все библиотеки зависимостей включены в исполняемый файл JAR и доступны в пути к классам среды выполнения.
* Это создает файл манифеста в файле JAR, который обозначает класс начальной загрузки (TacoCloudApplication, в вашем случае) как основной класс для исполняемого JAR.

Говоря о классе начальной загрузки (bootstrap), давайте откроем его и посмотрим поближе.

Начальная загрузка приложения

Поскольку вы будете запускать приложение из исполняемого файла JAR, важно иметь основной класс, который будет выполняться при запуске этого файла JAR. Вам также потребуется по крайней мере минимальный объем конфигурации Spring для начальной загрузки приложения. Это то, что вы найдете в классе TacoCloudApplication, показанном в следующем списке.

Листинг 1.2 Taco Cloud bootstrap class

package tacos;

import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class TacoCloudApplication {  
   public static void main(String[] args) {  
      SpringApplication.run(TacoCloudApplication.class, args);  
   }  
}

Одна из самых мощных строк кода также является одной из самых коротких. Аннотация @SpringBootApplication ясно указывает, что это приложение Spring Boot. Но в @SpringBootApplication есть больше, чем кажется на первый взгляд.

@SpringBootApplication - это составное приложение, которое объединяет три других аннотации:

- @SpringBootConfiguration - обозначает, что этот класс является классом конфигурации. Несмотря на то, что в классе еще не так много конфигурации, при необходимости можно добавить Java-based Spring Framework конфигурацию. Эта аннотация является, фактически, специализированной формой аннотации @Configuration.

- @EnableAutoConfiguration - включает автоматическое Spring Boot конфигурирование. Мы поговорим об автоконфигурации позже. На данный момент знайте, что эта аннотация говорит Spring Boot автоматически настраивать любые компоненты, которые, по его мнению, вам понадобятся.

- @ComponentScan - включает сканирование компонентов. Это позволяет объявлять другие классы с аннотациями, такими как @Component, @Controller, @Service и другими, чтобы Spring автоматически обнаруживала их и регистрировала как компоненты в контексте приложения Spring.

Другой важной частью *TacoCloudApplication* является метод *main()*. Это метод, который будет выполняться при выполнении файла JAR. По большей части, этот метод шаблонный код; каждое приложение Spring Boot, которое вы пишете, будет иметь метод, аналогичный или идентичный этому (за исключением различий в именах классов).  
Метод main () вызывает статический метод run () класса SpringApplication, который выполняет фактическую загрузку приложения, создавая контекст приложения Spring. Два параметра, переданные методу run (), являются конфигурационным классом и аргументы командной строки. Хотя это не обязательно, что конфигурационный класс, передаваемый в run(), будет таким же, как класс начальной загрузки, это наиболее удобно и стандартный выбор.

Скорее всего, вам не нужно будет ничего менять в классе начальной загрузки. Для простого приложения, вы можете найти его удобным для настройки одного или двух других компонентов в классе начальной загрузки, но для большинства приложений, вам лучше создать отдельный класс конфигурации для всего, что не настроено автоматически. Вы определите несколько конфигураций на протяжении всего курса этой книги, так что следите за подробностями.

ТЕСТИРОВАНИЕ ПРИЛОЖЕНИЯ

Тестирование является важной частью разработки программного обеспечения. Признавая это, Spring Инициализатор предоставляет тестовый класс для начала работы. Ниже приведен базовый уровень тестового класса.

Листинг 1.3 Базовый тест приложения

package tacos;  
import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.springframework.boot.test.context.SpringBootTest;  
import org.springframework.test.context.junit4.SpringRunner;  
  
      @RunWith(SpringRunner.class)  /\*Использование Spring runner\*/  
@SpringBootTest /\* String Boot тест\*/  
public class TacoCloudApplicationTests {  
   @Test /\*Метод теста\*/  
   public void contextLoads() {  
   }  
}

В тестах TacoCloudApplicationTests мало что можно увидеть: единственный метод теста в классе пуст. Тем не менее, этот тестовый класс действительно выполняет существенную проверку, чтобы гарантировать, что контекст приложения Spring может быть загружен успешно. При внесении каких-либо изменений, препятствующих созданию Spring applicationcontext, этот тест завершается неудачей, и можно реагировать путем устранения проблемы.

Также обратите внимание на класс с аннотацией @RunWith (SpringRunner.класс.) @RunWith является аннотацией JUnit, обеспечивая тестовый runner, помогающий JUnit в выполнении теста. Думайте об этом как о применении плагина к JUnit для обеспечения пользовательского поведения тестирования. В этом случае JUnit получает SpringRunner, тестовый модуль, предоставляемый Spring, который обеспечивает создание контекста приложения Spring, с которым будет выполняться тест.

ТЕСТ-RUNNER ПОД ДРУГИМ ИМЕНЕМ…

Если вы уже знакомы с написанием тестов Spring или, возможно, видели какие-то реальные классы тестов Spring, вы, возможно, видели тестовый runner с именем SpringJUnit4ClassRunner. SpringRunner является псевдонимом SpringJUnit4ClassRunner и был введен в Spring 4.3, чтобы удалить связь с определенной версией JUnit (например, JUnit 4). И нет никаких сомнений в том, что псевдоним легче читать и печатать.

@SpringBootTest говорит JUnit что, запуск тестов должен осуществляться с использованием Spring Boot. Сейчас достаточно считать, что этот тест класса эквивалентно вызову SpringApplication.run() в методе main(). В течение этой книги вы увидите @SpringBootTest несколько раз, и мы раскроем часть его силы.

Наконец, есть сам метод тестирования. Хотя @RunWith(SpringRunner.class) и @SpringBootTest поручено загрузить контекст приложения Spring для теста, они не будут иметь ничего общего, если нет никаких методов тестирования. Даже без каких-либо объявлений или кода, этот пустой метод тестирования предложит двум аннотациям выполнить свою работу и загрузить контекст приложения Spring. Если при этом возникают какие-либо проблемы, тест не выполняется.

На этом мы завершили обзор кода, предоставленного Spring Initializr. Вы видели некоторые стандартные основы, которые можно использовать для разработки приложения Spring, но вы до сих пор не написали ни одной строки кода. Теперь пришло время запустить IDE, убрать пыль с клавиатуры и добавить пользовательский код в приложение Taco Cloud.

Написание приложения Spring

Так как, вы только начинаете, мы начнем с относительно небольшого изменения в приложении Taco Cloud, но такого, которое продемонстрирует большую полезность Spring. Кажется уместным, что, поскольку вы только начинаете, первая функция, которую вы добавите в приложение Taco Cloud, - это домашняя страница. Чтобы добавить главную страницу, вы создадите два artifact:

* контроллер-класс, который обрабатывает запросы для главной страницы
* визуальный шаблона, который определяет, как выглядит домашняя страница

И помня как важно тестирование, вы также написать простой тестовый класс для проверки домашней страницы. Но сначала о главном ... давайте напишем контроллер.

**1.3.1 Обработка веб-запросов**

Spring поставляется с мощным веб-фреймворком, известным как Spring MVC. В центре Spring MVC находится концепция контроллера, класса, который обрабатывает запросы и отвечает какой-либо информацией. В случае обращения веб-приложения, контроллер отвечает, при необходимости заполняя данные модели и передавая запрос представлению для создания HTML-кода, возвращаемого браузеру. Вы узнаете много нового о Spring MVC в главе 2. Но пока вы напишете простой класс контроллера, который обрабатывает запросы корневого пути (например,/) и пересылает эти запросы в представление домашней страницы без заполнения данных модели. Ниже приведен простой класс контроллера.

Листинг 1.4 контроллер домашней страницы

package tacos;

import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.GetMapping;  
@Controller                 /\*Контроллер\*/  
public class HomeController {  
  
 @GetMapping("/")          /\*Обрабатывает запросы корневого пути \*/  
 public String home() {  
 return "home"; /\*Возвращает имя представления\*/  
   }  
}

Как вы можете видеть, этот класс аннотируется @Controller. Сам по себе @Controller ничего не делает. Его основное назначение-идентифицировать этот класс как компонент для сканирования компонентов. Поскольку HomeController аннотируется @Controller, при сканирование компонентов Spring автоматически обнаруживает его и создает экземпляр HomeController в качестве компонента в контексте приложения Spring.

Фактически, несколько различных аннотаций (включая @Component, @Service и @Repository) служат цели, подобной @Controller. Вы могли бы так же эффективно аннотировать HomeController с любой из этих других аннотаций, и он все равно работал бы так же. Выбор @Controller, однако, более описателен роли этого компонента в приложении.

Метод home() так же прост, как методы контроллера. Он аннотируется @GetMapping, чтобы указать, что если HTTP GET запрос получен для корневого пути /, то этот метод должен обработать такой запрос. Все что он делает - это возвращение String значения “home”.

Это значение интерпретируется как логическое имя представления. Реализация этого представления зависит от нескольких факторов, но поскольку Thymeleaf находится в classpath, можно для этого задать шаблон с помощью Thymeleaf.

ПОЧЕМУ ИМЕННО THYMELEAF?

Вы можете быть удивлены, почему мы выбрали Thymeleaf для шаблонизатора. Почему не JSP? Почему не FreeMarker? Почему не один из нескольких других вариантов?

Проще говоря, я должен был выбрать что-то, и я люблю Thymeleaf и вообще предпочитаю его над всеми другими вариантами. И хотя JSP может показаться очевидным выбором, есть некоторые проблемы, которые необходимо преодолеть при использовании JSP с Spring Boot. Я не хотел спускаться в кроличью нору в первой главе. Держись. Мы рассмотрим другие варианты шаблонов, включая JSP, в главе 2.

Имя шаблона является производным от логическое имени представления, предварив его с префиксом /templates/ и .HTML. Результирующий путь для шаблона - /templates/home.HTML. Поэтому вам нужно будет разместить шаблон в вашем проекте в /src/main/resources/templates/home.HTML. Давайте создадим этот шаблон сейчас.

**1.3.2 Задание представления**

В интересах сохранения простоты домашней страницы, она не должно делать ничего, кроме как приветствовать пользователей на сайте. В следующем листинге показан базовый шаблон Thymeleaf, определяющий домашнюю страницу Taco Cloud.

Листинг 1.5 Шаблон домашней страницы Taco Cloud

<!DOCTYPE html>  
<html xmlns="<http://www.w3.org/1999/xhtml>"  
  xmlns:th="[http://www.thymeleaf.org](http://www.thymeleaf.org/)">  
 <head>  
   <title>Taco Cloud</title>  
 </head>  
 <body>  
<h1>Welcome to...</h1>  
<img th:src="@{/images/TacoCloud.png}"/>  
 </body>  
</html>

Нечего обсуждать по поводу этого шаблона. Единственной заметной строкой кода является строка с тегом <img> для отображения логотипа Taco Cloud. Он использует атрибут Thymeleaf th:src и @{...} выражение для ссылки на изображения с контекст-относительным путем. Кроме этой особенности, это не намного больше, чем страница Hello World.

Но давайте поговорим об этой картинке немного подробнее. Я оставлю это на вас, чтобы определить логотип Taco Cloud, который вам нравится. Вам нужно будет убедиться, что вы разместите его в нужном месте в проекте.

Изображения есть ссылка с контексто-относительным путем /images/TacoCloud.png. Как вы помните из нашего обзора структуры проекта, статическое содержимое, такое как изображения, хранится в папке/src/main/resources /static. Это означает, что изображение логотипа Taco Cloud  также должно находиться в проекте по адресу / src/main/resources/static/images/TacoCloud.png.

Теперь, когда у вас есть контроллер для обработки запросов для главной страницы и шаблон для отображения домашней страницы, вы почти готовы, чтобы запустить приложение и увидеть его в действии. Но сначала давайте посмотрим, как можно написать тест контроллера.

1.3.3 Тестирование контроллера

Тестирование веб-приложений может быть сложным при утверждении содержимого HTML-страницы. К счастью, Spring поставляется с мощной тестовой поддержкой, которая упрощает тестирование веб-приложения. Для целей домашней страницы вы напишете тест, сравнимый по сложности с самой домашней страницей. Ваш тест выполнит HTTP-запрос GET для корневого пути / и ожидает успешного результата, когда имя представления является домашним, а результирующее содержимое содержит фразу " Добро пожаловать...”.

Листинг 1.6 Тест для контроллера страницы

package tacos;

import static org.hamcrest.Matchers.containsString;  
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.status;  
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;  
import org.junit.Test; import org.junit.runner.RunWith;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;  
import org.springframework.test.context.junit4.SpringRunner;  
import org.springframework.test.web.servlet.MockMvc;  
  
@RunWith(SpringRunner.class)  
@WebMvcTest(HomeController.class) /\*Веб-тест для HomeController\*/  
public class HomeControllerTest {  
     
   @Autowired  
   private MockMvc mockMvc;       /\*Внедряет MockMvc\*/

 @Test  
  public void testHomePage() throws Exception {  
     mockMvc.perform(get("/"))              /\*Выполняет GET \*/  
       .andExpect(status().isOk())    /\*-Ожидает HTTP 200\*/  
       .andExpect(view().name("home"))    /\*Ожидает home\*/  
       .andExpect(content().string( /\*Welcome to...\*/  
  
   containsString("Welcome to...")));  }  
}

Первое, что вы могли бы заметить в этом тесте, это то, что написанное немного отличается от класса TacoCloudApplicationTests в части аннотаций, примененных к нему. Вместо разметки @SpringBootTest, HomeControllerTest аннотируется @WebMvcTest. Это специальная тестовая аннотация, предоставляемая Spring Boot, которая организует выполнение теста в контексте приложения Spring MVC. Более конкретно, в этом случае, он организует для HomeController регистрацию с использованием Spring MVC, так что вы можете отправлять запросы из него.

@WebMvcTest также настраивает поддержку Spring для тестирования Spring MVC.

Метод testHomePage() определяет тест, который вы хотите выполнить на главной странице. Он начинается с объекта MockMvc для выполнения запроса HTTP GET для / (корневой путь). К этому запросу устанавливаются следующие ожидания:

* ответ должен иметь состояние HTTP 200 (OK).
* view должен иметь логическое название home.
* Отображаемое представление должно содержать текст "Welcome to...”

Если после того, как объект MockMvc выполняет запрос, любое из этих ожиданий не выполняется, то тест завершается неудачей. Но ваш контроллер и шаблон представления написаны, чтобы удовлетворить эти ожидания, поэтому тест должен пройти удачно — или, по крайней мере, с некоторым оттенком зеленого, указывающим на прохождение теста.

Контроллер был написан, шаблон представления создан, и у вас есть тест. Похоже, что вы успешно реализовали домашнюю страницу. Но даже несмотря на то, что тест проходит, есть что-то более важное, визуальный результат в браузере. В конце концов, именно это клиенты Taco Cloud увидят. Давайте создадим приложение и запустим его.

1.3.4 Создание и запуск приложения

Так же, как существует несколько способов инициализации приложения Spring, существует несколько способов его запуска.

Так как для инициализации и работы над проектом вы выбрали Spring Tool Suite, у вас есть удобная функция Spring Boot Dashboard, которая поможет запустить приложение в среде IDE. Панель мониторинга Spring Boot отображается в виде вкладки, обычно в левом нижнем углу окна IDE. На рисунке 1.7 показан аннотированный снимок экрана панели управления Spring Boot Dashboard.

Я не хочу тратить много времени на все, что делает Spring Boot Dashboard, хотя рисунок 1.7 охватывает некоторые из самых полезных деталей. Сейчас важно знать, как использовать его для запуска приложения Taco Cloud. Убедитесь, что приложение taco-cloud выделено в списке проектов (это единственное приложение, показанное на рисунке 1.7), а затем нажмите кнопку start (самая левая кнопка с зеленым треугольником и красным квадратом). Приложение должно запуститься.

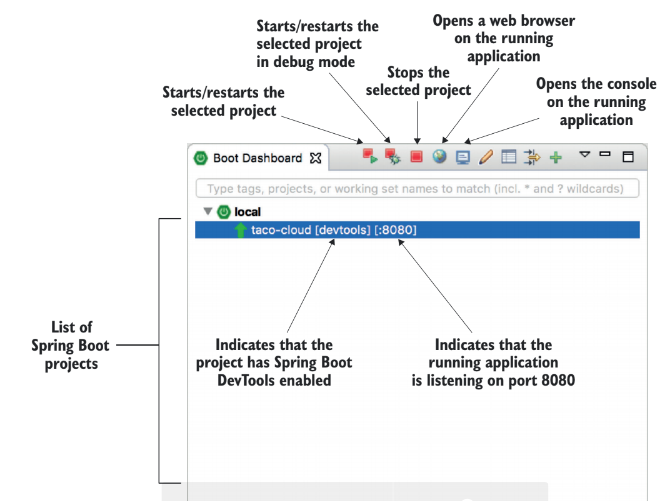


Рисунок 1.7 основные моменты панели Spring Boot Dashboard

* Запуск/перезапуск выбранного проекта
* Запуск/перезапуск выбранного проекта в режиме отладки
* Остановить выбранный проект
* Открывает веб-браузер с запущенным приложении
* Открывает консоль запущенного приложения
* Список проектов Spring Boot
* Указывает, что в проекте включены средства разработки Spring Boot
* Указывает, что запущенное приложение прослушивает порт 8080

При запуске приложения вы увидите, как в консоли пролетает Spring ASCII art, а затем некоторые записи журнала, описывающие шаги при запуске приложения. Прежде чем журнал остановится, вы увидите запись в журнале о том, что Tomcat запущен на портах: 8080 (http), что означает, что вы готовы указать свой веб-браузер на главной странице, чтобы увидеть плоды своего труда.

Подожди минутку. Tomcat запустился? Вы развернули приложение в Tomcat?

Приложения Spring Boot, как правило, приносят с собой все, что им нужно, и их не нужно развертывать на каком-либо сервере приложений. Вы никогда не развертывали приложение в Tomcat ... Tomcat является частью вашего приложения! (Я опишу детали того, как Tomcat стал частью вашего приложения в разделе 1.3.6.)

Теперь, когда приложение запущено, укажите в веб-браузере http://localhost:8080 (или нажмите кнопку Глобус на панели Spring Boot Dashboard), и вы увидите что-то вроде рисунка 1.8. Ваши результаты могут отличаться, если вы разработали свой собственный логотип. Но это не должно сильно отличаться от того, что вы видите на рисунке 1.8.

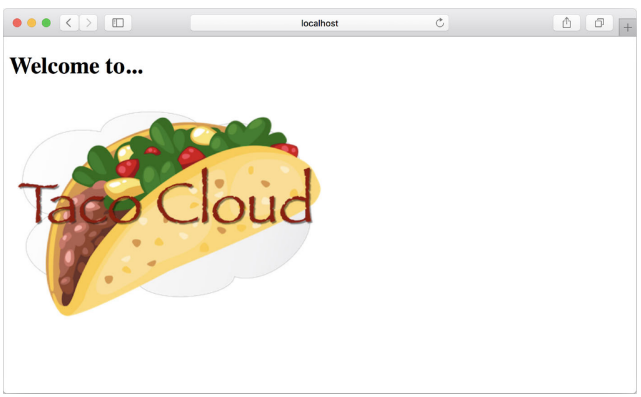


Рисунок 1.8 Начальная страница Taco Cloud

Вроде особо не на что смотреть. Но это не совсем книга по графическому дизайну. Скромной главной страницы более чем достаточно.  
Одна вещь, которую я не рассмотрел до сих пор это DevTools. Вы выбрали его в качестве зависимости при инициализации проекта. Это отражено в зависимостях в pom.xml файле. И Spring Boot Dashboard даже показывает, что в проект включен DevTools. Но что такое DevTools и что он делает для вас? Давайте рассмотрим несколько наиболее полезных функций DevTools.

**1.3.5 Знакомство с Spring Boot DevTools**

Как следует из названия, DevTools предоставляет разработчикам Spring несколько удобных инструментов для разработки. Среди них

* автоматическая перезагрузки приложения если код меняется
* автоматическое обновление приложения в браузера когда обновляются ресурсы предназначенные для браузерного отображения (такие как шаблоны, Скрипты, таблицы стилей и т. д.)
* автоматическое отключение кэша шаблона
* сборка в Н2 консоли, если используется Н2 БД

Важно понимать, что DevTools это не плагин IDE, т.е. не требуют использования конкретных IDE. Он одинаково хорошо работает в Spring Tool Suite, IntelliJ IDEA и NetBeans. Кроме того, поскольку он предназначен только для целей разработки, он достаточно умен, чтобы отключить себя при развертывании в рабочей среде. (Мы обсудим, как это делается, когда вы приступите к развертыванию приложения в главе 19.) Теперь давайте сосредоточимся на наиболее полезных функциях Spring Boot DevTools, начиная с автоматического перезапуска приложения.

Автоматический перезапуск приложения

С помощью DevTools в рамках вашего проекта вы сможете вносить изменения в код Java и файлы свойств в проекте и видеть, как эти изменения применяются почти сразу.

DevTools отслеживает изменения, и когда он видит, что что-то изменилось, он автоматически перезапустить приложение.

Точнее, когда DevTools запущен, приложение загружается в два отдельных загрузчика класса в виртуальной машине Java (JVM). Один загрузчик класса загружается с вашим кодом Java, файлами свойств и почти всем, что находится в src/main/path проекта. Это элементы, которые могут часто изменяться. Другой класс loader загружается библиотеками зависимостей, которые вряд ли будут меняться так часто.

При обнаружении изменения DevTools перезагружает только загрузчик классов, содержащий код проекта, и перезапускает контекст приложения Spring, но оставляет другой загрузчик классов и JVM нетронутыми. Хотя эта стратегия и тонкая, она позволяет немного сократить время, необходимое для запуска приложения.

Недостатком этой стратегии является то, что изменения зависимостей будут недоступны при автоматическом перезапуске. Это происходит потому, что загрузчик классов, содержащий библиотеки зависимостей, не перезагружается автоматически. Это означает, что каждый раз, когда вы добавляете, изменяете или удаляете зависимость в спецификации сборки, вам нужно будет выполнить жесткий перезапуск приложения, чтобы эти изменения вступили в силу.

АВТОМАТИЧЕСКОЕ ОБНОВЛЕНИЕ БРАУЗЕРА И ОТКЛЮЧЕНИЕ КЭША ШАБЛОНОВ

По умолчанию параметры шаблона, такие как Thymeleaf и FreeMarker, настроены для кэширования результатов синтаксического анализа шаблона, так чтобы шаблоны не нужно было анализировать с каждым запросом, который они обслуживают. Это хорошо для продакшена.

Кэшированные шаблоны, однако, мешают во время разработки. Кэшированные шаблоны делают невозможным внесение изменений в шаблоны во время работы приложения и просмотр результатов после обновления браузера. Даже если вы внесли изменения, кэшированный шаблон будет использоваться до перезапуска приложения.

DevTools устраняет эту проблему, автоматически отключая кэширование всех шаблонов. Внесите столько изменений, сколько вы хотите, в свои шаблоны и проверьте их сразу обновив браузер.

Но если вы похожи на меня, вы даже не хотите обременяться усилиями нажатия кнопки обновления браузера. Было бы намного лучше, если бы вы могли внести изменения и сразу увидеть результаты в браузере. К счастью, в DevTools есть что-то особенное для тех из нас, кому лень нажимать кнопку обновления.

Когда DevTools запущен, он автоматически включает LiveReload (http://livereload.com/) сервер вместе с вашим приложением. Сам по себе сервер LiveReload не очень полезен. Но в сочетании с соответствующим плагином браузера LiveReload он заставляет ваш браузер автоматически обновляться при внесении изменений в шаблоны, изображения, таблицы стилей, JavaScript и т. д.—фактически, почти все, что в конечном итоге отправляется в ваш браузер.

LiveReload имеет плагины для браузеров Google Chrome, Safari и Firefox. ((Сорри, Internet Explorer и Edge фанаты)) Посетите http://livereload.com/extensions/ чтобы найти информацию о том, как установить LiveReload для Вашего браузера.

Сборка в H2 консоле

Хотя ваш проект еще не использует базу данных, это изменится в главе 3. Если вы решите использовать базу данных H2 для разработки, DevTools также автоматически активирует консоль H2, доступ к которой можно получить из веб-браузера. Вам нужно только указать в веб-браузере http://localhost:8080/h2-console чтобы получить представление о данных, с которыми работает приложение.

На данный момент Вы написали полное, хотя и простое, Spring приложение. Вы будете расширять его на протяжении всего курса книги. Но сейчас самое время сделать шаг назад и пересмотреть то, что вы сделали, и как Spring вам в этом помогла.

1.3.6 Давайте рассмотрим

Вспомните, как вы дошли до этого момента. Короче говоря, это шаги, которые Вы предприняли для создания Spring-based Taco Cloud приложения:

* Вы создали первоначальную структуру проекта с помощью Spring Initializr.
* Вы написали класс контроллера для обработки запроса домашней страницы.
* Вы определили шаблон представления для отображения главной страницы.
* Вы написали простой тестовый класс, чтобы проверить что все работает.

Кажется довольно просто, не так ли? За исключением первого шага по начальной загрузке проекта, каждое действие, которое Вы предприняли, было сосредоточено на достижении цели создания домашней страницы

Фактически, почти каждая строка написанного вами кода направлена на достижение этой цели. Не считая операторов import, я считаю только две строки кода в вашем классе контроллера и никаких строк в шаблоне представления, которые являются специфичными для Spring. И хотя основная часть тестового класса использует поддержку Spring тестирования, это кажется немного менее инвазивным в контексте теста.

Это важное преимущество разработки с Spring. Можно сосредоточиться на коде, удовлетворяющем требованиям приложения, а не на удовлетворении требованиям framework-а. Хотя вам, несомненно, придется время от времени писать специфичный для framework-а код, обычно это будет лишь небольшая часть вашей работы. Как я уже говорил, Spring (с Spring Boot)можно считать frameworkless framework.

Как это вообще работает? Что Spring делают под капотом, чтобы убедиться, что ваши требования выполняются? Чтобы понять, что делает Spring, давайте начнем с рассмотрения спецификации сборки.

В  файле pom.xml, вы объявили зависимость от Web и Thymeleaf starter-ов. Эти две зависимости транзитивно принесли несколько других зависимостей, в том числе

* Spring’s MVC framework
* Встроенный Tomcat
* Thymeleaf и макет Thymeleaf диалекта

Также добавлены Spring Boot-овские конфигурируемые при запуске библиотеки. При запуске приложения Spring Boot автоконфигурирование обнаруживает эти библиотеки и автоматически:

* Настраивает bean-ы в контексте приложения Spring для включения Spring MVC
* Настраивает встроенный сервер Tomcat в контексте приложения Spring
* Настраивает Thymeleaf view resolver для рендеринга Spring MVC представления с Thymeleaf шаблонами

Короче говоря, автоконфигурация выполняет всю сложную работу, позволяя сосредоточиться на написании кода, реализующего функциональность приложения. Это довольно приятно, если вас интересует мое мнение!

Ваше Spring путешествие только началось. Приложение Taco Cloud затронуло лишь небольшую часть того, что Spring может предложить. Прежде чем вы сделаете следующий шаг, давайте рассмотрим Spring пейзаж и посмотрим, с какими достопримечательностями вы столкнетесь в своем путешествии.

1.4 Съемка Spring ландшафта

Чтобы получить представление о Spring ландшафте, посмотрите на огромный список флажков в полной версии веб-формы Spring Initializr. В нем перечислены более 100 вариантов зависимостей, поэтому я не буду пытаться перечислить их все здесь или предоставить скриншот. Но я призываю вас взглянуть самим. В то же время, я упомяну некоторые основные моменты.

1.4.1 Core Spring Framework

Как и следовало ожидать, ядро Spring Framework является основой всего остального в Spring вселенной. Оно предоставляет базовый контейнер и инфраструктуру внедрения зависимостей. Но оно также предоставляет несколько других важных функций.

Среди них Spring MVC, веб-фреймворк Spring. Вы уже видели, как использовать Spring MVC для написания класса контроллера для обработки веб-запросов. Однако вы еще не видели, что Spring MVC также можно использовать для создания REST API, которые возвращают ответ не в виде HTML. Мы собираемся углубиться в Spring MVC в главе 2, а затем еще раз взглянуть на то, как использовать его для создания REST API в главе 6.

Ядро Spring Framework также предлагает некоторую элементарную поддержку сохраняемости данных, в частности на основе шаблонов поддержки JDBC. Вы увидите, как использовать JdbcTemplate в главе 3.

В последней версии Spring (5.0.8) была добавлена поддержка программирования в реактивном стиле, включая новую реактивную веб-платформу Spring WebFlux, которая в значительной степени заимствуется из Spring MVC. Вы посмотрите на модель реактивного программирования Spring в части 3 и Spring WebFlux отдельно в главе 10.

1.4.2 Spring Boot

Мы уже видели многие преимущества Spring Boot, включая зависимости от стартера и автоконфигурацию. Будьте уверены, что мы будем использовать как можно больше Spring Boot в этой книге и избегать любой формы явной конфигурации, если только без этого не обойтись. Но в дополнение к зависимостям запуска и автоконфигурации Spring Boot также предлагает несколько других полезных функций:

* Actuator provide обеспечивает понимание внутренней работы приложения, включая метрики, информацию о дампе потока, работоспособность приложения и свойства среды, доступные приложению.
* Гибкая спецификация свойств среды.
* Дополнительное тестирование поддержки поверх тестирования core framework.

Более того, Spring Boot предлагает альтернативную модель программирования, основанную на скриптах Groovy, которая называется Spring Boot CLI (интерфейс командной строки). С помощью интерфейса командной строки Spring Boot можно создавать целые приложения в виде набора сценариев Groovy и запускать их из командной строки. Мы не будем тратить много времени на Spring Boot CLI, но мы коснемся его, когда он будет нам необходим.

Spring Boot стал неотъемлемой частью разработки на Spring; я не могу представить разработку Spring-приложения без него. Следовательно, эта книга имеет структуру, ориентированную на Spring Boot, и вы можете поймать меня, на использование слова Spring, когда я имею в виду что-то, что делает Spring Boot.

1.4.3 Spring Data

Несмотря на то, что базовая платформа Spring поставляется с поддержкой сохранения базовых данных, Spring Data предоставляет нечто удивительное: возможность определять репозитории данных вашего приложения как простые интерфейсы Java, используя соглашение об именовании при определении методов управления хранением и извлечением данных. Более того, Spring Data способна работать с несколькими различными типами баз данных, включая реляционные (JPA), document (Mongo), graph (Neo4j) и другие. Вы будете использовать Spring Data для создания репозиториев для приложения Taco Cloud в главе 3.

1.4.4 Spring Security

Безопасность приложений всегда была важной темой, и с каждым днем она становится все более важной. К счастью, Spring имеет надежные границы безопасности в Spring Security. Spring Security удовлетворяет широкий спектр потребностей безопасности приложений, включая проверку подлинности, авторизацию и безопасность API. Хотя объем Spring Security слишком велик, чтобы быть должным образом рассмотрены в этой книге, мы коснемся некоторых из наиболее распространенных случаев использования в главах 4 и 12.

1.4.5 Spring Integration и Spring Batch

В какой-то момент большинству приложений потребуется интеграция с другими приложениями или даже с другими компонентами того же приложения. Для удовлетворения этих потребностей было разработано несколько моделей интеграции прикладных программ. Spring Integration и Spring Batch обеспечивают реализацию этих шаблонов для приложений на основе Spring.

Spring Integration относиться к интеграции в реальном времени, где данные обрабатываются по мере их доступности. В отличие от этого, Spring Batch относиться к пакетной интеграции, где данные разрешено собирать в течение времени, пока какой-либо триггер (возможно, триггер времени) не сигнализирует, что пришло время отправки пакета данных на обработку. Вы изучите как Spring Batch, так и Spring Integration в главе 9.

1.4.6 Spring Cloud

Сейчас, когда я пишу эту статью, мир разработки приложений вступает в новую эру, когда мы больше не будем разрабатывать наши приложения как монолиты единого блока развертывания и вместо этого будем создавать приложения из нескольких отдельных блоков развертывания, известных как микрослужбы (микросервисы).

Микросервисы - это актуальная тема, затрагивающая несколько практических аспектов разработки и среды выполнения. Однако при этом они порождают свои собственные проблемы. Эти проблемы решаются с помощью Spring Cloud, коллекции проектов для разработки облачных приложений с помощью Spring.

Spring Cloud содержит множество полезных компонентов, и было бы невозможно охватить все это в этой книге. Мы рассмотрим некоторые из наиболее распространенных компонентов Spring Cloud в главах 13, 14 и 15. Для более полного обсуждения Spring Cloud я предлагаю взглянуть на Spring Microservices in Action от John Carnell (Manning, 2017, www.manning .com/books/spring-microservices-in-action).

Итог

* Spring стремится упростить решение задач разработчиками, такие как создание веб-приложений, работа с базами данных, защита приложений и разработку микросервисов.
* Spring Boot опирается на Spring, чтобы сделать Spring еще проще с упрощенным управлением зависимостями, автоматической конфигурацией и анализ среды выполнения.
* Spring приложения могут быть инициализированы с помощью Spring Initializr, который web-ориентирован и поддерживается изначально в большинстве  Java-вских средах разработки.
* Компоненты, обычно называемые bean-ами, в контексте приложения Spring могут быть объявлены явным образом с помощью Java или XML, обнаружены при сканировании компонентов или автоматически настроены с автоматической настройкой Spring Boot.

[Spring in Action Covers Spring 5.0 перевод на русский. Глава 2](https://beginner-mercenary.blogspot.com/2019/04/spring-in-action-covers-spring-50-2.html)

**2.Разработка web-приложений**

*В этой главе рассматривается*

* *представление данных модели в браузере*
* *обработка и проверка форм ввода*
* *выбор библиотеки шаблонов представлений*

Первое впечатление очень важно. Отличная презентация может продать дом задолго до того, как покупатель дома войдет в дверь. Вишневый окрас автомобиля повернет больше голов, чем то, что находится под капотом. А литература изобилует историями о любви с первого взгляда. То, что внутри, очень важно, но то, что снаружи-то, что видно первым - важнее.

Приложения, которые вы создадите с помощью Spring, будут выполнять все виды действий, включая проверку данных, чтение информации из базы данных и взаимодействие с другими приложениями. Но первое впечатление, которое получат пользователи вашего приложения, - это пользовательский интерфейс. И во многих приложениях, UI-это веб-приложение, представленное в браузере.

В главе 1 Вы создали свой первый контроллер Spring MVC для отображения домашней страницы приложения. Но Spring MVC может делать гораздо больше, чем просто отображать статический контент. В этой главе вы разработаете первый крупный компонент функциональности в вашем Taco Cloud приложении - возможность создавать собственные такос. При этом вы углубитесь в Spring MVC и увидите, как отображать данные модели и обрабатывать данные форм ввода.

2.1 Отображение информации

По сути, Taco Cloud-это место, где вы можете заказать тако онлайн. Но более того, в Taco Cloud хочется предоставить возможность своим клиентам выразить свою творческую сторону и разработать собственные тако из богатой палитры ингредиентов.

Поэтому веб-приложению Taco Cloud требуется страница, на которой отображается выбор ингредиентов для создателей тако. Выбор ингредиентов может измениться в любое время, поэтому они не должны быть жестко закодированы в HTML-страницу. Скорее, список доступных ингредиентов должен быть извлечен из базы данных и передан на страницу, которая будет отображаться для клиента.

В веб-приложении Spring задача контроллера заключается в получении и обработке данных. Также должна производить обработка данных в  представление, чтобы отобразить эти данные в HTML, которые будут отображаться в браузере. Мы собираемся создать следующие компоненты для страницы создания тако:

* доменный класс, который определяет свойства тако ингредиента
* Spring MVC класс контроллера, который вытаскивает информацию об ингредиенте и  передает ее на отображение
* шаблон отображения информации, который отображает список ингредиентов в браузере пользователя

Взаимосвязь между этими компонентами показана на рисунке 2.1.

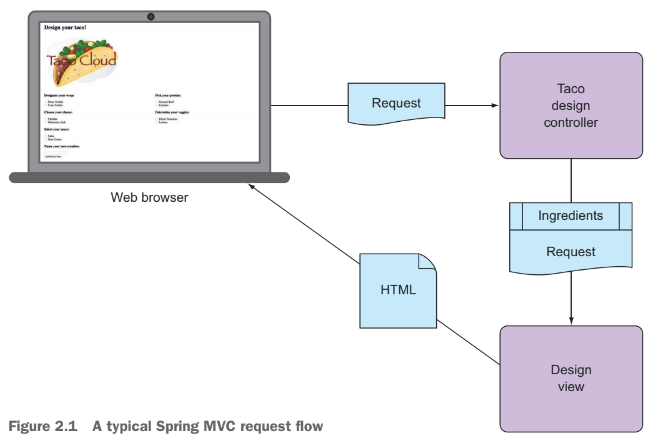


Рисунок 2.1 типичный поток запросов Spring MVC

Поскольку в этой главе основное внимание уделяется веб-платформе Spring, мы перенесем весь материал базы данных на Главу 3. На данный момент контроллер будет нести полную ответственность за предоставление ингредиентов для представления. В главе 3 Вы переработаете контроллер для совместной работы с репозиторием, который извлекает данные ингредиентов из базы данных.

Прежде чем написать контроллер и представление, давайте выберем тип домена, представляющий компонент. Это создаст основу для разработки веб-компонентов.

2.1.1 Создание домена

Домен приложения-это тематика, которую оно затрагивает—идеи и концепции, которые влияют на понимание приложения (Для более глубокого обсуждения областей применения я предлагаю дизайн, управляемый доменами Эрика Эванса (Addison-Wesley Professional, 2003).). В приложении Taco Cloud домен включает такие объекты, как описание тако, ингредиенты, из которых состоят эти описания, клиенты и заказы тако, размещенные клиентами. Для начала мы сосредоточимся на ингредиентах тако.

В нашем домене тако-ингредиенты довольно простые объекты. Каждый из них имеет имя, а также тип, так что он может быть визуально классифицирован (протеины, сыры, соусы, и так далее). Каждый из них также имеет идентификатор, по которому на него можно легко и однозначно ссылаться. Следующий класс Ingredient определяет необходимый объект домена.

Листинг 2.1 Определение ингредиента тако

package tacos;

import lombok.Data;  
import lombok.RequiredArgsConstructor;  
  
@Data  
@RequiredArgsConstructor  
public class Ingredient {  
 private final String id;  
 private final String name;  
 private final Type type;  
  
 public static enum Type {  
   WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE  
 }  
}

Как вы можете видеть, это заурядный класс домена Java, определяющий три свойства, необходимые для описания ингредиента. Возможно, самое необычное в классе Ingredient, как он определен в листинге 2.1, заключается в том, что в нем отсутствует обычный набор методов getter и setter, не говоря уже о таких полезных методах, как equals (), hashCode (), toString () и других.

Вы не видите их в списке частично, чтобы сэкономить место, но и потому, что вы используете удивительную библиотеку под названием Lombok для автоматического создания этих методов во время выполнения. Фактически, аннотация @Data на уровне класса предоставляется Lombok и указывает Lombok генерировать все эти отсутствующие методы, а также конструктор, который принимает все конечные свойства в качестве аргументов. Путем использование Lombok, вы можете держать код для Ingredient коротким и понятным.

Lombok-это не Spring-овая библиотека, но она настолько невероятно полезна, что мне уже трудно без нее. И она спасает, когда мне нужно чтобы примеры кода в книге были короткими и понятными.

Чтобы использовать Lombok, вам нужно добавить его в качестве зависимости в свой проект. Если вы используете Spring Tool Suite, это простой вопрос, щелкнув правой кнопкой мыши на pom.xml-файл и выбор редактировать стартеры из опции контекстного меню Spring. Появится тот же выбор зависимостей, что и в главе 1 (на рис. 1.4), что дает вам возможность добавить или изменить выбранные зависимости. Найдите выбор Lombok, убедитесь, что он отмечен, и нажмите OK; Spring Tool Suite автоматически добавит его в вашу спецификацию сборки.

Кроме того, можно вручную добавить его со следующей записью в pom.xml:

<dependency>  
   <groupId>org.projectlombok</groupId>  
   <artifactId>lombok</artifactId>  
   <optional>true</optional>  
</dependency>

Эта зависимость предоставит вам аннотации Lombok (например, @Data) во время разработки и с автоматической генерацией методов во время выполнения. Но вам также потребуется добавить Lombok в качестве расширения в IDE, иначе IDE будет жаловаться на ошибки, связанные с отсутствующими методами и конечными свойствами, которые не задаются. Посетите https://projectlombok.org/ чтобы узнать, как установить Lombok в выбранной среде IDE.

Я думаю, что вы найдете Ломбок очень полезным, но знайте, что он необязателен. Он не нужен для разработки приложений Spring, поэтому если вы не хотите его использовать, не стесняйтесь писать эти недостающие методы вручную. Дерзайте... Я подожду.

2.1.2 Создание класса контроллера

Контроллеры являются основными компонентами в рамках Spring MVC. Их основная задача-обрабатывать HTTP-запросы и либо передать запрос в представление для отображения HTML (отображаемого браузером), либо записать данные непосредственно в тело ответа (RESTful). В этой главе мы сосредоточимся на типах контроллеров, использующих представления для создания содержимого для веб-браузеров. Когда мы перейдем к главе 6, мы рассмотрим написание контроллеров, которые обрабатывают запросы в REST API.

Для Taco Cloud, вам нужен простой контроллер который делает следующее:

* Обрабатывать HTTP-запросы GET, для пути запроса /design
* Составьте список ингредиентов
* Обработать запрос и данные по ингредиентам на основе шаблона для отображения как HTML и отправить в веб-браузер.

Следующий класс DesignTacoController отвечает этим требованиям

Листинг 2.2 Начальный класс Spring контроллера

package tacos.web;

import java.util.Arrays;  
import java.util.List;  
import java.util.stream.Collectors;  
import javax.validation.Valid;  
import org.springframework.stereotype.Controller;  
import org.springframework.ui.Model;  
import org.springframework.validation.Errors;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestMapping;  
import lombok.extern.slf4j.Slf4j;  
import tacos.Taco;  
import tacos.Ingredient;  
import tacos.Ingredient.Type;  
  
@Slf4j  
@Controller  
@RequestMapping("/design")  
public class DesignTacoController {  
  
   @GetMapping  
   public String showDesignForm(Model model) {  
      List<Ingredient> ingredients = Arrays.asList(  
         new Ingredient("FLTO", "Flour Tortilla", Type.WRAP),  
         new Ingredient("COTO", "Corn Tortilla", Type.WRAP),  
         new Ingredient("GRBF", "Ground Beef", Type.PROTEIN),  
         new Ingredient("CARN", "Carnitas", Type.PROTEIN),  
         new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES),  
         new Ingredient("LETC", "Lettuce", Type.VEGGIES),  
         new Ingredient("CHED", "Cheddar", Type.CHEESE),  
         new Ingredient("JACK", "Monterrey Jack", Type.CHEESE),  
         new Ingredient("SLSA", "Salsa", Type.SAUCE),  
         new Ingredient("SRCR", "Sour Cream", Type.SAUCE)  
      );  
      Type[] types = Ingredient.Type.values();  
      for (Type type : types) {  
         model.addAttribute(type.toString().toLowerCase(),  
         filterByType(ingredients, type));  
      }  
      model.addAttribute("design", new Taco());  
      return "design";  
   }

}

Первое, что следует отметить об DesignTacoController, - это набор аннотаций, применяемых на уровне класса. Первый, @Slf4j, является аннотацией, предоставляемой Lombok, которая во время выполнения автоматически генерирует SLF4J (Simple Logging Facade for Java, https://www.slf4j.org/) Logger в классе. Эта скромная аннотация имеет тот же эффект, что и при явном добавлении следующих строк в класс:

private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(DesignTacoController.class);

Вы будете использовать этот Logger немного позже.

Следующая аннотация, примененная к DesignTacoController это - @Controller. Эта аннотация служит, чтобы идентифицировать этот класс как контроллер и пометить его как кандидата на компонентное сканирование, так, чтобы Spring обнаружил его и автоматически создал экземпляр DesignTacoController как bean в контексте приложения Spring.

DesignTacoController аннотирован @RequestMapping. @RequestMapping аннотация, используемая на уровне класса, определяет тип запросов, обрабатываемых этим контроллером. В этом случае он указывает, что контроллер DesignTacoController будет обрабатывать запросы, путь к которым начинается с /design.

ОБРАБОТКА ЗАПРОСА GET

@GetMapping - относительно новая аннотация, появившаяся в Spring 4.3. До Spring 4.3 вы, возможно, использовали аннотацию @RequestMapping уровня метода вместо:

@RequestMapping(method=RequestMethod.GET)

Очевидно, что @GetMapping более сжатый и конкретный метод HTTP. @GetMapping является лишь одним членом семьи request-mapping аннотации. В таблице 2.1 перечислены все аннотации request-mapping, доступные в Spring MVC.

Таблица 2.1 Spring MVC request-mapping аннотации

|  |  |
| --- | --- |
| Аннотации | Описание |
| @RequestMapping | обработка запросов общего назначения |
| @GetMapping | обрабатывает запросы HTTP GET |
| @PostMapping | обрабатывает запросы HTTP POST |
| @PutMapping | обрабатывает запросы HTTP PUT |
| @DeleteMapping | обрабатывает HTTP-запросы на удаление |
| @Pathmapping | обрабатывае HTTP-запросов на патч |

Делайте правильные вещи легкими

Всегда рекомендуется быть как можно более конкретным при объявлении сопоставлений запросов в методах контроллера. По крайней мере, это означает объявление как пути (или наследование пути от класса уровня @RequestMapping), так и метода HTTP, который он будет обрабатывать.

Длинный @RequestMapping((method=RequestMethod.GET) сделал заманчивым взять ленивый выход и оставить атрибут метода. Благодаря новым аннотациям отображения Spring 4.3, правильная вещь также является легкой - с меньшим количеством ввода.

Новые аннотации request-mapping имеют те же атрибуты, что и @RequestMapping, поэтому их можно использовать везде, где пришлось бы использовать @RequestMapping.

Как правило, я предпочитаю использовать только @RequestMapping на уровне класса, чтобы указать базовый путь. Я использую более конкретные @GetMapping, @PostMapping и т. д. Для каждого из методов обработчика.

Теперь, когда вы знаете, что метод showDesignForm() будет обрабатывать запрос, давайте посмотрим на тело метода, чтобы увидеть, как он устроен. Основное - это то что метод создает список объектов Ingredient. Список пока жестко закодирован. Когда мы к этому вернемся в главе 3, вы получите список доступных ингредиентов тако из базы данных

Как только список ингредиентов готов, следующие несколько строк showDesignForm() фильтруют список по типу ингредиентов. Затем список типов ингредиентов добавляется в качестве атрибута к объекту Model, который передается в showDesignForm()). Model - объект, который передает данные между контроллером и любым представлением, отвечающим за визуализацию этих данных. В конечном счете данные, помещенные в атрибуты Model, копируются в атрибуты ответа сервлета, где их можно найти в представлении. Метод showDesignForm() завершается возвратом "design", логическим именем представления, которое будет использоваться для отображения модели в браузере.

Ваш DesignTacoController действительно начинает обретать форму. Если запустить приложение сейчас и указать в браузере путь /design, будет задействована функция showDesignForm() DesignTacoController, извлекающая данные из репозитория и помещающая их в модель перед передачей запроса в представление. Но поскольку вы еще не определили представление, запрос будет не корректен, что приведет к ошибке HTTP 404 (Not Found). Чтобы исправить это, давайте переключим наше внимание на представление, где данные будут представлены в виде HTML, которые будут отображаться в веб-браузере пользователя.

2.1.3 Проектирование представления

После того, как работы над контроллером закончены, пришло время для просмотра. Spring предлагает несколько отличных опций для определения представлений, включая JavaServer Pages (JSP), Thymeleaf, FreeMarker, Mustache и шаблоны на основе Groovy. На данный момент мы будем использовать Thymeleaf, выбор, который мы сделали в главе 1 при запуске проекта. Мы рассмотрим несколько других вариантов в разделе 2.5.

Чтобы использовать Thymeleaf, необходимо добавить еще одну зависимость в сборку проекта. Следующая запись <dependency> добавит Thymeleaf в Spring Boot, чтобы сделать Thymeleaf доступным для рендеринга представления, которое вы собираетесь создать:

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

Во время выполнения Spring Boot autoconfiguration увидит, что Thymeleaf находится в classpath и автоматически создаст bean, которые поддерживают представления Thymeleaf для Spring MVC.

Визуальные библиотеки, такие как Thymeleaf предназначены для отделения от определенного web  framework. Таким образом, они не знают об абстракции модели Spring и не могут работать с данными, которые контроллер помещает в Model. Но они могут работать с атрибутами запроса сервлета. Поэтому, прежде чем Spring передаст запрос представлению, он копирует данные модели в атрибуты запроса, к которым Thymeleaf и другие визуальные шаблонизаторы представлений имеют доступ.

Шаблоны Thymeleaf - это просто HTML с некоторыми дополнительными атрибутами элементов, которые корректируют шаблон при отображении данных запроса. Например, если бы существовал атрибут запроса, ключом которого является "message", и вы хотели, чтобы он отображался в теге HTML <p> с помощью Thymeleaf, вы бы написали следующее в шаблоне Thymeleaf:

<p th:text="${message}">placeholder message</p>

При отображении шаблона в HTML тело элемента <p> будет заменено значением атрибута запроса сервлета, ключом которого является"message". Атрибут th:text является атрибутом пространства имен Thymeleaf, который выполняет замену. Оператор ${} указывает ему использовать значение атрибута запроса (в данном случае" message").

Thymeleaf также предлагает другой атрибут, th: each, который перебирает коллекцию элементов, добавляя HTML код для каждого элемента в коллекции. Это пригодится, когда вы создадите отображение, чтобы перечислить ингредиенты taco из модели. Например, чтобы отобразить только список ингредиентов "wrap", можно использовать следующий фрагмент HTML:

<h3>Designate your wrap:</h3>  
<div th:each="ingredient : ${wrap}">  
   <input name="ingredients" type="checkbox" th:value="${ingredient.id}" />  
   <span th:text="${ingredient.name}">INGREDIENT</span><br/>  
</div>

Здесь используется атрибут th: each тега <div>, чтобы повторить отрисовку <div> один раз для каждого элемента коллекции, найденного в атрибуте wrap запроса. На каждой итерации элемент ингредиента привязывается к переменной Thymeleaf с именем ингредиента.

Внутри элемента <div> есть элемент checkbox <input> и элемент <span>, чтобы предоставить метку для флажка. Флажок использует Thymeleaf th:value, чтобы задать для <input> элемента значение атрибута value, найденное в свойстве ID компонента. Элемент <span> использует th:text для замены текста-заполнителя "INGREDIENT" значением свойства name ингредиента.

При визуализации с фактическими данными модели одна итерация цикла <div> может выглядеть следующим образом:

<div>  
   <input name="ingredients" type="checkbox" value="FLTO" />  
   <span>Flour Tortilla</span><br/>  
</div>

В конечном счете, предыдущий фрагмент Thymeleaf является лишь частью более крупной HTML-формы, через которую ваши творцы тако представят свои вкусные творения.  Полный шаблон Thymeleaf, включая все типы ингредиентов и форму, показан в следующем списке.

Листинг 2.3 Полный вид страницы дизайна тако

<!DOCTYPE html>  
<html xmlns="http://www.w3.org/1999/xhtml"  
 xmlns:th="http://www.thymeleaf.org">  
 <head>  
   <title>Taco Cloud</title>  
   <link rel="stylesheet" th:href="@{/styles.css}" />  
 </head>  
 <body>  
   <h1>Design your taco!</h1>  
   <img th:src="@{/images/TacoCloud.png}"/>  
   <form method="POST" th:object="${design}">  
     <div class="grid">  
       <div class="ingredient-group" id="wraps">  
         <h3>Designate your wrap:</h3>  
         <div th:each="ingredient : ${wrap}">  
           <input name="ingredients" type="checkbox" th:value="${ingredient.id}"/>  
           <span th:text="${ingredient.name}">INGREDIENT</span><br/>  
         </div>  
       </div>  
       <div class="ingredient-group" id="proteins">  
         <h3>Pick your protein:</h3>  
         <div th:each="ingredient : ${protein}">  
         <input name="ingredients" type="checkbox" th:value="${ingredient.id}"/>  
         <span th:text="${ingredient.name}">INGREDIENT</span><br/>  
       </div>  
     </div>  
     <div class="ingredient-group" id="cheeses">  
       <h3>Choose your cheese:</h3>  
       <div th:each="ingredient : ${cheese}">  
             <input name="ingredients" type="checkbox" th:value="${ingredient.id}"/>  
       <span th:text="${ingredient.name}">INGREDIENT</span><br/>  
       </div>  
   </div>  
   <div class="ingredient-group" id="veggies">  
     <h3>Determine your veggies:</h3>  
     <div th:each="ingredient : ${veggies}">  
      <input name="ingredients" type="checkbox" th:value="${ingredient.id}"/>  
      <span th:text="${ingredient.name}">INGREDIENT</span><br/>  
    </div>  
   </div>  
   <div class="ingredient-group" id="sauces">  
     <h3>Select your sauce:</h3>  
     <div th:each="ingredient : ${sauce}">  
       <input name="ingredients" type="checkbox" th:value="${ingredient.id}"/>  
       <span th:text="${ingredient.name}">INGREDIENT</span><br/>  
     </div>  
   </div>  
 </div>  
 <div>  
   <h3>Name your taco creation:</h3>  
   <input type="text" th:field="\*{name}"/>  
   <br/>  
   <button>Submit your taco</button>  
 </div>  
</form>  
</body>  
</html>

Как вы можете видеть, вы повторяете фрагмент <div> для каждого из типов ингредиентов. И вы включаете кнопку отправки и поле, где пользователь может назвать свое тако-творение.

Также стоит отметить, что полный шаблон включает в себя изображение логотипа Taco Cloud и ссылку на таблицу стилей(Содержание таблицы стилей не имеет отношения к нашему обсуждению; она содержит только стиль, чтобы представить ингредиенты в двух столбцах вместо одного длинного списка ингредиентов) в обоих случаях оператор @{} используется для создания относительного к контексту пути к статическим артефактам, на которые они ссылаются. Как вы узнали из главы 1, статическое содержимое в приложении Spring Boot подается из каталога /static в корне пути к классам.

Теперь, когда ваш контроллер и визуальная часть полностью готовы, вы можете запустить приложение, чтобы увидеть плоды вашего труда. Существует множество способов запуска приложения Spring Boot. В главе 1 я показал вам, как запустить приложение, сначала создав его в исполняемый файл JAR, а затем запустив JAR с java-jar. Я также показал, как запустить приложение непосредственно из сборки с помощью mvn spring-boot:run.

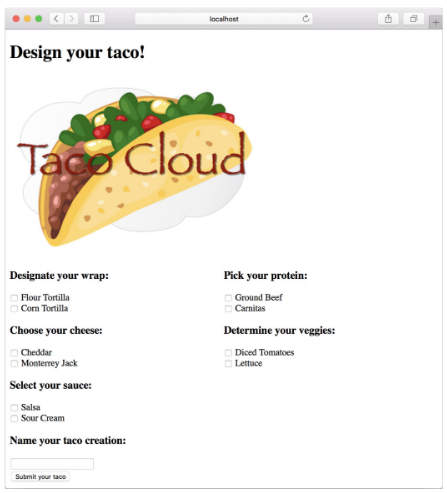
Независимо от того, как вы запустите приложение Taco Cloud, как только оно запустится, укажите браузеру http://localhost:8080/design. Вы должны увидеть страницу, которая выглядит примерно как на рисунке 2.2.  


Рисунок 2.2 Страница тако-создания

Это выглядит замечательно! Творцам тако, посетившим ваш сайт, предоставляется форма, содержащая палитру ингредиентов тако, из которых они могут создать свой шедевр. Но что происходит, когда они нажимают кнопку Submit Your Taco?

Ваш DesignTacoController еще не готов принять созданный тако. Если форма дизайна отправлена, пользователю будет представлена ошибка. (В частности, это будет ошибка HTTP 405: метод запроса "POST" не поддерживается.) Давайте исправим это, написав еще один код контроллера, который обрабатывает отправку формы.

2.2 Обработка отправки формы

Если еще раз взглянуть на тег <form> в представлении, можно увидеть, что его атрибут method имеет значение POST. Кроме того, <form> не объявляет атрибут action. Это означает, что когда форма будет отправлена, браузер соберет все данные в форме и отправит ее на сервер в запросе HTTP POST к тому же пути, для которого запрос GET отобразил форму—путь /design.

Таким образом необходим метод обработчика контроллера на принимающей стороне запроса POST. Вам нужно написать новый метод обработчика в контроллере дизайна тако, который обрабатывает запрос POST для /design.

В листинге 2.2 вы использовали аннотацию @GetMapping, чтобы указать, что метод showDesignForm() должен обрабатывать запросы HTTP GET для /design. Также как @GetMapping обрабатывает GET-запросы, вы можете использовать @PostMapping для обработки POST-запросов. Для обработки созданных тако добавьте метод processDesign() как в следующем листинге для DesignTacoController.

Листинг 2.4 Обработка POST-запросов с @PostMapping

@PostMapping public String processDesign(Design design) {  
   // Save the taco design…  
   // We'll do this in chapter 3  
   log.info("Processing design: " + design);  
   return "redirect:/orders/current";  
}

Применительно к методу processDesign(),  @PostMapping взаимодействует с уровнем класса @RequestMapping, чтобы указать, что processDesign() должен обрабатывать POST-запросы для /design. Это именно то, что вам нужно для обработки представленных творений создателя тако.

При отправке формы поля в форме привязываются к свойствам объекта Taco (класс которого показан в следующем листинге), который передается в качестве параметра в processDesign(). Метод processDesign() может делать с объектом Taco все, что угодно.

Листинг 2.5 Объект домена, определяющий состав тако

package tacos;

import java.util.List;  
import lombok.Data;  
  
@Data public class Taco {  
   private String name;  
   private List ingredients;  
}

Как вы можете видеть, Taco - это простой объект Java с несколькими свойствами. Как и ингредиент, класс Taco аннотируется @Data для автоматического создания необходимых методов JavaBean для вас во время выполнения.

Если вы посмотрите на форму в листинге 2.3, вы увидите несколько элементов checkbox, все с именем ingredients, и элемент ввода текста с именем name. Эти поля в форме соответствуют непосредственно ingredients и свойствам name класса Taco.

Поле Name в форме должно содержать только простое текстовое значение. Таким образом, свойство name Taco имеет тип String. У флажков "ингредиенты" также есть текстовые значения, но поскольку можно выбрать ни одного или несколько из них, свойство ingredients, к которому они привязаны, является List<String>, которое будет включать каждый из выбранных ингредиентов.

На данный момент метод processDesign() ничего не делает с объектом Taco. На самом деле, он вообще ничего не делает. Все нормально. В главе 3 вы добавите некоторую логику сохраняемости, которая сохранит отправленное Taco в базу данных.

Как и в случае с методом showDesignForm(),  processDesign() завершается возвращением String значения. И так же, как showDesignForm(), возвращаемое значение указывает на представление, которое будет показано пользователю. Но отличается то, что значение, возвращаемое из processDesign(), имеет префикс "redirect:", указывающий, что это представление перенаправления. Более конкретно, это указывает, что после того, как processDesign() завершится, браузер пользователя должен быть перенаправлен к относительному пути /order/current.

Идея заключается в том, что после создания тако, пользователь будет перенаправлен на форму заказа, из которых он может разместить заказ, чтобы его творение - тако был доставлен. Но у вас еще нет контроллера, который будет обрабатывать запрос /orders/current.

Учитывая то, что вы теперь знаете о @Controller, @RequestMapping и @GetMapping, вы можете легко создать такой контроллер. Это может выглядеть примерно так:

Листинг 2.6 Контроллер для представления формы заказа тако

package tacos.web;

import javax.validation.Valid;  
import org.springframework.stereotype.Controller;  
import org.springframework.ui.Model;  
import org.springframework.validation.Errors;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.RequestMapping;  
import lombok.extern.slf4j.Slf4j;  
import tacos.Order;  
  
@Slf4j @Controller  
@RequestMapping("/orders")  
public class OrderController {  
   @GetMapping("/current")  
   public String orderForm(Model model) {  
      model.addAttribute("order", new Order());  
      return "orderForm";  
   }  
}

Вы используете @Slf4j аннотацию Lombok для создания свободного объекта slf4j Logger во время выполнения. Вы будете использовать этот регистратор в момент получения деталей заказа, который был создан.

@RequestMapping уровня класса указывает, что любые методы обработки запросов в этом контроллере будут обрабатывать запросы, путь к которым начинается с /orders. В сочетании с методом уровня @GetMapping, он указывает, что метод orderForm() будет обрабатывать HTTP GET запросы для /orders/current.

Что касается самого метода orderForm(), он чрезвычайно прост, возвращая только логическое имя представления orderForm. Как только у вас будет способ сохранить созданного тако в базе данных в главе 3, вы вернетесь к этому методу и измените его, чтобы заполнить модель списком объектов Taco, которые будут размещены в очереди.

Представление формы заказа предоставляется шаблоном Thymeleaf с именем orderForm.html, который показан далее.

Листинг 2.7 Вид формы заказа тако

<!DOCTYPE html>  
<html xmlns="http://www.w3.org/1999/xhtml"  
xmlns:th="http://www.thymeleaf.org">  
<head>  
   <title>Taco Cloud</title>  
   <link rel="stylesheet" th:href="@{/styles.css}" />  
</head>  
<body>  
   <form method="POST" th:action="@{/orders}" th:object="${order}">  
      <h1>Order your taco creations!</h1>  
      <img th:src="@{/images/TacoCloud.png}"/>  
     <a th:href="@{/design}" id="another">Design another taco</a><br/>  
     <div th:if="${#fields.hasErrors()}">  
        <span class="validationError">  
           Please correct the problems below and resubmit.  
         </span>  
      </div>  
      <h3>Deliver my taco masterpieces to...</h3>  
      <label for="name">Name: </label>  
      <input type="text" th:field="\*{name}"/>  
      <br/>  
      <label for="street">Street address: </label>  
      <input type="text" th:field="\*{street}"/>  
      <br/>  
      <label for="city">City: </label>  
      <input type="text" th:field="\*{city}"/>  
      <br/>  
      <label for="state">State: </label>  
      <input type="text" th:field="\*{state}"/>  
      <br/>  
      <label for="zip">Zip code: </label>  
      <input type="text" th:field="\*{zip}"/>  
      <br/>  
      <h3>Here's how I'll pay...</h3>  
      <label for="ccNumber">Credit Card #: </label>  
      <input type="text" th:field="\*{ccNumber}"/>  
      <br/>  
      <label for="ccExpiration">Expiration: </label>  
      <input type="text" th:field="\*{ccExpiration}"/>  
      <br/>  
      <label for="ccCVV">CVV: </label>  
      <input type="text" th:field="\*{ccCVV}"/>  
      <br/>  
      <input type="submit" value="Submit order"/>  
   </form>  
</body>  
</html>

По большей части, orderForm.html отображение - это типичный HTML/Thymeleaf контент, с очень небольшим количеством заметок. Но обратите внимание, что тег <form> здесь отличается от тега <form>, используемого в листинге 2.3, тем, что он также определяет действие формы. Без указанного действия форма отправит запрос HTTP POST обратно на тот же URL-адрес, что и форма. Но здесь вы указываете, что форма должна заPOSTедь в /orders (используя оператор Thymeleaf @{...} для относительного контекста).

Поэтому в класс OrderController необходимо добавить еще один метод, обрабатывающий POST-запросы для /orders. У вас не будет способа сохранять заказы до следующей главы, поэтому поставим заглушку здесь-что-то вроде той, что вы видите в следующем листинге:

Листинг 2.8 обработка отправки заказа тако

@PostMapping  
public String processOrder(Order order) {  
   log.info("Order submitted: " + order);  
   return "redirect:/";  
}

При вызове метода processOrder() для обработки отправленного заказа ему присваивается объект Order, свойства которого привязаны к полям отправленной формы. Order, как и Taco, является довольно простым классом,который несет информацию о заказе.

Листинг 2.9 доменный объект для заказов тако

package tacos;

import javax.validation.constraints.Digits;  
import javax.validation.constraints.Pattern;  
import org.hibernate.validator.constraints.CreditCardNumber;  
import org.hibernate.validator.constraints.NotBlank;  
import lombok.Data;  
  
@Data  
public class Order {  
   private String name;  
   private String street;  
   private String city;  
   private String state;  
   private String zip;  
   private String ccNumber;  
   private String ccExpiration;  
   private String ccCVV;  
}

Теперь, когда вы разработали OrderController и представление формы заказа, вы готовы испытать его. Откройте браузер, с адресом http://localhost:8080/design, выберите некоторые ингредиенты для вашего тако,и нажмите кнопку Submit Your Taco. Вы должны увидеть форму, подобную показанной на рис. 2.3.

Заполните некоторые поля формы и нажмите кнопку Submit Order. И следите за логами приложений, чтобы увидеть информацию о вашем заказе. Когда я нажал, запись лога выглядела примерно так (переформатирована, чтобы соответствовать ширине этой страницы):

Order submitted: Order(name=Craig Walls,street1=1234 7th Street, city=Somewhere, state=Who knows?, zip=zipzap, ccNumber=Who can guess?, ccExpiration=Some day, ccCVV=See-vee-vee)

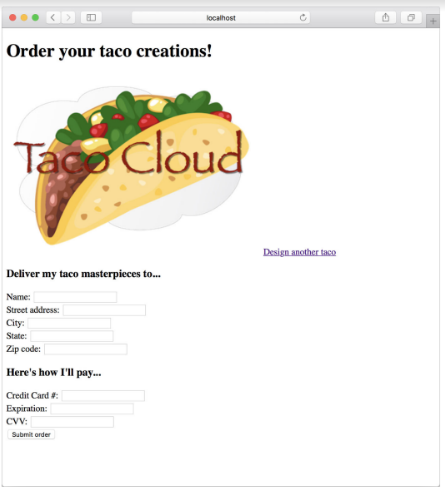
Если вы внимательно посмотрите на запись лога из моего тестового заказа, вы увидите, что хотя метод processOrder() выполнил свою работу и обработал отправку формы, он позволил пройти немного не корректной информации. Большинство полей в форме содержат данные, которые не могут быть правильными. Давайте добавим некоторую проверку, чтобы убедиться, что предоставленные данные по крайней мере похожи на требуемую информацию.  


Рисунок 2.3

2.3 Проверка формы ввода

При сборке созданного тако, что делать, если пользователь не выбирает ингредиенты или не указал имя при создании? При отправке заказа, что делать, если создатель тако не заполнил необходимые поля адреса? Или что, если он ввел значение в поле кредитной карты, которое не является действительным номером кредитной карты?

В настоящее время ничто не помешает пользователю создать тако без ингредиентов или с пустым адресом доставки, или даже предоставить текст своей любимой песни в качестве номера кредитной карты. Это потому что вы еще не определили, как эти поля должны быть проверены.

Одним из способов выполнения проверки формы является заполнение методов processDesign() и processOrder() множеством блоков if/then, проверяя каждое поле, чтобы убедиться, что оно соответствует соответствующим правилам проверки. Но это было бы громоздко и трудно читать и отлаживать.

К счастью, Spring поддерживает API проверки компонентов Java (также известный как JSR-303; https://jcp.org/en/jsr/detail?id=303). Это упрощает объявление правил проверки в отличие от явного написания логики объявления в коде приложения. И с Spring Boot, вам не нужно делать ничего особенного, чтобы добавить библиотеки проверки в свой проект, потому что API проверки и реализация Hibernate API проверки автоматически добавляются в проект в качестве временных зависимостей веб-стартера Spring Boot.

Чтобы применить проверку в Spring MVC, необходимо:

* Объявите правила проверки классам, которые должны быть проверены: в частности, класс Taco.
* Указать, что проверка должна быть выполнена в методах контроллера, которые требуют проверки: в частности, метод processDesign в DesignTacoController() и в OrderController метод processOrder().
* Измените представления формы для отображения ошибок проверки.

API проверки предлагает несколько аннотаций, которые можно поместить в свойства объектов домена для объявления правил проверки. Реализация Hibernate API проверки добавляет еще больше аннотаций проверки. Давайте посмотрим, как вы можете применить некоторые из этих аннотаций для проверки представленного Taco или Order.

2.3.1 Определение правил проверки

Для класса Taco необходимо убедиться, что свойство name не пустое и не null и что в списке выбранных ингредиентов есть хотя бы один элемент. В следующем списке показан обновленный класс Taco, который использует @NotNull и @Size для объявления этих правил проверки.

Листинг 2.10 Добавленные проверки в класс Taco

package tacos;  
import java.util.List;  
import javax.validation.constraints.NotNull;  
import javax.validation.constraints.Size; import lombok.Data;  
  
@Data  
public class Taco {  
 @NotNull  
 @Size(min=5, message="Name must be at least 5 characters long")  
 private String name;  
  
 @Size(min=1, message="You must choose at least 1 ingredient")  
 private List<String> ingredients;  
}

Вы заметите, что в дополнение к требованию, чтобы свойство name не было null, вы объявляете, что оно должно иметь значение длиной не менее 5 символов.

Когда дело доходит до объявления проверки представленных заказов тако, необходимо применить аннотации к классу Order.  Для свойств адреса необходимо только убедиться, что пользователь не оставляет пустыми ни одно из полей. Для этого вы будете использовать аннотацию Hibernate Validator @NotBlank.

Однако проверка полей платежей немного более экзотична.  Необходимо не только убедиться, что свойство ccNumber не пусто, но и что оно содержит значение, которое может быть допустимым номером кредитной карты. Свойство ccExpiration должно соответствовать формату MM/YY (двухзначные месяц и год). И свойство ccCVV должно быть трехзначным. Для достижения такого рода проверки необходимо использовать несколько других аннотаций Java Bean Validation API и заимствовать аннотацию проверки из коллекции аннотаций Hibernate Validator.  В следующем листинге показаны изменения, необходимые для проверки класса Order.

Листинг 2.11 Проверка полей заказа

package tacos;

import javax.validation.constraints.Digits;  
import javax.validation.constraints.Pattern;  
import org.hibernate.validator.constraints.CreditCardNumber;  
import javax.validation.constraints.NotBlank;  
import lombok.Data;  
  
@Data  
public class Order {  
 @NotBlank(message="Name is required")  
 private String name;  
  
 @NotBlank(message="Street is required")  
 private String street;

 @NotBlank(message="City is required")  
 private String city;  
  
 @NotBlank(message="State is required")  
 private String state;  
  
 @NotBlank(message="Zip code is required")  
 private String zip;  
  
 @CreditCardNumber(message="Not a valid credit card number")  
 private String ccNumber;  
  
 @Pattern(regexp="^(0[1-9]|1[0-2])([\\/])([1-9][0-9])$", message="Must be formatted MM/YY")  
 private String ccExpiration;  
  
 @Digits(integer=3, fraction=0, message="Invalid CVV")  
  private String ccCVV;  
}

Как видите, свойство ccNumber  снабжено аннотацией @CreditCardNumber.  Эта аннотация объявляет, что значение свойства должно быть допустимым номером кредитной карты, который проходит проверку алгоритма Luhn (https://en.wikipedia.org/wiki/Luhn\_algorithm). Это предотвращает ошибки пользователей и преднамеренно неверные данные, но не гарантирует, что номер кредитной карты фактически принадлежит учетной записи или что номер карты пригоден для оплаты.

К сожалению, нет готовой аннотации для проверки формата MM/YY свойства ccExpiration. Я применил аннотацию @Pattern, предоставив ей регулярное выражение, которое гарантирует, что значение свойства придерживается желаемого формата. Если вам интересно, как расшифровать регулярное выражение, я рекомендую вам ознакомиться со многими онлайн-руководствами по регулярным выражениям, в том числе http://www.regular-expressions.info/. Синтаксис регулярных выражений является темным искусством и, безусловно, выходит за рамки этой книги.

Наконец, свойство ccCVV аннотируется @Digits, чтобы убедиться, что значение содержит ровно три цифры.

Все аннотации проверки содержат атрибут message, определяющий сообщение, которое будет отображаться пользователю, если введенная им информация не соответствует требованиям объявленных правил проверки.

2.3.2 Выполнение проверок

Теперь, когда вы объявили, как Taco и Order должны быть проверены, нам нужно пересмотреть каждый из контроллеров, указав, что проверка должна быть выполнена, когда формы будут отправлены в их соответствующие методы обработчика.

Чтобы проверить отправленное Taco, необходимо добавить аннотацию Java Bean Validation API’s @Valid

к аргументу Taco метода processDesign() у DesignTacoController.

Листинг 2.12 Проверка POST-запроса Taco

@PostMapping  
public String processDesign(@Valid Taco design, Errors errors) {  
 if (errors.hasErrors()) {  
   return "design";  
 }  
 // Save the taco design...  
 // We'll do this in chapter 3  
 log.info("Processing design: " + design);  
 return "redirect:/orders/current";  
}

@Valid аннотация говорит Spring MVC, чтобы выполнить проверку на представленном объекте Taco после того, как он привязан к представленным данным формы и перед вызовом метода processDesign(). Если есть какие-либо ошибки проверки, сведения об этих ошибках будут записаны в объект Errors, который передается в processDesign(). Первые несколько строк processDesign() проверяют объект Errors, через его метод hasErrors (), есть ли какие-либо ошибки проверки. Если есть, метод завершается без обработки Taco и возвращает имя представления "design" так, чтобы форма была перерисована.

Для выполнения проверки переданных объектов Order, аналогичные изменения также требуются в методе processOrder() в OrderController.

Листинг 2.13 Проверка -POST-запроса Order

@PostMapping  
public String processOrder(@Valid Order order, Errors errors) {  
 if (errors.hasErrors()) {  
   return "orderForm";  
 }  
 log.info("Order submitted: " + order);  
 return "redirect:/";  
}

В обоих случаях методу будет разрешено обрабатывать отправленные данные при отсутствии ошибок проверки.  Если есть ошибки проверки, запрос будет перенаправлен в представление формы, чтобы дать пользователю возможность исправить свои ошибки.

Но как пользователь узнает, какие ошибки требуют исправления? Если вы не отобразите ошибки в форме, пользователю останется только гадать о том, как успешно отправить форму.

2.3.3 Отображение ошибок проверки

Thymeleaf предлагает удобный доступ к объекту Errors через свойство fields и с его атрибутом th:errors. Например, чтобы отобразить ошибки проверки в поле номер кредитной карты, можно добавить элемент <span>, который использует эти ссылки на ошибки в шаблоне формы заказа следующим образом.

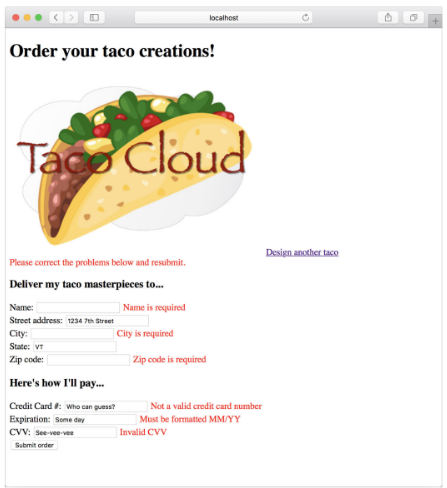
Листинг 2.14 Отображение ошибок проверки

<label for="ccNumber">Credit Card #: </label>  
<input type="text" th:field="\*{ccNumber}"/>  
<span class="validationError"  
 th:if="${#fields.hasErrors('ccNumber')}"  
 th:errors="\*{ccNumber}">CC Num Error</span>

Помимо атрибута класса, который можно использовать для стилизации ошибки, чтобы привлечь внимание пользователя, элемент <span> использует атрибут th:if, чтобы решить, отображать ли <span>. Метод hasErrors() свойства fields проверяет наличие ошибок в поле ccNumber. Если да, то <span> будет отрисован.

Атрибут th: errors ссылается на поле ccNumber и, если для этого поля имеются ошибки, он заменит содержимое заполнителя элемента <span> сообщением о результате проверки.

Если вы добавите похожие теги <span> для других полей, вы увидите форму, которая выглядит как на рисунке 2.4, когда вы предоставляете недопустимую информацию. Ошибки указывают, что поля name, city и ZIP code оставлены пустыми и что все поля платежа не соответствуют критериям проверки.

  
Рисунок 2.4 Отображение ошибок проверки в форме заказа

Теперь контроллеры Taco Cloud не только отображают и записывают входные данные, но и проверяют соответствие информации некоторым основным правилам проверки. Давайте вернемся назад и пересмотрим HomeController из главы 1, глядя на альтернативную реализацию.

2.4 Работа с контроллерами отображения

До сих пор вы написали три контроллера для приложения Taco  Cloud. Хотя каждый контроллер служит определенной цели в функциональности приложения, все они в значительной степени следуют одной модели программирования:

* Они все аннотированы @Controller, чтобы указать, что они классы контроллеры, которые должны автоматически обнаруживаться при сканировании компонентов Spring и создаваться как bean в контексте приложения Spring.
* Все, кроме HomeController, аннотируются @RequestMapping на уровне класса, чтобы определить базовый шаблон запроса, который будет обрабатывать контроллер.
* Все они имеют один или несколько методов, аннотированных @GetMapping или@PostMapping, чтобы предоставить сведения о том, какие методы должны обрабатывать какие типы запросов.

Большинство контроллеров, которые вы напишете, будут следовать этому шаблону.  Но когда контроллер достаточно прост, что он не заполняет модель или процесс ввода-как в случае с вашим HomeController-есть еще один способ, которым вы можете определить контроллер. Посмотрите на следующий листинг, чтобы увидеть, как можно объявить контроллер представления - контроллер, который ничего не делает, но перенаправляет запрос к представлению.

Листинг 2.15 Объявление контроллера отображения

package tacos.web;  
  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;  
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;  
  
@Configuration  
public class WebConfig implements WebMvcConfigurer {  
  
 @Override  
 public void addViewControllers(ViewControllerRegistry registry) {  
   registry.addViewController("/").setViewName("home");  
 }  
}

Самое важное о WebConfig, то что он реализует интерфейс WebMvcConfigurer. WebMvcConfigurer определяет несколько методов для настройки Spring MVC. Несмотря на то, что это интерфейс, он обеспечивает реализацию всех методов по умолчанию, поэтому вам нужно переопределить только необходимые методы. В этом листинге переопределен метод addViewControllers().

Метод addViewControllers() получает ViewControllerRegistry, который можно использовать для регистрации одного или нескольких контроллеров представления. Здесь вы вызываете addViewController() в реестре, передавая "/", путь, по которому ваш контроллер представлений будет обрабатывать запросы GET.  Этот метод возвращает объект регистрации контроллера представлений, на котором вы немедленно вызываете setViewName(), чтобы указать home в качестве представления, на которое должен быть перенаправлен запрос "/".

И вот так просто, вы смогли заменить HomeController несколькими строками в классе конфигурации. Теперь вы можете удалить HomeController, и приложение должно вести себя так же, как и раньше. Единственное другое изменение, необходимое для пересмотра HomeControllerTest из главы 1, удаление ссылки на HomeController из аннотации @WebMvcTest, так чтобы тестовый класс будет компилироваться без ошибок.

Здесь вы создали новый класс конфигурации WebConfig для размещения объявления контроллера представления. Но любой класс конфигурации может реализовать WebMvcConfigurer и переопределить метод addViewController. Например, вы могли бы добавить такое же объявление контроллера представления в начальный класс приложения Taco Cloud, например так:

@SpringBootApplication  
public class TacoCloudApplication implements WebMvcConfigurer {  
 public static void main(String[] args) {  
    SpringApplication.run(TacoCloudApplication.class, args);  
 }  
  
 @Override  
 public void addViewControllers(ViewControllerRegistry registry) {  
   registry.addViewController("/").setViewName("home");  
 }  
}

Расширяя существующий класс конфигурации, можно избежать создания нового класса конфигурации, сохраняя обратное взаимодействие артефактов проекта. Но я предпочитаю создавать новый класс конфигурации для каждого типа конфигурации (веб, данные, безопасность и т. д.), сохраняя конфигурацию начальной загрузки приложения чистой и простой.

Говоря о контроллерах представлений и более общих представлениях, на которые контроллеры перенаправляют запросы, до сих пор вы использовали Thymeleaf для всех своих представлений.  Мне нравится Thymeleaf, но, может, вы предпочитаете другой шаблонизатор для отображения данных приложения. Давайте посмотрим на другие шаблонизаторы поддерживаемые Spring.

2.5 Выбор библиотеки шаблонов представлений

В большинстве случаев выбор библиотеки шаблонов представлений зависит от личных предпочтений. Spring очень гибка и поддерживает множество распространенных вариантов шаблонов.  За некоторыми небольшими исключениями, библиотека шаблонов, которую вы выберете, сама не будет иметь представления о том, что она даже работает с Spring (одним из таких исключений является -Spring Security dialect Thymeleaf, о котором мы поговорим в главе 4).

Таблица 2.2 Каталог опции шаблона, поддерживаемые автоконфигурацией Spring Boot.

|  |  |
| --- | --- |
| Шаблоны | Spring Boot starter зависимости |
| FreeMarker | spring-boot-starter-freemarker |
| Groovy Templates | spring-boot-starter-groovy-templates |
| JavaServer Pages (JSP) | Нет (предоставлено Tomcat или Jetty) |
| Mustache | spring-boot-starter-mustache |
| Thymeleaf | spring-boot-starter-thymeleaf |

Вообще говоря, вы выбираете нужную библиотеку шаблонов представления, добавляете ее в качестве зависимости в сборку и начинаете писать шаблоны в каталоге /templates (в каталоге src/main/resources в проекте, построенном на Maven или Gradle). Spring Boot обнаружит выбранную библиотеку шаблонов и автоматически настроит компоненты, необходимые для обслуживания представлений контроллеров Spring MVC.

Вы уже сделали это с Thymeleaf для приложения Taco Cloud. В главе 1 при инициализации проекта был установлен флажок Thymeleaf. Это привело к Spring Boot Thymeleaf starter-у в файле pom.xml. При запуске приложения автоконфигурация Spring Boot обнаруживает наличие Thymeleaf и автоматически настраивает bean-ы Thymeleaf для вас. Все, что вам нужно было сделать, это начать писать шаблоны в /templates.

Если вы предпочитаете использовать другую библиотеку шаблонов, просто выберите ее при инициализации проекта или отредактируйте существующую сборку проекта, чтобы включить новую выбранную библиотеку шаблонов.

Например, скажем, вы захотели использовать Mustache вместо Thymeleaf. Не проблема. Просто откройте в проекте pom.xml-файл и замените объявление,

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

на:

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

Конечно, вам нужно будет убедиться, что вы пишете все шаблоны с синтаксисом Mustache вместо тегов Thymeleaf. Особенности работы с Mustache(или любой из вариантов языка шаблона) выходят далеко за рамки этой книги, но чтобы дать вам представление о том, чего ожидать, вот фрагмент из шаблона Mustache, который будет отображать одну из групп ингредиентов в форме дизайна taco:

<h3>Designate your wrap:</h3>  
{{#wrap}}  
<div>  
 <input name="ingredients" type="checkbox" value="{{id}}" />  
 <span>{{name}}</span><br/>  
</div>  
{{/wrap}}

Это Mustache эквивалент фрагмента Thymeleaf в разделе 2.1.3.  Блок {{#wrap}} (который завершается {{/wrap}}) перебирает коллекцию в атрибуте запроса, ключом которого является wrap, и отображает встроенный HTML для каждого элемента. Теги {{{id}} и {{name}} ссылаются на свойства id и name элемента (который должен быть Ingredient).

Вы заметили в таблице 2.2, что JSP не требует каких-либо специальных зависимостей в сборке.  Это потому, что сам контейнер сервлета (Tomcat по умолчанию) реализует спецификацию JSP, таким образом не требуя никаких дополнительных зависимостей.

Но есть проблема, если вы решите использовать JSP. Как оказалось, контейнеры сервлетов Java, включая встроенные контейнеры Tomcat и Jetty, обычно ищут JSP где—то в /WEB-INF. Но если вы создаете приложение в виде исполняемого файла JAR, нет способа удовлетворить это требование.  Таким образом JSP является вариантом только, если вы создаете приложение в виде файла WAR и развертывания его в контейнере традиционного сервлета. При создании исполняемого файла JAR необходимо выбрать Thymeleaf, FreeMarker или один из других вариантов из таблицы 2.2.

2.5.1 Кэширование шаблонов

По умолчанию шаблоны анализируются только один раз при первом использовании и результаты этого анализа кэшируются для последующего использования. Это отличная функция для производительности, так как она предотвращает избыточный парсинг шаблонов по каждому запросу и, таким образом, повышает производительность.

Однако эта функция не так полезна во время разработки. Предположим, вы запускаете свое приложение и нажимаете страницу создания тако и решаете внести в нее несколько изменений. При обновлении веб-браузера вам все равно будет показана исходная версия. Единственный способ увидеть изменения-перезапустить приложение, что довольно неудобно.

К счастью, есть способ отключить кэширование. Все, что вам нужно сделать, это установить свойство кэширования, соответствующее шаблону, в false. В таблице 2.3 перечислены свойства кэширования для каждого из поддерживаемых библиотек шаблонов.

Таблица 2.3 Свойства включения/отключения кэширования шаблонов

|  |  |
| --- | --- |
| Шаблоны | Свойство включения кэширования |
| FreeMarker | spring.freemarker.cache |
| Groovy Templates | spring.groovy.template.cache |
| Mustache | spring.mustache.cache |
| Thymeleaf | spring.thymeleaf.cache |

По умолчанию, все эти свойства имеют значение true, чтобы включить кэширование.  Можно отключить кэширование для выбранного шаблонизатора, установив для его свойства cache значение false.  Например, чтобы отключить кэширование Thymeleaf, добавьте следующую строку в application.properties:

spring.thymeleaf.cache=false

Единственная загвоздка заключается в том, что перед развертыванием приложения в рабочей среде необходимо удалить эту строку (или задать для нее значение true). Одним из вариантов является установка свойства в профиле. (Мы поговорим о профилях в главе 5.)

Гораздо более простой вариант-использовать DevTools Spring Boot, как мы решили сделать в главе 1. Среди множества полезных функций, предлагаемых DevTools во время разработки, он отключит кэширование для всех библиотек шаблонов, но отключит себя (и, следовательно, включит кэширование шаблонов) при развертывании приложения.

ИТОГ

* Spring предлагает мощный веб-фреймворк под названием Spring MVC, который может быть использован для разработки веб-интерфейса для Spring-приложения.
* Spring MVC - базовые аннотации, позволяющие создать методы обработки запросов с аннотациями, такие как @RequestMapping, @GetMapping, и @PostMapping.
* Большинство методов обработки запросов завершаются возвращением логического имени представления, например шаблона Thymeleaf, на который передается запрос (вместе с любыми данными модели).
* Spring MVC поддерживает Java Bean Validation API и реализаций Validation API такие как Hibernate Validator.
* View контроллеры могут быть использованы для обработки HTTP-запросов GET, для которых не требуется модели данных или обработки.
* В дополнение к Thymeleaf Spring поддерживает различные параметры просмотра, включая FreeMarker, Groovy Templates и Mustache.

[Spring in Action Covers Spring 5.0 перевод на русский. Глава 3](https://beginner-mercenary.blogspot.com/2019/04/spring-in-action-covers-spring-50-3.html)

3. Работа с данными

*Эта глава охватывает:*

* *Используя Spring JdbcTemplate*
* *Вставка данных посредством SimpleJdbcInsert*
* *Объявление -JPA - репозиториев с данными Spring*

Большинство приложений предлагают больше, чем просто красивый интерфейс. Хотя пользовательский интерфейс может обеспечить взаимодействие с приложением, именно данные, которые он представляет и хранит, отличают приложения от статических веб-сайтов.

В приложении Taco Cloud необходимо поддерживать информацию об ингредиентах, тако и заказах. Без базы данных для хранения этой информации, приложение не сможет развиваться дальше, чем то, что вы разработали в главе 2.

В этой главе вы добавите сохраняемость данных в приложение Taco Cloud. Вы начнете с использования поддержки Spring для JDBC (Java Database Connectivity) для устранения шаблонного кода.  Затем вы переработаете репозитории данных для работы с JPA (Java Persistence API), исключив еще больше кода.

3.1 Чтение и запись данных с помощью JDBC

На протяжении десятилетий реляционные базы данных и SQL занимали лидирующие позиции в области сохранения данных. Несмотря на то, что в последние годы появилось много альтернативных типов баз данных, реляционная база данных по-прежнему является лучшим выбором для хранилища данных общего назначения и вряд ли будет узурпирована в ближайшее время.

Когда дело доходит до работы с реляционными данными, у разработчиков Java есть несколько вариантов. Два наиболее распространенных варианта это JDBC и JPA. Spring поддерживает оба из них с абстракциями, что делает работу с JDBC или JPA проще, чем было бы без Spring. В этом разделе мы сосредоточимся на том, как Spring поддерживает JDBC, а затем рассмотрим поддержку Spring для JPA в разделе 3.2.

Поддержка Spring JDBC внедрена в класс JdbcTemplate. JdbcTemplate предоставляет средство, с помощью которого разработчики могут выполнять операции SQL в реляционной базе данных без всех церемоний и шаблонов которые обычно требуется при работе с JDBC.

Чтобы понять, для чего нужен JdbcTemplate, давайте посмотрим на примере, как выполнить простой запрос в Java без JdbcTemplate.

листинг 3.1 Запрос к базе данных без JdbcTemplate

@Override  
public Ingredient findOne(String id) {  
 Connection connection = null;  
 PreparedStatement statement = null;  
 ResultSet resultSet = null;  
 try {  
   connection = dataSource.getConnection();  
   statement = connection.prepareStatement(  
     "select id, name, type from Ingredient");  
   statement.setString(1, id);  
   resultSet = statement.executeQuery();  
   Ingredient ingredient = null;  
   if(resultSet.next()) {  
    ingredient = new Ingredient(  
       resultSet.getString("id"),  
       resultSet.getString("name"),  
       Ingredient.Type.valueOf(resultSet.getString("type")));  
   }  
   return ingredient;  
 } catch (SQLException e) {  
 // ??? Что нужно сделать в этом случае ???  
 } finally {  
   if (resultSet != null) {  
     try {  
       resultSet.close();  
     } catch (SQLException e) {}  
   }  
   if (statement != null) {  
     try {  
       statement.close();  
     } catch (SQLException e) {}  
   }  
   if (connection != null) {  
     try {  
       connection.close();  
    } catch (SQLException e) {}  
   }  
 }  
 return null;  
}

Уверяю вас, что где-то в листинге 3.1 есть пара строк, которые запрашивают в базе данных ингредиенты. Но бьюсь об заклад, вам было трудно обнаружить иголку запроса в стоге сена JDBC. Он окружен кодом, который создает соединение, создает оператор и очищает, закрывая соединение, оператор и результирующий набор.

Что еще хуже, что угодно всего может пойти не так при создании соединения или инструкции, или при выполнении запроса.  Это требует, чтобы вы поймали SQLException, который может или не может быть полезен в выяснении того, что пошло не так или как решить проблему.

SQLException является проверяемым исключением, которое требует обработки в блоке catch. Но наиболее распространенные проблемы, такие как неспособность создать соединение с базой данных или неправильно набранный запрос, не могут быть решены в блоке catch и, вероятно, будут перепрофилированы для обработки в восходящем направлении. Теперь посмотрим чем отличается код с использованием JdbcTemplate.

листинг 3.2 Запрос к базе данных с JdbcTemplate

private JdbcTemplate jdbc;  
  
@Override  
public Ingredient findOne(String id) {  
 return jdbc.queryForObject(  
   "select id, name, type from Ingredient where id=?",  
   this::mapRowToIngredient, id);  
 }  
  
private Ingredient mapRowToIngredient(ResultSet rs, int rowNum)  
 throws SQLException {  
   return new Ingredient(  
     rs.getString("id"),  
     rs.getString("name"),  
     Ingredient.Type.valueOf(rs.getString("type")));  
}

Код в листинге 3.2 явно намного проще, чем исходный пример JDBC в листинге 3.1; никаких инструкций или соединений не создается. И, после того, как метод закончен, нет никакой очистки объектов. Наконец, нет никакой обработки исключений, которые не могут быть должным образом обработаны в блоке catch. То, что осталось - это код, который сосредоточился исключительно на выполнении запроса (вызова Jdbctemplate метода queryForObject()) и отображение результатов в объекте ингредиент (в mapRowToIngredient() методе).

Код в листинге 3.2 представляет собой фрагмент того, что необходимо сделать для использования JdbcTemplate для сохранения и чтения данных в приложении Taco Cloud. Давайте предпримем следующие шаги, необходимые для улучшения приложения с сохранением JDBC. Начнем с нескольких настроек объектов домена.

3.1.1 адаптация домена для сохранения.

При сохранении объектов в базе данных обычно рекомендуется иметь одно поле, которое однозначно идентифицирует объект. В вашем классе ингредиентов уже есть поле id, но вам нужно добавить поля id как в Taco, так и в Order.

Кроме того, может быть полезно знать, когда создан Taco и когда создан Order. Кроме того, к каждому объекту необходимо добавить поле для записи даты и времени сохранения объектов.  В следующем списке показаны новые поля id и createdAt, необходимые в классе Taco.

Листинг 3.3 Добавление ID и поля времени в класс Taco

@Data  
public class Taco {  
 private Long id;  
 private Date createdAt;  
 ...  
}

Поскольку Lombok используется для автоматического создания методов доступа во время выполнения, нет необходимости делать что-либо большее, чем объявлять свойства id и createdAt. Они будут иметь соответствующие методы getter и setter по мере необходимости во время выполнения. Подобные изменения требуются и в классе Order, как показано здесь:

@Data  
public class Order {  
 private Long id;  
 private Date placedAt;  
 ...  
}

Опять же, Lombok автоматически генерирует методы метода доступа, поэтому это единственные изменения, необходимые в Order. (Если по какой-то причине вы решили не использовать Lombok, вам нужно будет написать эти методы самостоятельно.)

Классы домена теперь готовы к сохранению. Давайте посмотрим, как использовать JdbcTemplate для чтения и записи в базу данных.

3.1.2 Работа с JdbcTemplate

Перед началом использования JdbcTemplate необходимо добавить его в путь к классам проекта. Это можно легко сделать, добавив в сборку зависимость JDBC starter от Spring Boot:

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

Вам также понадобится база данных, в которой будут храниться ваши данные. Для наших целей, встроенная база данных будет идеальна. Я предпочитаю встроенную базу данных H2, поэтому я добавил следующую зависимость в сборку:

<dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>runtime</scope>  
</dependency>

Позже, вы увидите, как настроить приложение для использования внешней базы данных. Но сейчас, давайте перейдем к написанию репозитория, который извлекает и сохраняет данные Ingredient.

ОПРЕДЕЛЕНИЕ РЕПОЗИТОРИЕВ JDBC

Репозиторий Ingredient должен выполнять следующие операции:

* Query запрос всех ингредиентов в коллекцию Ingredient объектов
* Запрос одного Ingredient по его id
* Сохранить объект Ingredient

Следующий интерфейс IngredientRepository  определяет эти три операции объявляя методы:

package tacos.data;  
  
import tacos.Ingredient;  
public interface IngredientRepository {  
 Iterable<Ingredient> findAll();  
 Ingredient findOne(String id);  
 Ingredient save(Ingredient ingredient);  
}

Несмотря на то, что интерфейс отражает суть того, что вам нужно сделать в репозитории ингредиентов, вам все равно нужно написать реализацию IngredientRepository, использующую JdbcTemplate для запроса к базе данных. Код, показанный далее, является первым шагом в написании этой реализации.

листинг 3.4 -Начало репозиторий ингредиент с JdbcTemplate

package tacos.data;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.jdbc.core.RowMapper;  
import org.springframework.stereotype.Repository;  
import tacos.Ingredient;  
  
@Repository  
public class JdbcIngredientRepository implements IngredientRepository {  
 private JdbcTemplate jdbc;  
  
 @Autowired  
 public JdbcIngredientRepository(JdbcTemplate jdbc) {  
   this.jdbc = jdbc;  
 }  
 ...  
}

Как вы можете видеть, JdbcIngredientRepository аннотируется @Repository. Эта аннотация является одной из немногих аннотаций стереотипов, которые определяет Spring, включая @Controller и @Component. Аннотируя JdbcIngredientRepository с помощью @Repository, вы объявляете, что он должен быть автоматически обнаружен при сканировании компонентов Spring и создан как bean в контексте приложения Spring.

Когда Spring создает bean JdbcIngredientRepository, он внедряет  JdbcTemplate через аннотированную конструкцию @Autowired. Конструктор назначает JdbcTemplate переменной экземпляра, которая будет использоваться в других методах для запроса и вставки в базу данных. Говоря об остальных методах, давайте взглянем на реализации findAll () и findById ().

Листинг 3.5 запросы к базе данных с JdbcTemplate

@Override  
public Iterable<Ingredient> findAll() {  
 return jdbc.query("select id, name, type from Ingredient",  
   this::mapRowToIngredient);  
}  
  
@Override  
public Ingredient findOne(String id) {  
 return jdbc.queryForObject(  
   "select id, name, type from Ingredient where id=?",  
   this::mapRowToIngredient, id);  
}  
  
private Ingredient mapRowToIngredient(ResultSet rs, int rowNum) throws QLException {  
 return new Ingredient(  
   rs.getString("id"),  
   rs.getString("name"),  
   Ingredient.Type.valueOf(rs.getString("type")));  
}

findAll() и findById() используют JdbcTemplate аналогичным образом. Метод findAll(), ожидающий возврата коллекции объектов, использует JdbcTemplate метод запроса. Метод query() принимает SQL для запроса, а также реализацию Spring-овского RowMapper с целью сопоставления каждой строки в результирующем наборе объекту. findAll() также принимает в качестве окончательного аргумента (аргументов) список любых параметров, необходимых в запросе. Но, в данном случае, нет никаких обязательных параметров.

Метод findById() возвращает только один объект ингредиента, поэтому он использует метод queryForObject() JdbcTemplate вместо query(). queryForObject() работает так же, как query(), за исключением того, что возвращает один объект вместо списка объектов. В этом случае ему дается запрос для выполнения, RowMapper и id Ingredient для выборки, который используется вместо ? в запросе.

Как показано в листинге 3.5, параметр RowMapper для findAll() и findById() задается как ссылка на метод mapRowToIngredient(). Ссылки на методы Java 8 и лямбда-выражения удобны при работе с JdbcTemplate в качестве альтернативы явной реализации RowMapper.  Но если по какой-то причине вы хотите или нуждаетесь в явном RowMapper, то следующая реализация findAll() показывает, как это сделать:

@Override  
public Ingredient findOne(String id) {  
 return jdbc.queryForObject(  
   "select id, name, type from Ingredient where id=?",  
   new RowMapper<Ingredient>() {  
     public Ingredient mapRow(ResultSet rs, int rowNum) throws SQLException {  
       return new Ingredient(  
       rs.getString("id"),  
       rs.getString("name"),  
       Ingredient.Type.valueOf(rs.getString("type")));  
     };  
   }, id);  
}

Чтение данных из базы данных-это только часть истории. В какой-то момент данные должны быть записаны в базу данных, чтобы их можно было прочитать.  Итак, давайте рассмотрим реализацию метода save().

ВСТАВКА СТРОКИ

Метод update() JdbcTemplate может использоваться для любого запроса, который записывает или обновляет данные в базе данных. И, как показано в следующем листинге, его можно использовать для вставки данные в базу данных.

Листинг 3.6 вставка данных с использованием JdbcTemplate

@Override  
public Ingredient save(Ingredient ingredient) {  
 jdbc.update(  
   "insert into Ingredient (id, name, type) values (?, ?, ?)",  
   ingredient.getId(),  
   ingredient.getName(),  
   ingredient.getType().toString());  
 return ingredient;  
}

Поскольку нет необходимости сопоставлять данные ResultSet с объектом, метод update() намного проще, чем query() или queryForObject (). Для него требуется только String, содержащий SQL, а также значения, для параметров запроса. В данном случае запрос имеет три параметра, которые соответствуют последним трем параметрам запроса метода save() - ID ингредиента, name и type.

С JdbcIngredientRepository закончили, теперь вы можете внедрить его в DesignTacoController и использовать его для предоставления списка объектов Ingredient  вместо использования жестко закодированных значений (как вы сделали в главе 2). Изменения в DesignTacoController показаны далее.

Листинг 3.7 Внедрение и использование репозитория в контроллере

@Controller  
@RequestMapping("/design")  
@SessionAttributes("order")  
public class DesignTacoController {  
 private final IngredientRepository ingredientRepo;  
  
 @Autowired  
 public DesignTacoController(IngredientRepository ingredientRepo) {  
   this.ingredientRepo = ingredientRepo;  
 }  
  
 @GetMapping  
 public String showDesignForm(Model model) {  
   List<Ingredient> ingredients = new ArrayList<>();  
   ingredientRepo.findAll().forEach(i -> ingredients.add(i));  
   Type[] types = Ingredient.Type.values();  
   for (Type type : types) {  
     model.addAttribute(type.toString().toLowerCase(),  
     filterByType(ingredients, type));  
   }

   return "design";

 }  
 ...  
}

Обратите внимание, что вторая строка метода showDesignForm() теперь вызывает метод findAll() внедренного IngredientRepository. Метод findAll() извлекает все ингредиенты из базы данных перед их фильтрацией в различные типы в модели.

Вы почти готовы запустить приложение и попробовать эти изменения. Но прежде чем начать чтение данных из таблицы Ingredient, которая используется в запросе, вероятно, следует создать эту таблицу и заполнить ее хоть какими-то данными ингредиентов.

3.1.3 Создание схемы и предварительная загрузка данных

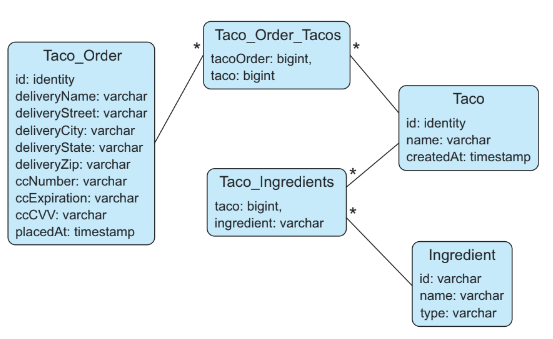
Помимо таблицы Ingredient, вам также понадобятся некоторые таблицы, содержащие информацию о заказах и составе тако. На рисунке 3.1 показаны необходимые таблицы, а также связи между ними.  


Рисунок 3.1 таблицы схемы Taco Cloud

Таблицы на рис. 3.1 служат следующим целям:

* Ingredient - содержит информацию об ингредиенте
* Taco - содержит важную информацию о составе taco
* Taco\_Ingredients - содержит одну или несколько строк для каждой строки в Taco, сопоставляя taco с ингредиентами для этого taco
* Taco\_Order-содержит важные данные order
* Taco\_Order\_Tacos - содержит одну или несколько строк для каждой строки в Taco\_Order, сопоставляя order с taco-ми в заказе

Следующий листинг показывает SQL, который создает таблицы.

Листинг 3.8 Схема Taco Cloud

create table if not exists Ingredient (  
 id varchar(4) not null,  
 name varchar(25) not null,  
 type varchar(10) not null  
);  
  
create table if not exists Taco (  
 id identity,  
 name varchar(50) not null,  
 createdAt timestamp not null  
);  
  
create table if not exists Taco\_Ingredients (  
 taco bigint not null,  
 ingredient varchar(4) not null  
);  
alter table Taco\_Ingredients add foreign key (taco) references Taco(id);  
alter table Taco\_Ingredients add foreign key (ingredient) references Ingredient(id);  
create table if not exists Taco\_Order (  
 id identity,  
 deliveryName varchar(50) not null,  
 deliveryStreet varchar(50) not null,  
 deliveryCity varchar(50) not null,  
 deliveryState varchar(2) not null,  
 deliveryZip varchar(10) not null,  
 ccNumber varchar(16) not null,  
 ccExpiration varchar(5) not null,  
 ccCVV varchar(3) not null,  
 placedAt timestamp not null  
);  
  
create table if not exists Taco\_Order\_Tacos (  
 tacoOrder bigint not null,  
 taco bigint not null  
);  
  
alter table Taco\_Order\_Tacos add foreign key (tacoOrder) references Taco\_Order(id);  
alter table Taco\_Order\_Tacos add foreign key (taco) references Taco(id);

Большой вопрос заключается в том, куда поместить это определение схемы. Как оказалось, Spring Boot содержит ответ на этот вопрос.

Если есть файл с именем schema.sql в корне classpath приложения, а затем SQL в этом файле будет выполняться в базе данных при запуске приложения. Поэтому содержимое листинга 3.8 следует поместить в проект в виде файла с именем schema.sql в папке src/main/resources.

Вам также необходимо предварительно загрузить базу данных с данными по ингредиентам. К счастью, Spring Boot также выполнит файл с именем data.sql из корня classpath при запуске приложения. Таким образом, вы можете заполнить базу данных данными ингредиентов, используя инструкции insert в следующем листинге, размещенном в src/main/resources/data.sql.

Листинг 3.9 Предзаполнение БД

delete from Taco\_Order\_Tacos;  
delete from Taco\_Ingredients;  
delete from Taco;  
delete from Taco\_Order;  
delete from Ingredient;  
  
insert into Ingredient (id, name, type) values ('FLTO', 'Flour Tortilla', 'WRAP');  
insert into Ingredient (id, name, type) values ('COTO', 'Corn Tortilla', 'WRAP');  
insert into Ingredient (id, name, type) values ('GRBF', 'Ground Beef', 'PROTEIN');  
insert into Ingredient (id, name, type) values ('CARN', 'Carnitas', 'PROTEIN');  
insert into Ingredient (id, name, type) values ('TMTO', 'Diced Tomatoes', 'VEGGIES');  
insert into Ingredient (id, name, type) values ('LETC', 'Lettuce', 'VEGGIES');  
insert into Ingredient (id, name, type) values ('CHED', 'Cheddar', 'CHEESE');  
insert into Ingredient (id, name, type) values ('JACK', 'Monterrey Jack', 'CHEESE');  
insert into Ingredient (id, name, type) values ('SLSA', 'Salsa', 'SAUCE');  
insert into Ingredient (id, name, type) values ('SRCR', 'Sour Cream', 'SAUCE');

Несмотря на то, что вы только разработали хранилище для данных ингредиентов, вы можете запустить приложение Taco Cloud на этом этапе и посетить страницу проектирования, чтобы увидеть JdbcIngredientRepository в действии. Ну же… попробуйте. Когда вы вернетесь к чтению, вы напишете репозитории для сохранения Taco, Order и data.

3.1.4 Вставка данных

Вы уже имели представление о том, как использовать JdbcTemplate для записи данных в базу данных. Метод save() в JdbcIngredientRepository использовал метод update() JdbcTemplate для сохранения объектов ингредиентов в базе данных.

Хотя это был хороший первый пример, возможно, это было слишком просто. Как вы скоро увидите, сохранение данных может быть более сложным, чем то, что необходимо JdbcIngredientRepository. Два способа сохранения данных с помощью JdbcTemplate включают следующее:

* Напрямую, используя метод update()
* С помощью класса-оболочки SimpleJdbcInsert

Давайте сначала посмотрим, как использовать метод update(), когда задача сложнее, чем та, что была при сохранение Ingredient.

СОХРАНЕНИЕ ДАННЫХ С ПОМОЩЬЮ JDBCTEMPLATE

На данный момент единственное, что нужно сделать в репозиториях taco и order, это сохранение их соответствующие объекты. Чтобы сохранить объекты Taco, TacoRepository объявляет метод save():

package tacos.data;  
  
import tacos.Taco;  
public interface TacoRepository {  
 Taco save(Taco design);  
}

Аналогичным образом OrderRepository также объявляет метод save:

package tacos.data;  
  
import tacos.Order;  
  
public interface OrderRepository {  
 Order save(Order order);  
}

Кажется достаточно простым, верно? Не так быстро. Сохранение состава taco требует, чтобы вы также сохранили ингредиенты, связанные с этим taco в таблицу Taco\_Ingredients. Точно так же сохранение заказа требует, чтобы вы также сохранили tacos, привязанные к заказу в таблице Taco\_Order\_Tacos. Это делает сохранение тако и заказов немного более сложным, чем то, что требовалось для сохранения ингредиента.

Для реализации TacoRepository необходим метод save(), который начинается с сохранения основных деталей состава taco (например, имени и времени создания), а затем вставляет одну строку в Taco\_Ingredients для каждого ингредиента в объекте Taco. Ниже приведен полный класс JdbcTacoRepository.

Листинг 3.10 реализация TacoRepository с JdbcTemplate

package tacos.data;  
  
import java.sql.Timestamp;  
import java.sql.Types;  
import java.util.Arrays;  
import java.util.Date;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.jdbc.core.PreparedStatementCreator;  
import org.springframework.jdbc.core.PreparedStatementCreatorFactory;  
import org.springframework.jdbc.support.GeneratedKeyHolder;  
import org.springframework.jdbc.support.KeyHolder;  
import org.springframework.stereotype.Repository;  
  
import tacos.Ingredient;  
import tacos.Taco;  
  
@Repository  
public class JdbcTacoRepository implements TacoRepository {  
  
 private JdbcTemplate jdbc;  
  
 public JdbcTacoRepository(JdbcTemplate jdbc) {  
   this.jdbc = jdbc;  
 }  
  
 @Override  
 public Taco save(Taco taco) {  
   long tacoId = saveTacoInfo(taco);  
   taco.setId(tacoId);  
   for (Ingredient ingredient : taco.getIngredients()) {  
     saveIngredientToTaco(ingredient, tacoId);  
   }  
   return taco;  
 }  
  
 private long saveTacoInfo(Taco taco) {  
   taco.setCreatedAt(new Date());  
   PreparedStatementCreator psc = new PreparedStatementCreatorFactory(  
     "insert into Taco (name, createdAt) values (?, ?)",  
     Types.VARCHAR, Types.TIMESTAMP  
   ).newPreparedStatementCreator(  
   Arrays.asList(  
     taco.getName(),  
     new Timestamp(taco.getCreatedAt().getTime()))  
   );  
   KeyHolder keyHolder = new GeneratedKeyHolder();  
   jdbc.update(psc, keyHolder);  
   return keyHolder.getKey().longValue();  
 }  
  
 private void saveIngredientToTaco(Ingredient ingredient, long tacoId) {  
   jdbc.update(  
     "insert into Taco\_Ingredients (taco, ingredient) " +  
     "values (?, ?)",  
     tacoId, ingredient.getId()  
   );  
 }  
}

Как вы можете видеть, метод save() начинается с вызова private метода saveTacoInfo(), а затем использует ID taco, возвращенный из этого метода, для вызова saveIngredientToTaco(), который сохраняет каждый ингредиент. Дьявол кроется в деталях saveTacoInfo().

Когда вы вставляете строку в Taco, вам нужно знать идентификатор, сгенерированный базой данных, чтобы вы могли ссылаться на него в каждом из ингредиентов.  Метод update(), используемый при сохранении данных ингредиента, не помогает вам получить сгенерированный идентификатор, поэтому вам нужен другой метод update().

Метод update () принимает PreparedStatementCreator и KeyHolder. Это KeyHolder, который предоставит сгенерированный идентификатор taco. Но чтобы использовать его, необходимо также создать PreparedStatementCreator.

Как видно из листинга 3.10, создание PreparedStatementCreator нетривиально. Начните с создания PreparedStatementCreatorFactory, предоставив ему SQL, который вы хотите выполнить, а также типы каждого параметра запроса. Затем вызовите newPreparedStatementCreator() на этой фабрике, передав значения, необходимые в параметрах запроса для создания PreparedStatementCreator.

С PreparedStatementCreator на руках, вы можете вызвать update(), передав в PreparedStatementCreator и KeyHolder (в этом случае экземпляр GeneratedKeyHolder).  Как только update() закончено, можно возвратить ID taco, возвращая keyHolder.getKey().longValue().

Возвращаясь к save(), в цикле для каждого каждый Ingredient в Taco, вызывается saveIngredientToTaco(). В saveIngredientToTaco() методе используется простая форма update() для сохранения списка ингредиентв таблицу Taco\_Ingredients.

Все, что осталось сделать с TacoRepository, - это внедрить его в DesignTacoController и использовать его при сохранении Taco. В следующем листинге показаны изменения, необходимые для внедрения репозитория.

Листинг 3.11 Внедрение и использование TacoRepository

@Controller  
@RequestMapping("/design")  
@SessionAttributes("order")  
public class DesignTacoController {  
 private final IngredientRepository ingredientRepo;  
 private TacoRepository designRepo;  
  
 @Autowired  
 public DesignTacoController( IngredientRepository ingredientRepo, TacoRepository designRepo) {  
   this.ingredientRepo = ingredientRepo;  
   this.designRepo = designRepo;  
 }  
 …  
}

Как вы можете видеть, конструктор принимает как IngredientRepository, так и TacoRepository. Он назначает переменные экземпляра, чтобы их можно было использовать в методах showDesignForm() и processDesign().

Говоря о методе processDesign(), его изменения немного более обширны, чем изменения, которые вы сделали в showDesignForm(). В следующем списке показан новый метод processDesign().

Листинг 3.12 сохранение состава taco и их привязка к заказам

@Controller  
@RequestMapping("/design")  
@SessionAttributes("order")  
public class DesignTacoController {  
  
 @ModelAttribute(name = "order")  
 public Order order() {  
   return new Order();  
 }  
  
 @ModelAttribute(name = "taco")  
 public Taco taco() {  
   return new Taco();  
 }  
  
 @PostMapping  
 public String processDesign( @Valid Taco design, Errors errors, @ModelAttribute Order order) {  
   if (errors.hasErrors()) {  
     return "design";  
   }  
   Taco saved = designRepo.save(design);  
   order.addDesign(saved);  
   return "redirect:/orders/current";  
 }  
 ...  
}

Первое, что можно заметить в коде в листинге 3.12, это то, что DesignTacoController теперь аннотируется SessionAttributes("order") и что у него есть новый аннотированный @ModelAttribute метод order(). Как и в случае с методом taco(), аннотация @ModelAttribute для order() гарантирует, что объект Order будет создан в модели. Но в отличие от объекта Taco в сеансе необходимо, чтобы order присутствовал в нескольких запросах, чтобы можно было создать несколько тако и добавить их в заказ. Аннотация @SessionAttributes на уровне класса задает любые объекты модели, такие как атрибут order, которые должны храниться в сеансе и доступны для нескольких запросов.

Реальная обработка состава taco происходит в методе processDesign(), который теперь принимает объект Order в качестве параметра, в дополнение к объектам Taco и Errors. Параметр Order аннотируется @ModelAttribute, чтобы указать, что его значение должно исходить из модели и что Spring MVC не должен пытаться привязать к нему параметры запроса.

После проверки на наличие ошибок, processDesign() использует внедренный TacoRepository для сохранения taco. Затем он добавляет объект Taco в Order, который сохраняется в сеансе.

Фактически объект Order остается в сеансе и не сохраняется в базе данных до тех пор, пока пользователь не завершит и не отправит форму заказа. В этот момент OrderController  должен вызвать реализацию OrderRepository, чтобы сохранить заказ. Давайте напишем эту реализацию.

ВСТАВКА ДАННЫХ С SempleJdbcInsert

Вы помните, что сохранение тако связано не только с сохранением имени тако и времени создания в таблицу Taco, но и с сохранением ссылки на ингредиенты, связанные с тако, в таблицу Taco\_Ingredients. И вы также помните, что это потребовало от вас знать идентификатор Taco, который вы получили с помощью KeyHolder и PreparedStatementCreator.

Когда дело доходит до сохранения заказов, существует аналогичное обстоятельство. Необходимо не только сохранить данные заказа в таблице Taco\_Order, но также и ссылки на каждый taco в заказе в таблице Taco\_Order\_Tacos. Но вместо того, чтобы использовать громоздкий PreparedStatementCreator, позвольте мне представить вам SimpleJdbcInsert, объект, который обертывает JdbcTemplate, чтобы упростить вставку данных в таблицу.

Начнем с создания JDBCOrderRepository, implementation OrderRepository.  Но прежде чем писать реализацию метода save(), давайте сосредоточимся на конструкторе, где вы создадите пару экземпляров SimpleJdbcInsert для вставки значений в таблицы Taco\_Order и Taco\_Order\_Tacos.  В следующем листинге показан JDBCOrderRepository (без метода save ()).

Листинг 3.13 создание SimpleJdbcInsert из JdbcTemplate

package tacos.data;  
  
import java.util.Date;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.jdbc.core.simple.SimpleJdbcInsert;  
import org.springframework.stereotype.Repository;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import tacos.Taco;  
import tacos.Order;  
  
@Repository  
public class JdbcOrderRepository implements OrderRepository {  
 private SimpleJdbcInsert orderInserter;  
 private SimpleJdbcInsert orderTacoInserter;  
 private ObjectMapper objectMapper;  
  
 @Autowired  
 public JdbcOrderRepository(JdbcTemplate jdbc) {  
   this.orderInserter = new SimpleJdbcInsert(jdbc)  
     .withTableName("Taco\_Order")  
     .usingGeneratedKeyColumns("id");  
   this.orderTacoInserter = new SimpleJdbcInsert(jdbc)  
     .withTableName("Taco\_Order\_Tacos");  
   this.objectMapper = new ObjectMapper();  
 }  
 ...  
}

Как и JdbcTacoRepository, JdbcOrderRepository inject-ид JdbcTemplate через его конструктор. Но вместо назначения JdbcTemplate непосредственно переменной экземпляра конструктор использует его для создания нескольких экземпляров SimpleJdbcInsert.

Первый экземпляр, который назначен переменной экземпляра orderInserter, настроен для работы с таблицей Taco\_Order и предполагает, что свойство id будет предоставлено или сгенерировано базой данных. Второй экземпляр, назначенный orderTacoInserter, настроен для работы с таблицей Taco\_Order\_Tacos, но не содержит никаких инструкций о том, как ID будут генерироваться в этой таблице.

Конструктор также создает экземпляр Jackson ObjectMapper и присваивает его переменной экземпляра. Хотя Jackson предназначен для обработки JSON, вы увидите, как мы перепрофилируем его, чтобы помочь вам сохранять заказы и связанные с ними тако.

Теперь давайте посмотрим, как метод save() использует экземпляры SimpleJdbcInsert. В следующем списке показан метод save(), а также несколько частных методов save(), которые делегируются для реальной работы.

Листинг 3.14 использование SimpleJdbcInsert для вставки данных

@Override  
public Order save(Order order) {  
 order.setPlacedAt(new Date());  
 long orderId = saveOrderDetails(order);  
 order.setId(orderId);  
 List<Taco> tacos = order.getTacos();   
  for (Taco taco : tacos) {  
   saveTacoToOrder(taco, orderId);  
 }  
 return order;  
}  
  
private long saveOrderDetails(Order order) {  
 @SuppressWarnings("unchecked")  
 Map<String, Object> values = objectMapper.convertValue(order, Map.class);  
 values.put("placedAt", order.getPlacedAt());  
 long orderId =orderInserter  
   .executeAndReturnKey(values)  
   .longValue();  
 return orderId;  
}  
  
private void saveTacoToOrder(Taco taco, long orderId) {  
 Map<String, Object> values = new HashMap<>();  
 values.put("tacoOrder", orderId);  
 values.put("taco", taco.getId());  
 orderTacoInserter.execute(values);  
}

Метод save() ничего не сохраняет. Он определяет поток для сохранения Order и связанных с ним объектов Taco и делегирует работу saveOrderDetails() и saveTacoToOrder().

SimpleJdbcInsert имеет несколько полезных методов для выполнения вставкиt: execute() и executeAndReturnKey(). Оба принимают Map<String, Object>, где map-ключи соответствуют именам столбцов в таблице, в которую вставляются данные. Map-значения вставляются в эти столбцы.

Такую Map легко создать, скопировав значения из Order в записи Map. Но у Order есть несколько свойств, и все они имеют одинаковое имя со столбцами, в которые они входят.  Из-за этого в saveOrderDetails() я решил использовать Jackson ObjectMapper и его метод convertValue() для преобразования Order в Map(Я признаю, что это хакерское использование ObjectMapper, но у вас уже есть Jackson в classpath; Spring Boot’s web starter включает его. Кроме того, использование ObjectMapper для сопоставления объекта с Map намного проще, чем копирование каждого свойства из объекта в Map. Не стесняйтесь заменить использование ObjectMapper любым кодом, который вы предпочитаете, который строит Map, который вы передадите на вставку объектов.). После создания Map задайте для записи placedAt значение свойства placedAt объекта Order.  Это необходимо, поскольку в противном случае ObjectMapper преобразует свойство Date в long, что несовместимо с полем placedAt в таблице Taco\_Order.

С заполненным Map данными заказов, вы можете вызвать executeAndReturnKey() в orderInserter. Это сохраняет информацию о заказе в таблице Taco\_Order и возвращает сгенерированный базой данных ID как объект типа Number, который преобразуется long с помощью вызова longValue(), и возвращается из метода.

Метод saveTacoToOrder() значительно проще. Вместо того чтобы использовать ObjectMapper для преобразования объекта в Map, создается  Map и задаются соответствующие значения. Еще раз, ключи Map соответствуют именам столбцов в таблице. Вызов метода orderTacoInserter.execute() выполняет insert.

Теперь вы можете inject OrderRepository в OrderController и начать использовать его. Следующий листинг показывает OrderController, включая изменения для использования inject OrderRepository.

Листинг 3.15 использование OrderRepository в OrderController

package tacos.web;  
  
import javax.validation.Valid;  
import org.springframework.stereotype.Controller;  
import org.springframework.validation.Errors;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.SessionAttributes;  
import org.springframework.web.bind.support.SessionStatus;  
import tacos.Order;  
import tacos.data.OrderRepository;  
  
@Controller  
@RequestMapping("/orders")  
@SessionAttributes("order")  
public class OrderController {  
 private OrderRepository orderRepo;  
  
 public OrderController(OrderRepository orderRepo) {  
   this.orderRepo = orderRepo;  
 }  
  
 @GetMapping("/current")  
 public String orderForm() {  
   return "orderForm";  
 }  
  
 @PostMapping  
 public String processOrder(@Valid Order order, Errors errors,SessionStatus sessionStatus) {  
   if (errors.hasErrors()) {  
     return "orderForm";  
   }  
   orderRepo.save(order);  
   sessionStatus.setComplete();  
   return "redirect:/";  
 }  
}

Помимо внедрения OrderRepository в контроллер, единственные существенные изменения в OrderController происходят в методе processOrder(). Здесь объект Order, представленный в форме (который также является тем же самым объектом Order, поддерживаемым в сеансе), сохраняется с помощью метода save() внедренного OrderRepository.

После того, как заказ будет сохранен, вам больше не нужно, чтобы он висел в сеансе. Фактически, если вы не очистите его, заказ остается в сеансе, включая связанные с ним тако, и следующий заказ начнется с тако, содержащимися в старом заказе. Поэтому метод processOrder() запрашивает параметр состояния сеанса и вызывает метод setComplete() для сброса сеанса.

Весь код сохранения JDBC закончен. Теперь вы можете запустить приложение Taco Cloud и проверить его работоспособность.  Не стесняйтесь создавать столько тако и столько заказов, сколько вы хотите.

Вы также можете найти полезным покопаться в базе данных.  Поскольку вы используете H2 в качестве встроенной базы данных, и поскольку у вас есть Spring Boot DevTools, вы можете обратиться в вашем браузере по адресу  http://localhost:8080/h2-console чтобы увидеть консоль H2. По умолчанию поле JDBC URL должно быть установлено как jdbc:h2:mem:testdb. После входа в систему, вы должны быть в состоянии выполнить любой запрос к таблицам схемы Taco Cloud.

Spring JdbcTemplate, наряду с SimpleJdbcInsert, делает работу с реляционными базами данных значительно проще, чем простой vanilla JDBC.  Но вы можете обнаружить, что JPA делает это еще проще. Давайте пересмотрим вашу работу и посмотрим, как использовать Spring Data, чтобы сделать сохранение данных еще проще.

3.2 Сохранение данных с помощью Spring Data JPA

Проект Spring Data представляет собой довольно крупный зонтичный (umbrella) проект, состоящий из нескольких подпроектов, большинство из которых сосредоточены на сохранении данных с различными типами баз данных. Некоторые из самых популярных Spring Data проектов включают в себя следующие:

* Spring Data JPA — Сохранение JPA в реляционной базе данных
* Spring Data MongoDB — Сохранение в базе данных документов Mongo
* Spring Data Neo4j — Сохранение в базе данных графов Neo4j
* Spring Data Redis — Сохранение в хранилище ключей и значений Redis
* Spring Data Cassandra — Сохранение в базе данных Cassandra

Одной из наиболее интересных и полезных функций Spring Data для всех этих проектов является возможность автоматического создания репозиториев на основе интерфейса спецификации репозитория.

Чтобы увидеть, как работают Spring Data, потребуется начать все сначала, заменив репозитории на основе JDBC из ранее в этой главе репозиториями, созданными Spring Data JPA. Но сначала нужно добавить Spring Data JPA в сборку проекта.

3.2.1 Добавление Spring Data JPA в проект

Spring  Data JPA доступны для Spring  Boot приложений с JPA starter-ом. Эта зависимость стартера не только приносит в Spring Data JPA, но и транзитивно включает Hibernate как реализацию JPA:

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

Если вы хотите использовать другую реализацию JPA, то вам нужно, по крайней мере, исключить зависимость Hibernate и включить библиотеку JPA по вашему выбору. Например, чтобы использовать EclipseLink вместо Hibernate, необходимо изменить сборку следующим образом:

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 <exclusions>  
   <exclusion>  
     <artifactId>hibernate-entitymanager</artifactId>  
     <groupId>org.hibernate</groupId>  
   </exclusion>  
 </exclusions>  
</dependency>  
<dependency>  
 <groupId>org.eclipse.persistence</groupId>  
 <artifactId>eclipselink</artifactId>  
 <version>2.5.2</version>  
</dependency>

Обратите внимание, что могут потребоваться другие изменения в зависимости от выбранного варианта реализации JPA. Обратитесь к документации к выбранной реализации JPA для деталей. Теперь давайте вернемся к вашим доменным объектам и аннотируем их для сохранения JPA.

3.2.2 Аннотирование домена как сущностей

Как вы скоро увидите, Spring Data делает удивительные вещи, когда дело доходит до создания репозиториев. Но, к сожалению, это не очень помогает, когда дело доходит до аннотирования объектов домена аннотациями сопоставления JPA. Вам нужно будет открыть Ingredient, Taco и Order классы и добавить в них несколько аннотаций. Первым изменим класс Ingredient.

Листинг 3.16 аннотирование Ingredient для сохранения JPA

package tacos;  
  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import lombok.AccessLevel;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import lombok.RequiredArgsConstructor;  
  
@Data  
@RequiredArgsConstructor  
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)  
@Entity  
public class Ingredient {  
 @Id  
 private final String id;  
 private final String name;  
 private final Type type;  
  
 public static enum Type {  
   WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE  
 }  
}

Чтобы объявить это как сущность JPA, Ingredient должен быть аннотирован @Entity. И его свойство id должно быть аннотировано @Id, чтобы обозначить его как свойство, которое будет однозначно идентифицировать сущность в базе данных.

В дополнение к аннотациям, специфичным для JPA, вы также заметили, что вы добавили аннотацию @NoArgsConstructor на уровне класса. JPA требует, чтобы сущности имели конструктор без аргументов, поэтому @NoArgsConstructor Lombok-а делает это за вас. Однако, вы не хотите, чтобы имелась возможность использовать его, поэтому сделайте его private, установив атрибут доступа AccessLevel.PRIVATE. И поскольку есть final свойства, которые должны быть установлены, вы также устанавливаете атрибут force в true, что приводит к тому, что конструктор, сгенерированный Lombok-ом, устанавливает их в null.

Также добавляется @RequiredArgsConstructor. @Data неявно добавляет конструктор обязательных аргументов, но при использовании @NoArgsConstructor этот конструктор удаляется. Явный @RequiredArgsConstructor гарантирует, что у вас все еще будет конструктор обязательных аргументов в дополнение к закрытому конструктору без аргументов.

Теперь давайте перейдем к классу Taco и посмотрим, как аннотировать его как сущность JPA.

Листинг 3.17 аннотирование Taco как сущности

package tacos;  
  
import java.util.Date;  
import java.util.List;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
import javax.persistence.ManyToMany;  
import javax.persistence.OneToMany;  
import javax.persistence.PrePersist;  
import javax.validation.constraints.NotNull;  
import javax.validation.constraints.Size;   
import lombok.Data;   
  
@Data  
@Entity  
public class Taco {  
 @Id  
 @GeneratedValue(strategy=GenerationType.AUTO)  
 private Long id;  
  
 @NotNull  
 @Size(min=5, message="Name must be at least 5 characters long")  
 private String name;  
 private Date createdAt;  
  
 @ManyToMany(targetEntity=Ingredient.class)  
 @Size(min=1, message="You must choose at least 1 ingredient")  
 private List<Ingredient> ingredients;  
 @PrePersist  
 void createdAt() {  
   this.createdAt = new Date();  
 }  
}

Как и в случае с Ingredient, класс Taco теперь аннотируется @Entity и имеет свойство id, аннотированное @Id. Поскольку вы полагаетесь на базу данных для автоматического создания значения идентификатора, вы также аннотируете свойство id с помощью @GeneratedValue, указывая стратегию AUTO.

Чтобы объявить связь между Taco и связанным с ним списком Ingredient-ов, аннотируйте  ingredients с помощью @ManyToMany. Taco может иметь много объектов -Ingredient, и Ingredient может быть частью многих Taco.

Вы также заметили, что есть новый метод createdAt(), который аннотируется @PrePersist. Вы будете использовать его, чтобы установить свойство createdAt в текущую дату и время, прежде чем сохранить Taco. Наконец, давайте аннотируем объект Order как сущность. В следующем листинге показан новый класс Order.

Листинг 3.18 Аннотирование Order как сущность JPA

package tacos;  
  
import java.io.Serializable;  
import java.util.ArrayList;  
import java.util.Date;  
import java.util.List;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
import javax.persistence.ManyToMany;  
import javax.persistence.OneToMany;  
import javax.persistence.PrePersist;  
import javax.persistence.Table;  
import javax.validation.constraints.Digits;  
import javax.validation.constraints.Pattern;  
import org.hibernate.validator.constraints.CreditCardNumber;  
import org.hibernate.validator.constraints.NotBlank;  
import lombok.Data;  
  
@Data  
@Entity  
@Table(name="Taco\_Order")  
public class Order implements Serializable {  
 private static final long serialVersionUID = 1L;  
  
 @Id  
 @GeneratedValue(strategy=GenerationType.AUTO)  
 private Long id;  
 private Date placedAt;  
 ...  
  
 @ManyToMany(targetEntity=Taco.class)  
 private List<Taco> tacos = new ArrayList<>();  
  
 public void addDesign(Taco design) {  
   this.tacos.add(design);  
 }  
  
 @PrePersist  
 void placedAt() {  
   this.placedAt = new Date();  
 }  
}

Как вы можете видеть, изменения Order похожи на изменения в Taco. Но есть одна новая аннотация на уровне класса: @Table. Это указывает, что сущности Order должны сохраняться в таблице с именем Taco\_Order в базе данных.

Хотя вы могли бы использовать эту аннотацию на любом из объектов, это необходимо с Order. Без него JPA по умолчанию сохранит сущности в таблице с именем Order, но order является зарезервированным словом в SQL и вызовет проблемы. Теперь, когда сущности должным образом аннотированы, пришло время написать ваши репозитории.

3.2.3 Объявление репозиториев JPA

В версиях репозиториев JDBC вы явным образом объявили методы, которые должен предоставить репозиторий. Но с Spring Data вместо этого можно расширить интерфейс CrudRepository. Например, вот новый интерфейс IngredientRepository:

package tacos.data;  
  
import org.springframework.data.repository.CrudRepository;  
import tacos.Ingredient;  
  
public interface IngredientRepository  
 extends CrudRepository<Ingredient, String> {  
}

CrudRepository объявляет около десятка методов для операций CRUD (create, read, update, delete). Обратите внимание, что он параметризован, при этом первым параметром является тип сущности, который должен сохраняться в репозитории, а вторым параметром тип свойства entity ID. Для IngredientRepository параметрами должны быть Ingredient и String.

Аналогичным образом можно определить TacoRepository следующим образом:

package tacos.data;  
  
import org.springframework.data.repository.CrudRepository;  
import tacos.Taco;  
  
public interface TacoRepository  
 extends CrudRepository<Taco, Long> {  
}

Единственные существенные различия между IngredientRepositoryв и TacoRepository - это параметры CrudRepository. Здесь они установлены как Taco и Long, чтобы указать сущность Taco (и ее тип идентификатора) в качестве единицы сохранения для этого интерфейса репозитория. Наконец, те же изменения могут быть применены к OrderRepository:

package tacos.data;  
  
import org.springframework.data.repository.CrudRepository;  
import tacos.Order;  
  
public interface OrderRepository  
 extends CrudRepository<Order, Long> {  
}

И теперь у вас есть три репозитория.  Вы можете подумать, что вам нужно написать реализации для всех трех, включая дюжину методов для каждой реализации. Но в Spring Data JPA - нет необходимости писать реализацию! При запуске приложения Spring Data JPA автоматически создает реализацию на лету. Это означает, что репозитории готовы к использованию с самого начала. Просто вставьте их в контроллеры, как вы сделали для реализаций на основе JDBC, и все готово.

Методы, предоставляемые CrudRepository обширны и отлично подходят для большинства наиболее распространенных задач. Но что, если у вас есть некоторые требования, выходящие за рамки базовой настойчивости? Давайте посмотрим как настроить репозитории для выполнения запросов, уникальных для вашего домена.

3.2.4 Настройка репозиториев JPA

Представьте, что в дополнение к основным операциям CRUD, предоставляемым CrudRepository, вам также необходимо получить все заказы, доставленные по заданному почтовому индексу. Как оказалось, это можно легко решить, добавив следующее объявление метода в OrderRepository:

List<Order> findByDeliveryZip(String deliveryZip);

При создании реализации репозитория Spring Data проверяет все методы в интерфейсе репозитория, анализирует имя метода и пытается понять назначение метода в контексте сохраняемого объекта (в данном случае-порядок).  По сути, Spring Data определяет своего рода миниатюрный доменный язык (DSL), в котором сведения о сохраняемости выражаются в сигнатурах методов репозитория.

Spring Data знает, что этот метод предназначен для поиска заказов, потому что вы параметризовали CrudRepository с Order.  Имя метода, findByDeliveryZip(), дает понять, что этот метод должен найти все сущности Order, сопоставив их свойство deliveryZip со значением, переданным в качестве параметра в метод.

Метод findByDeliveryZip() достаточно прост, но Spring Data может обрабатывать еще более интересные имена методов. Методы репозитория состоят из глагола, необязательного субъекта, слова By и предиката. В случае findByDeliveryZip() глагол find и предикат DeliveryZip; субъект не указан и подразумевается как Order.

Рассмотрим другой, более сложный пример. Предположим, что вам нужно запросить все заказы, доставленные по заданному почтовому индексу в заданном диапазоне дат. В этом случае может оказаться полезным следующий метод в OrderRepository:

List<Order> readOrdersByDeliveryZipAndPlacedAtBetween(String deliveryZip, Date startDate, Date endDate);

На рис. 3.2 показано, как Spring Data анализирует и понимает readOrdersByDeliveryZipAndPlacedAtBetween() при генерации реализации репозитория. Как вы можете видеть, глагол в readOrdersByDeliveryZipAndPlacedAtBetween() read. Spring Data также понимает find, read и get как синонимы для извлечения одной или нескольких сущностей. Кроме того, можно также использовать count в качестве глагола, если требуется, чтобы метод возвращал значение int с числом совпадающих сущностей.

-Этот метод будет считывать (read) данные (”get “и” find " также разрешены здесь).

-Обозначает начало свойств для соответствия

-...and…

-Значение должно находиться между заданными значениями.

-Соотвествие .deliveryZip или -.delivery.zip свойство

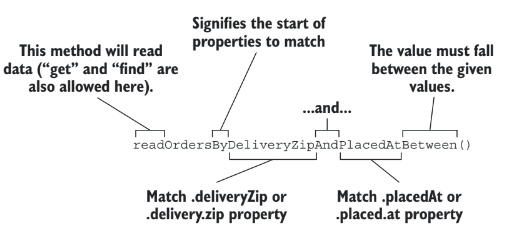
-Соотвествие .placedAt или .placed.at свойство  


Рис. 3.2. Spring Data анализирует сигнатуры методов репозитория для определения запроса, который необходимо выполнить.

Хотя объект метода является необязательным, здесь он Orders.  Spring Data игнорирует большинство слов в объекте, поэтому вы можете назвать метод readPuppiesBy... и он все равно найдет сущности Order, так как это тип, с которым параметризуется CrudRepository.

Сказуемое следует за словом в имени метода и является наиболее интересной частью сигнатуры метода. В этом случае предикат ссылается на два свойства заказа: deliveryZip и placedAt. Свойство deliveryZip должно быть равно значению, переданному в первый параметр метода. Ключевое слово Between указывает, что значение deliveryZip должно находиться между значениями, переданными в последние два параметра метода.

В дополнение к неявной операции Equals и Between, подписи метода Spring Data могут также включать любой из этих операторов:

-IsAfter,After,IsGreaterThan,GreaterThan

-IsGreaterThanEqual,GreaterThanEqual

-IsBefore,Before,IsLessThan,LessThan

-IsLessThanEqual,LessThanEqual

-IsBetween,Between

-IsNull,Null

-IsNotNull,NotNull

-IsIn,In

-IsNotIn,NotIn

-IsStartingWith,StartingWith,StartsWith

-IsEndingWith,EndingWith,EndsWith

-IsContaining,Containing,Contains

-IsLike,Like

-IsNotLike,NotLike

-IsTrue,True

-IsFalse,False

-Is,Equals

-IsNot,Not

-IgnoringCase,IgnoresCase

В качестве альтернативы IgnoringCase и IgnoresCase можно использовать AllIgnoringCase либо AllIgnoresCase в методе, чтобы игнорировать регистр для всех сравнений строк. Например, рассмотрим следующий метод:

List<Order> findByDeliveryToAndDeliveryCityAllIgnoresCase(String deliveryTo, String deliveryCity);

Наконец, можно также разместить OrderBy в конце имени метода для сортировки результатов по указанному столбцу. Например, deliveryTo в order:

List<Order> findByDeliveryCityOrderByDeliveryTo(String city);

Хотя правила именования могут быть полезны для относительно простых запросов, это не займет много воображения, чтобы увидеть, что имена методов могут выйти из-под контроля для более сложных запросы. В этом случае не стесняйтесь называть метод как угодно и аннотировать его @Query, чтобы явно указать запрос, который будет выполняться при вызове метода, как показано в этом примере:

@Query("Order o where o.deliveryCity='Seattle'")

List<Order> readOrdersDeliveredInSeattle();

В этом простом примере @Query вы запрашиваете все заказы, доставленные в Seattle. Но вы можете использовать @Query для выполнения практически любого запроса, который вы можете придумать, даже если трудно или невозможно выполнить запрос, следуя соглашению об именовании.

ИТОГ

-Spring JdbcTemplate значительно упрощает работу с JDBC.

-PreparedStatementCreator и KeyHolder можно использовать вместе, если необходимо узнать значение идентификатора, созданного базой данных.

-Для простого выполнения вставок данных, используйте SimpleJdbcInsert.

-Spring Data JPA делает сохранение JPA таким же простым, как написание интерфейса репозитория.

[Spring in Action Covers Spring 5.0 перевод на русский. Глава 4](https://beginner-mercenary.blogspot.com/2019/05/spring-in-action-covers-spring-50-4.html)

4. Securing Spring

*Эта глава охватывает:*

* *Автоконфигурирование Spring Security*
* *Определение пользовательского хранилища пользователя*
* *Настройка страницы входа*
* *Защита от CSRF-атак*
* *Определение ваших пользователей*

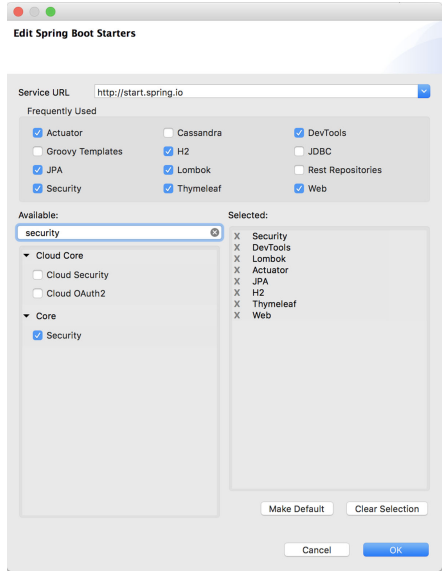
Вы когда-нибудь замечали, что большинство людей в телевизионных ситкомах не закрывают свои двери? Во времена Leave it to Beaver, не было ничего необычного в том, что люди оставляли свои двери незапертыми. Но кажется сумасшедшим, что в тот день, когда мы заботимся о конфиденциальности и безопасности, мы видим телевизионных персонажей, обеспечивающих беспрепятственный доступ к их квартирам и домам.

Информация, вероятно, самый ценный элемент, который мы сейчас имеем; мошенники ищут способы, чтобы украсть наши данные и идентичности, пробираясь в незащищенных приложений. Как разработчики программного обеспечения, мы должны принять меры для защиты информации, которая находится в наших приложениях. Является ли это учетной записью электронной почты, защищенной парой имени пользователя и пароля, или брокерским счетом, защищенным торговым PIN-кодом, безопасность является важным аспектом большинства приложений.

4.1 Включение Spring Security

Самым первым шагом в обеспечении безопасности приложения Spring является добавление зависимости Spring Boot security starter в сборку. В pom.xml файле, добавьте следующую запись <dependency> :

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

Если вы используете Spring Tool Suite, это еще проще. Щелкните правой кнопкой мыши на pom.xml файле и выберите Edit Starters из контекстного меню Spring. Откроется диалоговое окно Starter Dependencies. Выберите запись Security под категорией Core, как показано на рисунке 4.1.  
    
   
Рис. 4.1 добавление стартер безопасности с Spring Tool Suite

Хотите верьте, хотите нет, но зависимость-это единственное, что требуется для защиты приложения. При запуске приложения автоконфигурация обнаружит, что Spring Security находится в пути к классам, и настроит базовую конфигурацию безопасности.

Если вы хотите попробовать, запустите приложение и попробуйте посетить домашнюю страницу (или любую другую страницу). Вам будет предложено выполнить проверку подлинности с помощью обычного HTTP диалогового окна. Чтобы пройти проверку, вам нужно будет предоставить имя пользователя и пароль. Имя пользователя user. Что касается пароля, то он генерируется случайным образом и записывается в файл журнала приложения. Запись журнала будет выглядеть примерно так:

Using default security password: 087cfc6a-027d-44bc-95d7-cbb3a798a1ea

Если вы введете имя пользователя и пароль правильно, вам будет предоставлен доступ к приложению.

Кажется, что обеспечение безопасности приложений Spring довольно простая работа. С защитой приложения Taco Cloud я полагаю, что закончено и я могу закончить эту главу сейчас и перейти к следующей теме. Но прежде чем мы забегаем вперед, давайте рассмотрим, какую автоконфигурацию безопасности мы обеспечили.

Не делая ничего, кроме добавления стартера безопасности в сборку проекта, вы получаете следующие функции безопасности:

-Все пути HTTP-запросов требуют аутентификации.

-Никаких конкретных ролей или полномочий не требуется.

-Нет страницы входа

-Проверка подлинности предлагается с обычной проверкой подлинности http.

-Есть только один пользователь; имя пользователя - *user*.

Это хорошее начало, но я думаю, что потребности в безопасности большинства приложений (включая Taco Cloud) будут сильно отличаться от этих элементарных функций безопасности.

От вас потребуется больше работы, если вы собираетесь правильно защитить приложение Taco Cloud. По крайней мере, необходимо настроить Spring Security для выполнения следующих действий:

-Запрашивать аутентификацию на странице входа, а не в диалоговом окне HTTP basic.

-Создать несколько пользователей и включить страницу регистрации, чтобы новые клиенты Taco Cloud могли зарегистрироваться.

-Применять различные правила безопасности для различных путей запроса. Например, домашняя страница и страницы регистрации вообще не должны требовать аутентификации.

Чтобы удовлетворить ваши потребности в безопасности для Taco Cloud, вам придется написать некоторую явную конфигурацию, переопределяя то, что дала вам автоконфигурация. Вы начнете с настройки надлежащего хранилища пользователей, чтобы иметь более одного пользователя.

4.2 Конфигурирование Spring Security

На протяжении многих лет существует несколько способов настройки Spring Security, включая долгую настройку на основе XML. К счастью, несколько последних версий Spring Security поддерживают конфигурацию на основе Java, которая намного проще для чтения и записи.

Прежде чем эта глава будет закончена, вы настроите все все параметры безопасности Taco Cloud на основаннии Java-based  Spring Security конфигурации. Но чтобы начать работу, вы упростите ее, написав класс заготовки (-barebones) конфигурации, показанный в следующем листинге.

Листинг 4.1 Класс barebones конфигурации для Spring Security

package tacos.security;  
  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
  
@Configuration  
@EnableWebSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
}

Что делает эта barebones конфигурация для вас? Ну, не так много, но это делает вас на шаг ближе к функционалу безопасности который вам нужен. Если вы попытаетесь снова попасть на домашнюю страницу Taco Cloud, вам все равно будет предложено войти. Но вместо запроса диалогового окна обычной проверки подлинности HTTP вам будет показана форма входа в систему, как показано на рис. 4.2.

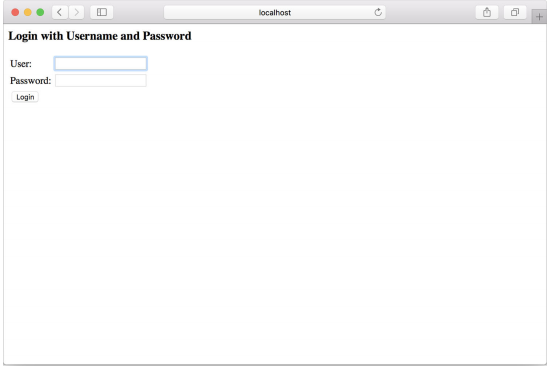


Рисунок 4.2 Spring Security дает вам простую страницу входа бесплатно.

TIP Переход в инкогнито: при ручном тестировании безопасности может оказаться полезным перевести браузер в частный режим или режим инкогнито. Это гарантирует, что у вас будет новая сессия каждый раз, когда вы открываете частное окно/инкогнито. Вам придется каждый раз входить в приложение, но вы можете быть уверены, что все изменения, внесенные в безопасность, будут применены, и что нет никаких остатков старого сеанса, которые не позволят вам увидеть ваши изменения.

Это небольшое улучшение-запрос на вход с веб-страницы (даже если это довольно просто по внешнему виду) всегда более удобно для пользователя, чем диалоговое окно HTTP basic. Вы настроите страницу входа в раздел 4.3.2. Однако текущей задачей является настройка хранилища пользователей, которое может обрабатывать более одного пользователя.

Как оказалось, Spring Security предлагает несколько вариантов настройки пользовательского хранилища, включая следующие:

-Хранилище пользователей в памяти

-Хранилище пользователей на основе JDBC

-Хранилище пользователей в LDAP

-Пользовательская (собственная) служба сведений о пользователе

Независимо от того, какое пользовательское хранилище выбрано, его можно настроить, переопределив метод configure(), определенный в базовом классе конфигурации WebSecurityConfigurerAdapter. Для начала добавьте в класс SecurityConfig следующее переопределение метода:

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 ...  
}

Теперь вам просто нужно заменить эти многоточия с кодом, который использует данный AuthenticationManagerBuilder, чтобы указать, как пользователи будут авторизоваться (looked) во время аутентификации. Сначала вы реализуете хранилище пользователей в памяти.

4.2.1 Хранилище пользователей в памяти

Одно из мест, где пользовательская информация может храниться - это в памяти. Предположим, у вас есть только несколько пользователей, ни один из которых, скорее всего, не изменится.  В этом случае может быть достаточно просто определить этих пользователей как часть конфигурации безопасности.

Листинг 4.2 Определение пользователей в хранилище пользователей в памяти

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {   
 auth.inMemoryAuthentication()  
  .withUser("buzz")  
  .password("infinity")  
  .authorities("ROLE\_USER")  
  .and()  
  .withUser("woody")  
  .password("bullseye")  
  .authorities("ROLE\_USER");  
}

Как вы можете видеть, AuthenticationManagerBuilder использует builder-style API, чтобы настроить параметры проверки подлинности. В этом случае вызов метода inMemoryAuthentication() дает возможность указать сведения о пользователе непосредственно в самой конфигурации безопасности.

Каждый вызов withUser() запускает конфигурацию для пользователя. Значение, указанное в withUser() является имя пользователя, а пароль и полномочий указываются в методах password() и authorities(). Как показано в листинге 4.2, оба пользователя имеют права доступа ROLE\_USER. Пользователь buzz настроен на infinity в качестве пароля. Аналогично, пароль woody -bullseye.

Хранилище пользователей в памяти удобно для тестирования или очень простых приложений, но оно не позволяет легко редактировать пользователей. Если необходимо добавить, удалить или изменить пользователя, необходимо внести необходимые изменения, а затем пересобрать и повторно развернуть приложение.

Для приложения Taco Cloud вы хотите, чтобы клиенты могли регистрироваться в приложении и управлять своими учетными записями пользователей. Это не соответствует ограничениям хранилища пользователей в памяти, поэтому давайте рассмотрим другой вариант, который позволяет использовать хранилище пользователей, поддерживаемое базой данных.

4.2.2 Хранилище пользователей на основе JDBC

Сведения о пользователях часто хранятся в реляционной базе данных, и хранилище пользователей на основе JDBC кажется подходящим. В следующем списке показано, как настроить Spring Security для проверки подлинности сведений о пользователях, хранящихся в реляционной базе данных с помощью JDBC.

Листинг 4.3 Идентификация через JDBC-базу

@Autowired  
DataSource dataSource;  
  
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.jdbcAuthentication()  
   .dataSource(dataSource);  
}

Эта реализация configure() вызывает jdbcAuthentication() на данном AuthenticationManagerBuilder. Для этого необходимо установить DataSource, чтобы он знал, как получить доступ к DataSource. DataSource, используемый здесь, обеспечивается магией autowiring.

**ПЕРЕОПРЕДЕЛЕНИЕ ПОЛЬЗОВАТЕЛЬСКИХ ЗАПРОСОВ ПО УМОЛЧАНИЮ**

Хотя эта минимальная конфигурация будет работать, она делает некоторые предположения о схеме базы данных. Предполагается, что существуют определенные таблицы, в которых будут храниться пользовательские данные. В частности, следующий фрагмент кода из внутренних компонентов Spring Security показывает SQL-запросы, которые будут выполняться при поиске сведений о пользователе:

public static final String DEF\_USERS\_BY\_USERNAME\_QUERY =  
 "select username,password,enabled " +  
 "from users " +  
 "where username = ?";  
  
public static final String DEF\_AUTHORITIES\_BY\_USERNAME\_QUERY =  
 "select username,authority " +  
 "from authorities " +  
 "where username = ?";  
  
public static final String DEF\_GROUP\_AUTHORITIES\_BY\_USERNAME\_QUERY =  
 "select g.id, g.group\_name, ga.authority " +  
 "from groups g, group\_members gm, group\_authorities ga " +  
 "where gm.username = ? " +  
 "and g.id = ga.group\_id " +  
 "and g.id = gm.group\_id";

Первый запрос извлекает имя пользователя, пароль и информацию о том, включены они или нет. Эта информация используется для аутентификации пользователя. Следующий запрос ищет предоставленные полномочия пользователя для целей авторизации, а последний запрос ищет полномочия, предоставленные пользователю как члену группы.

Если вы согласны с определением и заполнением таблиц в базе данных, удовлетворяющих этим запросам, вам больше нечего делать.  Но, скорее всего, ваша база данных не выглядит так, и вам понадобится больше контроля над запросами. В этом случае можно настроить собственные запросы.

Листинг 4.4 Настройка запросов сведений о пользователе

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.jdbcAuthentication()  
   .dataSource(dataSource)  
   .usersByUsernameQuery(  
     "select username, password, enabled from Users " +  
     "where username=?")  
   .authoritiesByUsernameQuery(  
     "select username, authority from UserAuthorities " +  
     "where username=?");  
}

В этом случае переопределяются только запросы проверки подлинности и обычной авторизации. Но можно также переопределить запрос полномочий группы, вызвав groupAuthoritiesByUsername () с помощью пользовательского запроса.

При замене SQL-запросов по умолчанию, запросами собственной разработки важно придерживаться базового контракта запросов. Все они принимают имя пользователя в качестве единственного параметра. Запрос проверки подлинности выбирает  username, password и enabled. Запрос привилегий выбирает ноль или более строк, содержащих username и authority. Запрос привилегий группы выбирает ноль или более строк, каждая с идентификатором группы, group\_name и authority.

**РАБОТА С ЗАКОДИРОВАННЫМИ ПАРОЛЯМИ**

Сосредоточившись на запросе проверки подлинности, можно увидеть, что пароли пользователей должны храниться в базе данных.  Единственная проблема с этим заключается в том, что если пароли хранятся в текстовом виде, они могут стать доступны для любопытных глаз хакеров. Но если вы закодируете пароли в базе данных, проверка подлинности завершится ошибкой, так как она не будет соответствовать паролю открытого текста, отправленному пользователем.

Для решения этой проблемы, вам нужно указать кодировщик пароля, обратившись к методу passwordEncoder():

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.jdbcAuthentication()  
   .dataSource(dataSource)  
   .usersByUsernameQuery(  
     "select username, password, enabled from Users " +  
     "where username=?")  
   .authoritiesByUsernameQuery(  
     "select username, authority from UserAuthorities " +  
     "where username=?")  
   .passwordEncoder(new StandardPasswordEncoder("53cr3t");  
}

Метод passwordEncoder() принимает любую реализацию интерфейса PasswordEncoder Spring Security. Криптографический модуль Spring Security включает в себя несколько таких реализаций:

-BCryptPasswordEncoder—применяется bcrypt строгое шифрование хэширования (Applies bcrypt strong hashing encryption)

-NoOpPasswordEncoder—не применяется шифрование

-Pbkdf2PasswordEncoder—применяется PBKDF2 шифрования

-SCryptPasswordEncoder—применяется scrypt ширование хэширования (Applies scrypt hashing encryption)

-StandardPasswordEncoder—применяется SHA-256 ширование хэширования (Applies SHA-256 hashing encryption)

Предыдущий код использует StandardPasswordEncoder. Но можно выбрать любую из других реализаций или даже предоставить собственную пользовательскую реализацию, если ни одна из готовых реализаций не соответствует вашим потребностям. Интерфейс PasswordEncoder довольно прост:

public interface PasswordEncoder {  
 String encode(CharSequence rawPassword);  
 boolean matches(CharSequence rawPassword, String encodedPassword);  
}

Независимо от того, какой кодировщик паролей вы используете, важно понимать, что пароль в базе данных никогда не декодируется. Вместо этого пароль, который пользователь вводит при входе в систему, кодируется с использованием того же алгоритма, а затем сравнивается с закодированным паролем в базе данных. Это сравнение выполняется в Passwordencoder методе matches().

В конечном счете, вы будете хранить пользовательские данные Taco Cloud в базе данных. Однако вместо того, чтобы использовать jdbcAuthentication(), у меня есть другой вариант аутентификации. Но прежде чем мы к нему перейдем, давайте посмотрим, как можно настроить Spring  Security полагаться на еще один источник данных пользователей: LDAP (облегченный протокол доступа к каталогам).

4.2.3 Хранилище пользователей в LDAP

Для настройки Spring Security для аутентификации на основе LDAP можно использовать метод ldapAuthentication(). Этот метод является LDAP аналогом jdbcAuthentication(). Следующий метод configure() показывает простую конфигурацию для аутентификации LDAP:

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.ldapAuthentication()  
   .userSearchFilter("(uid={0})")  
   .groupSearchFilter("member={0}");  
}

Методы userSearchFilter() и groupSearchFilter() используются для предоставления фильтров для базовых запросов LDAP, которые используются для поиска пользователей и групп. По умолчанию базовые запросы как для пользователей, так и для групп пусты, что указывает на то, что поиск будет выполняться из корня иерархии LDAP. Но вы можете изменить это, указав базовый запрос:

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.ldapAuthentication()  
   .userSearchBase("ou=people")  
   .userSearchFilter("(uid={0})")  
   .groupSearchBase("ou=groups")  
   .groupSearchFilter("member={0}");  
}

Метод userSearchBase() предоставляет базовый запрос для поиска пользователей. Аналогичным образом, метод groupSearchBase() определяет базовый запрос для поиска групп. Вместо поиска в корневом каталоге в этом примере указывается, что пользователей следует искать в организационном разделе людей. Группы следует искать в том месте, где находится организационный раздел групп.

**НАСТРОЙКА СРАВНЕНИЯ ПАРОЛЕЙ**

Стратегия по умолчанию для аутентификации перед LDAP должна выполнить операцию привязки, аутентифицируя пользователя непосредственно к серверу LDAP. Другой вариант - выполнить операцию сравнения. Для этого необходимо отправить введенный пароль в каталог LDAP и попросить сервер сравнить пароль с атрибутом пароля пользователя. Потому что сравнение делается в LDAP-сервером, пароль остается в тайне.

Если вы предпочитаете аутентификацию путем сравнения паролей, вы можете реализваоть это с помощью метода passwordCompare():

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.ldapAuthentication()  
   .userSearchBase("ou=people")  
   .userSearchFilter("(uid={0})")  
   .groupSearchBase("ou=groups")  
   .groupSearchFilter("member={0}")  
   .passwordCompare();  
}

По умолчанию пароль, указанный в форме входа, будет сравниваться со значением атрибута userPassword в записи LDAP пользователя. Если пароль хранится в другом атрибуте, можно указать имя атрибута пароля с помощью passwordAttribute():

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.ldapAuthentication()  
   .userSearchBase("ou=people")  
   .userSearchFilter("(uid={0})")  
   .groupSearchBase("ou=groups")  
   .groupSearchFilter("member={0}")  
   .passwordCompare()  
   .passwordEncoder(new BCryptPasswordEncoder())  
   .passwordAttribute("passcode");  
}

В этом примере указывается, что атрибут passcode должен сравниваться с заданным паролем. Кроме того, вы также указываете кодировщик паролей. Приятно, что фактический пароль хранится в секрете на сервере при сравнении паролей на стороне сервера. Но попытка ввода пароля все равно передается по проводам на сервер LDAP и может быть перехвачена хакером. Чтобы предотвратить это, можно указать стратегию шифрования, вызвав метод passwordEncoder().

В предыдущем примере пароли шифруются с помощью функции хэширования паролей bcrypt. Это предполагает, что пароли также зашифрованы с помощью bcrypt на сервере LDAP.

**ОБРАЩЕНИЕ К УДАЛЕННОМУ СЕРВЕРУ LDAP**

Единственное, я до сих пор так и не указал местоположение, где находится LDAP-сервер и на котором нужные данные фактически находяться. Вы успешно настроили Spring для аутентификации на сервере LDAP, но где этот сервер?

По умолчанию аутентификация LDAP Spring Security предполагает, что сервер LDAP прослушивает порт 33389 на localhost. Но если ваш сервер LDAP находится на другой машине, вы можете использовать метод contextSource() для настройки его местоположения:

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.ldapAuthentication()  
   .userSearchBase("ou=people")  
   .userSearchFilter("(uid={0})")  
   .groupSearchBase("ou=groups")  
   .groupSearchFilter("member={0}")  
   .passwordCompare()  
   .passwordEncoder(new BCryptPasswordEncoder())  
   .passwordAttribute("passcode")  
   .contextSource()  
   .url("ldap://tacocloud.com:389/dc=tacocloud,dc=com");  
}

Метод contextSource() возвращает ContextSourceBuilder, который, помимо прочего, предлагает метод url(), позволяющий указать расположение сервера LDAP.

**НАСТРОЙКА ВСТРОЕННОГО СЕРВЕРА LDAP**

Если у вас нет сервера LDAP, ожидающего аутентификации, Spring Security может предоставить вам встроенный сервер LDAP. Вместо установки URL-адреса удаленного сервера LDAP можно указать корневой суффикс для встроенного сервера с помощью метода root():

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.ldapAuthentication()  
   .userSearchBase("ou=people")  
   .userSearchFilter("(uid={0})")  
   .groupSearchBase("ou=groups")  
   .groupSearchFilter("member={0}")  
   .passwordCompare()  
   .passwordEncoder(new BCryptPasswordEncoder())  
   .passwordAttribute("passcode")  
   .contextSource()  
   .root("dc=tacocloud,dc=com");  
}

Когда сервер LDAP запустится, он попытается загрузить данные из любых LDIF-файлов, которые он может найти в пути к классам. LDIF (формат обмена данными LDAP) - это стандартный способ представления данных LDAP в текстовом файле. Каждая запись состоит из одной или нескольких строк, каждая из которых содержит имя и значение. Записи отделяются друг от друга пустыми строками.

Если вы предпочитаете, чтобы Spring не рылась в вашем classpath в поисках любых LDIF-файлов, которые он может найти, вы можете более четко указать, какой LDIF-файл загружается, вызвав метод ldif():

@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.ldapAuthentication()  
   .userSearchBase("ou=people")  
   .userSearchFilter("(uid={0})")  
   .groupSearchBase("ou=groups")  
   .groupSearchFilter("member={0}")  
   .passwordCompare()  
   .passwordEncoder(new BCryptPasswordEncoder())  
   .passwordAttribute("passcode")  
   .contextSource()  
   .root("dc=tacocloud,dc=com")  
   .ldif("classpath:users.ldif");  
}

Здесь вы специально просите сервер LDAP загрузить его содержимое о пользователях. ldif-файл в корне classpath приложения. Если вам интересно, вот файл LDIF, который вы можете использовать для загрузки встроенного сервера LDAP с пользовательскими данными:

dn: ou=groups,dc=tacocloud,dc=com  
objectclass: top  
objectclass: organizationalUnit  
ou: groups  
dn: ou=people,dc=tacocloud,dc=com  
objectclass: top  
objectclass: organizationalUnit  
ou: people  
dn: uid=buzz,ou=people,dc=tacocloud,dc=com  
objectclass: top  
objectclass: person  
objectclass: organizationalPerson  
objectclass: inetOrgPerson  
cn: Buzz Lightyear  
sn: Lightyear  
uid: buzz  
userPassword: password  
dn: cn=tacocloud,ou=groups,dc=tacocloud,dc=com  
objectclass: top  
objectclass: groupOfNames  
cn: tacocloud  
member: uid=buzz,ou=people,dc=tacocloud,dc=com

Spring Security’s built-in хранилища пользователей удобны и покрывают некоторые общие случаи использования. Но приложение Taco Cloud нуждается в чем-то особенном. Если готовые пользовательские хранилища не соответствуют вашим потребностям, вам потребуется создать и настроить собственную службу сведений о пользователе.

4.2.4 Собственная служба сведений о пользователе

В последней главе вы остановились на использовании Spring Data JPA в качестве опции сохранения для всех данных taco, ingredient и order. Таким образом, было бы целесообразно сохранить пользовательские данные таким же образом. Если вы сделаете это, данные будут в конечном счете находиться в реляционной базе данных, поэтому вы можете использовать аутентификацию на основе JDBC.  Но было бы еще лучше использовать хранилище данных Spring, используемое для хранения пользователей.

Но сначала о главном. Давайте создадим объект домена и интерфейс репозитория, который представляет и сохраняет информацию о пользователе.

**ОПРЕДЕЛЕНИЕ ДОМЕНА ПОЛЬЗОВАТЕЛЯ И СОХРАНЯЕМОСТИ(PERSISTENCE)**

Когда клиенты Taco Cloud регистрируются в приложении, им необходимо предоставить не только имя пользователя и пароль. Они также дадут вам свое полное имя, адрес и номер телефона. Эта информация может быть использована для различных целей, включая предварительное заполнение формы заказа (не говоря уже о потенциальных маркетинговых возможностях).

Чтобы получить всю эту информацию, создайте класс User, как показано ниже.

Листинг 4.5 Описание сущности user

package tacos;  
  
import java.util.Arrays;  
import java.util.Collection;  
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
import org.springframework.security.core.GrantedAuthority;  
import org.springframework.security.core.authority.  
SimpleGrantedAuthority;  
import org.springframework.security.core.userdetails.UserDetails;  
import lombok.AccessLevel;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import lombok.RequiredArgsConstructor;  
  
@Entity  
@Data  
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)  
@RequiredArgsConstructor  
public class User implements UserDetails {  
 private static final long serialVersionUID = 1L;  
  
 @Id  
 @GeneratedValue(strategy=GenerationType.AUTO)  
 private Long id;  
  
 private final String username;  
 private final String password;  
 private final String fullname;  
 private final String street;  
 private final String city;  
 private final String state;  
 private final String zip;  
 private final String phoneNumber;  
  
 @Override  
 public Collection<? extends GrantedAuthority> getAuthorities() {  
   return Arrays.asList(new SimpleGrantedAuthority("ROLE\_USER"));  
 }  
  
 @Override  
 public boolean isAccountNonExpired() {  
   return true;  
 }  
  
 @Override  
 public boolean isAccountNonLocked() {  
   return true;  
 }  
  
 @Override  
 public boolean isCredentialsNonExpired() {  
   return true;  
 }  
 @Override  
 public boolean isEnabled() {  
   return true;  
 }  
}

Вы, несомненно, заметили, что класс User немного более связанный, чем любая другая сущность, определенная в главе 3.  В дополнение к определению нескольких свойств, пользователь также реализует интерфейс UserDetails из Spring Security.

Implementations UserDetails предоставит некоторую важную информацию о пользователе от framework, например, какие полномочия предоставлены пользователю и включена ли учетная запись пользователя.

Метод getAuthorities() должен возвращать коллекцию полномочий, предоставленных пользователю. Различные методы is\_\_\_Expired () возвращают логическое значение, указывающее, включена или нет учетная запись пользователя.

Для сущности User метод getAuthorities () просто возвращает коллекцию, указывающую, что всем пользователям будут предоставлены полномочия ROLE\_USER. И, по крайней мере, на данный момент, Taco Cloud не нужно отключать пользователей, поэтому все методы is\_\_\_Expired() возвращают true, чтобы указать, что пользователи активны.

Определив сущность User, вы можете определить интерфейс репозитория:

package tacos.data;  
  
import org.springframework.data.repository.CrudRepository;  
import tacos.User;  
  
public interface UserRepository extends CrudRepository<User, Long> {  
 User findByUsername(String username);  
}

Помимо операций CRUD, предоставляемых расширением CrudRepository, UserRepository определяет метод findByUsername(), который будет использоваться в сервисе сведений о пользователе для поиска пользователя по его имени пользователя.

Как вы узнали в главе 3, Spring Data JPA будет автоматически генерировать реализацию этого интерфейса во время выполнения. Таким образом, теперь можно написать пользовательский сервис ,сведений о пользователе использующую этот репозиторий.

СОЗДАНИЕ СЕРВИСА СВЕДЕНИЙ О ПОЛЬЗОВАТЕЛЕ

UserDetailsService Spring Security - это довольно простой интерфейс:

public interface UserDetailsService {  
 UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;  
}

Как вы можете видеть, реализации этого интерфейса даны имя пользователя и должны либо вернуть объект UserDetails или бросить UsernameNotFoundException, если данный логин не появляется никаких результатов.

Поскольку класс User implements UserDetails, а UserRepository предоставляет метод findByUsername(), они идеально подходят для использования в пользовательской реализации UserDetailsService. В следующем листинге показан сервис сведений о пользователе, который вы будете использовать в приложении Taco Cloud.

Листинг 4.6 Определение собственного сервиса сведений о пользователе

package tacos.security;  
  
import org.springframework.beans.factory.annotation.Autowired;  
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 tacos.User;  
import tacos.data.UserRepository;  
  
@Service  
public class UserRepositoryUserDetailsService implements UserDetailsService {  
 private UserRepository userRepo;  
  
 @Autowired  
 public UserRepositoryUserDetailsService(UserRepository userRepo) {  
   this.userRepo = userRepo;  
 }  
  
 @Override  
 public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
   User user = userRepo.findByUsername(username);  
   if (user != null) {  
     return user;  
   }  
   throw new UsernameNotFoundException("User '" + username + "' not found");  
 }  
}

UserRepositoryUserDetailsService внедряется с экземпляром UserRepository через его конструктор. Затем в методе loadByUsername(), вызывается findByUsername () объекта UserRepository для поиска User.

Метод loadByUsername () имеет одно простое правило: он никогда не должен возвращать null. Поэтому, если вызов findByUsername() возвращает null, loadByUsername() будет бросать UsernameNotFoundException. В противном случае будет возвращен найденный пользователь.

Вы видите, что UserRepositoryUserDetailsService аннотируется @Service. Это еще одна из аннотаций стереотипа Spring, которые помечают его для включения в сканирование компонентов Spring, поэтому нет необходимости явно объявлять этот класс как bean. Spring автоматически обнаружит его и создаст его как bean.

Однако по-прежнему необходимо настроить сервис сведений о пользователе с Spring Security. Таким образом, вы вернетесь к методу configure() еще раз:

@Autowired  
private UserDetailsService userDetailsService;  
  
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(userDetailsService);  
}

На этот раз просто вызовите метод userDetailsService (), передав экземпляр UserDetailsService, который был autowired к SecurityConfig.

Как и при аутентификации на основе JDBC, вы можете (и должны) также настроить кодировщик паролей, чтобы пароль мог быть закодирован в базе данных. Это можно сделать, сначала объявив bean типа PasswordEncoder, а затем внедрив (injecting) его в конфигурацию сервиса сведений о пользователе, вызвав passwordEncoder():

@Bean  
public PasswordEncoder encoder() {  
 return new StandardPasswordEncoder("53cr3t");  
}  
  
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(userDetailsService)  
   .passwordEncoder(encoder());  
}

Важно обсудить последнюю строку в методе configure(). Казалось бы, вы вызываете метод encoder() и передаете его возвращаемое значение в passwordEncoder(). В действительности, однако, поскольку метод encoder() аннотируется @Bean, он будет использоваться для объявления компонента PasswordEncoder в контексте приложения Spring. Любые вызовы encoder() будут перехвачены, чтобы возвратить экземпляр bean из контекста приложения.

**РЕГИСТРАЦИЯ ПОЛЬЗОВАТЕЛЕЙ**

Несмотря на то, что Spring Security обрабатывает многие аспекты безопасности, она не принимает непосредственного участия в процессе регистрации пользователей, поэтому вы будете полагаться на Spring MVC для обработки этой задачи. Класс RegistrationController в следующем листинге представляет и обрабатывает регистрационные формы.

Листинг 4.7 Контроллер регистрации пользователей

package tacos.security;  
  
import org.springframework.security.crypto.password.PasswordEncoder;  
import org.springframework.stereotype.Controller;  
import org.springframework.web.bind.annotation.GetMapping;  
import org.springframework.web.bind.annotation.PostMapping;  
import org.springframework.web.bind.annotation.RequestMapping;  
import tacos.data.UserRepository;  
  
@Controller  
@RequestMapping("/register")  
public class RegistrationController {  
 private UserRepository userRepo;  
 private PasswordEncoder passwordEncoder;  
  
 public RegistrationController(UserRepository userRepo, PasswordEncoder passwordEncoder) {  
     this.userRepo = userRepo;  
     this.passwordEncoder = passwordEncoder;  
 }  
  
 @GetMapping  
 public String registerForm() {  
   return "registration";  
 }  
  
 @PostMapping  
 public String processRegistration(RegistrationForm form) {  
   userRepo.save(form.toUser(passwordEncoder));  
   return "redirect:/login";  
 }  
}

Как любой типичный Spring  MVC controller, RegistrationController аннотирован @Controller для того чтобы обозначить его как controller и отметить его для сканирования компонентов (component  scanning). Он также аннотируется @RequestMapping, так что он будет обрабатывать запросы, путь которых /register.

В частности, запрос GET для /register будет обрабатываться методом registerForm (), который просто возвращает логическое имя страницы регистрации. В следующем списке показан шаблон Thymeleaf, определяющий страницу регистрации.

Листинг 4.8 Thymeleaf структура формы регистрации

<!DOCTYPE html>  
<html xmlns="http://www.w3.org/1999/xhtml"  
   xmlns:th="http://www.thymeleaf.org">  
 <head>  
   <title>Taco Cloud</title>  
 </head>  
 <body>  
   <h1>Register</h1>  
   <img th:src="@{/images/TacoCloud.png}"/>  
  <form method="POST" th:action="@{/register}" id="registerForm">  
     <label for="username">Username: </label>  
     <input type="text" name="username"/><br/>  
     <label for="password">Password: </label>  
     <input type="password" name="password"/><br/>  
     <label for="confirm">Confirm password: </label>  
     <input type="password" name="confirm"/><br/>  
     <label for="fullname">Full name: </label>  
     <input type="text" name="fullname"/><br/>  
     <label for="street">Street: </label>  
     <input type="text" name="street"/><br/>  
     <label for="city">City: </label>  
     <input type="text" name="city"/><br/>  
     <label for="state">State: </label>  
     <input type="text" name="state"/><br/>  
     <label for="zip">Zip: </label>  
     <input type="text" name="zip"/><br/>  
     <label for="phone">Phone: </label>  
     <input type="text" name="phone"/><br/>  
    <input type="submit" value="Register"/>  
   </form>  
 </body>  
</html>

После отправки данных формы, запрос HTTP POST будет обработан методом processRegistration(). Объект RegistrationForm, передаваемый в processRegistration(), привязан к данным запроса и описывается следующим классом:

package tacos.security;  
  
import org.springframework.security.crypto.password.PasswordEncoder;  
import lombok.Data;  
import tacos.User;  
  
@Data  
public class RegistrationForm {  
 private String username;  
 private String password;  
 private String fullname;  
 private String street;  
 private String city;  
 private String state;  
 private String zip;  
 private String phone;  
  
 public User toUser(PasswordEncoder passwordEncoder) {  
   return new User(  
     username, passwordEncoder.encode(password),  
     fullname, street, city, state, zip, phone);  
 }  
}

По большей части RegistrationForm - это простой класс с поддержкой Lombok и несколькими свойствами. Но метод toUser() использует эти свойства для создания нового пользовательского объекта, который будет сохранен с помощью внедренного UserRepository.

Вы, без сомнения, заметили, что RegistrationController внедряется с PasswordEncoder. Это точно тот же bean PasswordEncoder который вы объявили ранее. При обработке данных формы, RegistrationController передает ее методу toUser(), который использует ее для кодирования пароля перед сохранением в базу данных. Таким образом, отправленный пароль записывается в закодированном виде, и сервис сведений о пользователе сможет пройти проверку подлинности с использованием этого закодированного пароля.

Теперь приложение Taco Cloud имеет полную поддержку регистрации и аутентификации пользователей. Но если вы запустите его сейчас, вы заметите, что вы даже не можете попасть на страницу регистрации без запроса на вход. Потому что, по умолчанию, все запросы требуют проверки подлинности. Давайте посмотрим, как веб-запросы перехватываются и защищаются, чтобы вы могли исправить эту странную ситуацию с курицей и яйцом.

4.3 Защита веб-запросов

Требования безопасности для Taco Cloud должны требовать, чтобы пользователь аутентифицировался перед созданием тако или размещением заказов. Но домашняя страница, страница входа и страница регистрации должны быть доступны неавторизованным пользователям.

Чтобы настроить эти правила безопасности, позвольте мне представить вам другой метод configure() класса WebSecurityConfigurerAdapter:

@Override  
protected void configure(HttpSecurity http) throws Exception {  
...  
}

Этот метод configure() принимает объект HttpSecurity, который может использоваться для настройки обработки безопасности на веб-уровне. Среди многих вещей, которые вы можете настроить с помощью HttpSecurity следующие:

-Требование соблюдения определенных условий безопасности перед обработкой запроса

-Настройка пользовательской страницы входа

-Возможность выхода пользователей из приложения

-Настройка защиты от подделки межсайтовых запросов

Перехват запросов, чтобы гарантировать, что у пользователя есть надлежащие полномочия, является одной из наиболее распространенных вещей, для которых вы настроите HttpSecurity.  Давайте сделаем, чтобы ваши клиенты Taco Cloud отвечали этим требованиям.

4.3.1 Защита запросов

Необходимо убедиться, что запросы /design и /orders доступны только авторизованным пользователям; все остальные запросы должны быть разрешены для всех пользователей. Следующая реализация configure() делает именно это:

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http.authorizeRequests()  
   .antMatchers("/design", "/orders")  
   .hasRole("ROLE\_USER")  
   .antMatchers(“/”, "/\*\*").permitAll();  
}

Вызов метода AuthorizationRequests() возвращает объект (ExpressionInterceptUrlRegistry), в котором можно указать URL-пути, шаблоны и требования безопасности для этих путей. В этом случае необходимо указать два правила безопасности:

-Запросы /design и /orders должны быть доступны для пользователей с предоставленными полномочиями ROLE\_USER.

-Все запросы должны быть разрешены всем пользователям.

Порядок этих правил важен.  Правила безопасности, объявленные первыми, имеют приоритет над правилами, объявленными ниже.  Если бы вы поменяли порядок этих двух правил безопасности, все запросы имели бы permitAll(), примененный к ним; правило для /design и /orders запросов не имело бы никакого эффекта.

Методы hasRole() и allowAll() - это всего лишь несколько методов объявления требований безопасности для путей запросов. Таблица 4.1 описывает все доступные методы.

Табл. 4.1 Методы настройки для определения способа защиты пути

|  |  |
| --- | --- |
| Метод | Описание |
| access(String) | Разрешает доступ, если данное выражение SpEL имеет значение true |
| anonymous() | Разрешает доступ анонимным пользователям |
| authenticated() | Разрешает доступ аутентифицированным пользователям |
| denyAll() | Безоговорочно запрещает доступ |
| fullyAuthenticated() | Разрешает доступ, если пользователь полностью аутентифицирован (не запоминается) |
| hasAnyAuthority(String...) | Разрешает доступ, если у пользователя есть какие-либо из указанных полномочий |
| hasAnyRole(String...) | Разрешает доступ, если пользователь имеет любую из указанных ролей |
| hasAuthority(String) | Разрешает доступ, если пользователь имеет полномочия |
| hasIpAddress(String) | Разрешает доступ, если запрос поступает с указанного IP-адреса |
| hasRole(String) | Разрешает доступ, если пользователь имеет данную роль |
| not() | Отрицает эффект любого другого метода доступа |
| permitAll() | Разрешает безоговорочный доступ |
| rememberMe() | Разрешает доступ пользователям, прошедшим проверку подлинности с помощью remember-me (запомни меня) |

Большинство методов, описанных в табл. 4.1, предоставляют основные правила безопасности для обработки запросов, но они являются самоограничивающимися и разрешают только правила безопасности, определенные этими методами. Кроме того, можно использовать метод access() для предоставления выражения SpEL для объявления расширенных правил безопасности. Spring Security расширяет SpEL, чтобы включить несколько определенных для безопасности значений и функций, как указано в таблице 4.2.

Таблица 4.2 Spring Security расширения Spring Expression Language

|  |  |
| --- | --- |
| Выражение безопасности | Возвращаемое значение |
| authentication | Объект проверки подлинности пользователя |
| denyAll | Всегда имеет значение false |
| hasAnyRole(list of roles) | true, если пользователь имеет любую из заданных ролей |
| hasRole(role) | true, если пользователь имеет заданную роль |
| hasIpAddress(IP address) | true, если запрос пришел с заданого IP-адреса |
| isAnonymous() | true, если пользователь является анонимным |
| isAuthenticated() | true, если пользователь прошел проверку подлинности |
| isFullyAuthenticated() | true, если пользователь полностью аутентифицирован (не включая аутентификацию с remember-me (запомни меня)) |
| isRememberMe() | true, если пользователь прошел аутентификацию через remember-me (запомни меня) |
| permitAll | Всегда принимает значение true |
| principal | Основной объект пользователя |

Как вы можете видеть, большинство расширений выражений безопасности в таблице 4.2 соответствуют аналогичным методам в таблице 4.1. На самом деле, используя метод access() вместе с выражениями hasRole() и permitAll, вы можете переписать configure() следующим образом.

Листинг 4.9 Использование выражений Spring для определения правил авторизации

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http.authorizeRequests()  
   .antMatchers("/design", "/orders")  
   .access("hasRole('ROLE\_USER')")  
   .antMatchers(“/”, "/\*\*").access("permitAll");  
}

Сначала это может показаться не таким уж большим делом. В конце концов, эти выражения отражают только то, что вы уже сделали с вызовами методов.  Но выражения могут быть гораздо более гибкими. Например, предположите, что (по какой-то сумасшедшей причине) вы только хотели позволить пользователям с полномочиями ROLE\_USER создавать новые тако в вторникам (например, Taco Tuesday); вы могли переписать выражение как показано в этой модифицированной версии configure():

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http.authorizeRequests()  
   .antMatchers("/design", "/orders")  
   .access("hasRole('ROLE\_USER') && " +  
     "T(java.util.Calendar).getInstance().get("+  
     "T(java.util.Calendar).DAY\_OF\_WEEK) == " +  
     "T(java.util.Calendar).TUESDAY")  
   .antMatchers(“/”, "/\*\*").access("permitAll");  
}

С SpEL-based ограничениями безопасности возможности практически безграничны. Бьюсь об заклад, что вы уже придумываете интересные ограничения безопасности, основанные на SpEL.

Потребности в авторизации для приложения Taco Cloud удовлетворяются простым использованием access() и выражений SpEL в листинге 4.9. Теперь давайте посмотрим  настройки страницы входа в систему приложения Taco Cloud.

4.3.2 Создание пользовательской страницы входа

Default-ная страница входа намного лучше, чем неуклюжее диалоговое окно HTTP basic, с которого вы начали, но она все еще довольно просто и не совсем вписывается в стиль остальной части приложения Taco Cloud.

Чтобы заменить встроенную страницу входа, сначала необходимо сообщить Spring Security, по какому пути будет находиться ваша пользовательская страница входа. Это можно сделать, вызвав formLogin() для объекта HttpSecurity, переданного в configure():

@Override  
protected void configure(HttpSecurity http) throws Exception {  
 http.authorizeRequests()  
   .antMatchers("/design", "/orders")  
   .access("hasRole('ROLE\_USER')")  
   .antMatchers(“/”, "/\*\*").access("permitAll")  
   .and()  
   .formLogin()  
   .loginPage("/login");  
}

Обратите внимание, что перед вызовом formLogin() этот раздел конфигурации и предыдущий раздел соединяются вызовом and(). Метод and() означает, что вы завершили настройку авторизации и готовы применить некоторые дополнительные настройки HTTP. Вы будете использовать and() несколько раз, когда начнете новые разделы конфигурации.

После перемычки and() вызовите formLogin(), чтобы начать настройку пользовательской формы входа. Вызов loginPage() после этого определяет путь, где будет предоставлена ваша пользовательская страница входа. Когда Spring Security определит, что пользователь не прошел проверку подлинности и должен войти в систему, он перенаправит их по этому пути.

Теперь необходимо предоставить контроллер, обрабатывающий запросы по этому пути. Поскольку ваша страница входа в систему будет довольно простой (ничего, кроме представления) достаточно легко объявить ее контроллером представления в WebConfig. Следующие метод addViewControllers() устанавливает login page view контроллер наряду с контроллером представления, для "/"  home контроллера:

@Override  
public void addViewControllers(ViewControllerRegistry registry) {  
 registry.addViewController("/").setViewName("home");  
 registry.addViewController("/login");  
}

Наконец, необходимо определить само представление страницы входа. Поскольку вы используете Thymeleaf в качестве механизма шаблонов, следующий шаблон Thymeleaf должен нам подойти:

<!DOCTYPE html>  
<html xmlns="<http://www.w3.org/1999/xhtml>" xmlns:th="http://www.thymeleaf.org">  
 <head>  
   <title>Taco Cloud</title>  
 </head>  
 <body>  
   <h1>Login</h1>  
   <img th:src="@{/images/TacoCloud.png}"/>  
   <div th:if="${error}"> Unable to login. Check your username and password. </div>  
   <p>New here? Click  
     <a th:href="@{/register}">here</a> to register.</p>  
   <!-- tag::thAction[] -->  
   <form method="POST" th:action="@{/login}" id="loginForm">  
   <!-- end::thAction[] -->  
     <label for="username">Username: </label>  
     <input type="text" name="username" id="username" /><br/>  
     <label for="password">Password: </label>  
     <input type="password" name="password" id="password" /><br/>  
     <input type="submit" value="Login"/>  
   </form>  
 </body>  
</html>

Ключевые вещи, которые следует отметить об этой странице входа в систему, - это путь, который она публикует, и имена полей имени пользователя и пароля.  По умолчанию Spring Security прослушивает запросы на вход в /login и ожидает, что поля username и password будут называться username и password. Но это можено настроить. Например, следующая конфигурация настраивает путь и имена полей :

.and().formLogin()  
 .loginPage("/login")  
 .loginProcessingUrl("/authenticate")  
 .usernameParameter("user")  
 .passwordParameter("pwd")

Здесь вы указываете, что Spring Security должна прослушивать запросы /authenticate для обработки запросов на вход. Кроме того, поля username и password теперь должны быть названы user и pwd.

По умолчанию успешный вход приведет пользователя непосредственно к странице, на которую он переходил, когда Spring Security определила, что ему необходимо залогиниться. Если бы пользователь должен был перейти непосредственно на страницу входа в систему, успешный вход в систему привел бы его к корневому пути (например, к домашней странице). Но вы можете изменить это, указав страницу на которую переходить при успешном входе в систему по умолчанию:

.and().formLogin()  
 .loginPage("/login")  
 .defaultSuccessUrl("/design")

Как настроено здесь, если пользователь должен был успешно войти в систему после прямого перехода на страницу входа, он будет перенаправлен на страницу / design.

При желании вы можете принудительно заставить пользователя перейти на страницу design после входа в систему, даже если он находился на другой странце до входа в систему, передав значение true в качестве второго параметра в defaultSuccessUrl:

.and().formLogin()  
 .loginPage("/login")  
 .defaultSuccessUrl("/design", true)

Теперь, когда вы имеете дело с пользовательской страницей входа, давайте перейдем на другую сторону монеты аутентификации и посмотрим, как вы можете позволить пользователю выйти из системы.

4.3.3 Выход из системы

Выход из системы так же важен, как и вход в приложение. Чтобы выйти, -вам просто нужно вызвать logout для объекта HttpSecurity:

.and().logout()  
 .logoutSuccessUrl("/")

Это настраивает фильтр безопасности, который перехватывает запросы POST для /logout. Поэтому, чтобы обеспечить возможность выхода из системы, вам просто нужно добавить форму и кнопку выхода из системы в представлениях вашего приложения:

<form method="POST" th:action="@{/logout}">  
 <input type="submit" value="Logout"/>  
</form>

Когда пользователь нажимает кнопку, его сеанс очищается, и он выходит из приложения. По умолчанию он будет перенаправлен на страницу входа, где сможет снова войти в систему. Но если вы хотите, чтобы он был отправлен на другую страницу, вы можете вызвать logoutSucessFilter (), чтобы указать другую целевую страницу после выхода:

.and().logout()  
 .logoutSuccessUrl("/")

В этом случае, пользователи будут отправлены на главную страницу после выхода.

4.3.4 Предотвращение подделки межсайтовых запросов

Подделка межсайтовых запросов (CSRF) является обычной атакой безопасности. Он включает в себя предоставление пользователю кода на злонамеренно разработанной веб-странице, которая автоматически (и обычно тайно) отправляет форму другому приложению от имени пользователя, который часто является жертвой атаки. Например, пользователю может быть представлена форма на веб-сайте злоумышленника, которая автоматически публикует URL-адрес на банковском веб-сайте пользователя (который предположительно плохо спроектирован и уязвим для такой атаки) для перевода денег. Пользователь может даже не знать, что атака произошла, пока не заметит пропажу денег со своего счета.

Для защиты от таких атак приложения могут создать токен CSRF при отображении формы, поместить этот токен в скрытое поле, а затем хранить его для последующего использования на сервере. После отправки формы токен отправляется обратно на сервер вместе с остальными данными формы. Затем запрос перехватывается сервером и сравнивается с первоначально созданным токеном. Если токены совпадают, запрос разрешается обработать. В противном случае форма должна быть отрисована вредоносным веб-сайтом без знания токена, сгенерированного сервером.

К счастью, Spring Security имеет встроенную защиту CSRF. Еще более удачным является то, что она включена по умолчанию, и вам не нужно явно настраивать её. Необходимо только убедиться, что все формы, отправляемые приложением, содержат поле с именем \_csrf, содержащее токен CSRF.

Spring Security даже упрощает это, помещая токен CSRF в атрибут запроса с именем \_csrf. Таким образом можно отобразить токен CSRF в скрытом поле в шаблоне Thymeleaf:

<input type="hidden" name="\_csrf" th:value="${\_csrf.token}"/>

Если вы используете библиотеку тегов Spring MVC’s JSP или Thymeleaf с Spring Security dialect, вам даже не нужно явно создавать скрытое поле. Скрытое поле будет создано автоматически.

В Thymeleaf, вам просто нужно убедиться, что один из атрибутов элемента <form> с префиксом в качестве атрибута Thymeleaf. Это обычно не проблема, так как довольно часто Thymeleaf отображает путь как относительный контекст.  Например, атрибут th:action-это все, что вам нужно для отображения скрытого поля Thymeleaf:

<form method="POST" th:action="@{/login}" id="loginForm">

Можно отключить поддержку CSRF, но я не решаюсь показать вам, как это сделать. CSRF-защита важна и легко обрабатывается в формах, поэтому нет причин ее отключать. Но если вы настаиваете на его отключении, вы можете сделать это, вызвав disable() следующим образом:

.and().csrf().disable()

Опять же, я предупреждаю вас не отключать защиту CSRF, особенно для производственных приложений.

Вся ваша безопасность веб-уровня теперь настроена для Taco Cloud.  Помимо прочего, теперь у вас есть настраиваемая страница входа и возможность аутентификации пользователей в репозитории пользователей с поддержкой JPA. Теперь давайте посмотрим, как можно получить информацию о зарегистрированных пользователей.

4.4 Определение пользователя

Часто недостаточно просто знать, что пользователь вошел в систему. Обычно важно также знать, кто он, чтобы вы могли представить вид и функционал приложения в зависимости от вошедшего пользователя.

Например, в OrderController при первоначальном создании объекта Order, связанного с формой заказа, было бы неплохо предварительно заполнить заказ именем и адресом пользователя, чтобы ему не пришлось повторно вводить его для каждого заказа. Возможно, еще более важно, что при сохранении заказа необходимо связать сущность Order с пользователем, создавшим заказ.

Для достижения желаемой связи между сущностью Order и сущностью User необходимо добавить новое свойство в класс Order:

@Data  
@Entity  
@Table(name="Taco\_Order")  
public class Order implements Serializable {  
...  
  
@ManyToOne  
private User user;  
...  
}

Аннотация @ManyToOne этого свойства указывает, что заказ принадлежит одному пользователю, и, наоборот, что у пользователя может быть много заказов.  (Поскольку вы используете Lombok, вам не нужно явно определять методы доступа для свойства.)

В OrderController метод processOrder() отвечает за сохранение заказа. Его необходимо изменить, чтобы определить, кто является аутентифицированным пользователем, и вызвать setUser () у объекта Order, чтобы связать заказ с пользователем.

Существует несколько способов определить пользователя. Вот несколько из наиболее распространенных способов:

-Inject основные(Principal) объекты в метод контроллера

-Inject объект аутентификации(Authentication) в метод контроллера

-Использовать SecurityContextHolder, чтобы получить в контексте безопасности

-Используйте аннотированный метод @AuthenticationPrincipal

Например, можно изменить processOrder(), чтобы он принимал java.security.Principal в качестве параметра.  Затем вы можете использовать основное(principal) имя для поиска пользователя используя UserRepository:

@PostMapping  
public String processOrder(@Valid Order order, Errors errors, SessionStatus sessionStatus,  
   Principal principal) {  
 ...  
 User user = userRepository.findByUsername(  
 principal.getName());  
 order.setUser(user);  
 ...  
}

Это отлично работает, но засоряет код, который иначе не связан с безопасностью с кодом безопасности. Можно урезать часть кода безопасности, изменив processOrder(), чтобы он принимал Authentication объект в качестве параметра вместо Principal:

@PostMapping  
public String processOrder(@Valid Order order, Errors errors, SessionStatus sessionStatus,  
   Authentication authentication) {  
 ...  
 User user = (User) authentication.getPrincipal();  
 order.setUser(user);  
 ...  
}

С Authentication в руках, вы можете вызвать getPrincipal(), чтобы получить основной (principal) объект, который в этом случае является User.  Обратите внимание, что getPrincipal() возвращает java.util.Object, поэтому вам нужно привести его к User.

Однако, возможно, самым чистым решением является просто принимать на вход объект User в методе processOrder(), но аннотировать его @AuthenticationPrincipal, чтобы он был субъектом проверки подлинности:

@PostMapping  
public String processOrder(@Valid Order order, Errors errors, SessionStatus sessionStatus,  
   @AuthenticationPrincipal User user) {  
 if (errors.hasErrors()) {  
   return "orderForm";  
 }  
 order.setUser(user);  
 orderRepo.save(order);  
 sessionStatus.setComplete();  
 return "redirect:/";  
}

Что хорошо в @AuthenticationPrincipal, так это то, что он не требует приведения (как с Authentication), и он ограничивает код безопасности самой аннотацией.  К моменту получения объекта User в processOrder() он готов к использованию для Оrder.

Есть еще один способ определить, кто является аутентифицированным пользователем, хотя это немного грязно в том смысле, что он очень тяжел для кода, специфичного для безопасности. Вы можете получить объект аутентификации из контекста безопасности и затем запросить его участника (principal) следующим образом:

Authentication authentication = SecurityContextHolder.getContext().getAuthentication();  
User user = (User) authentication.getPrincipal();

Несмотря на то, что этот фрагмент имеет большой объем кода для обеспечения безопасности, он имеет одно преимущество перед другими описанными подходами: его можно использовать в любом месте приложения, а не только в методах обработчика контроллера. Это делает его пригодным для использования на более низких уровнях кода.

Итог:

--Spring Security autoconfiguration  отличный способ начать работу с безопасностью, но большинству приложений необходимо явно настроить безопасность для удовлетворения своих уникальных требований безопасности.

-User  details могут управляться в хранилищах пользователей, поддерживаемых реляционными базами данных,

LDAP или полностью настраиваемыми реализациями.

-Spring Security автоматически защищает от CSRF-атак.

-Информация об аутентифицированном пользователе может быть получена через объект SecurityContext (возвращается из SecurityContextHolder.getContext () ) или внедряется в контроллеры с помощью @AuthenticationPrincipal.

[Spring in Action Covers Spring 5.0 перевод на русский. Глава 5](https://beginner-mercenary.blogspot.com/2019/05/spring-in-action-covers-spring-50-5.html)

5. Работа со свойствами конфигурации

В этой главе рассматриваются

* Тонкая настройка автоконфигурирования bean-ов
* Применение свойств конфигурации к компонентам приложения
* Работа с Spring профилями.

Вы помните те времена, когда iPhone впервые вышел? Небольшой кусок металла и стекла едва ли соответствовал описанию того, что мир представлял себе как телефон. И все же, он стал пионером современной эпохи смартфонов, изменив все в том, как мы общаемся. Хотя сенсорные телефоны во многих отношениях проще и мощнее, чем их предшественник, флип-телефон, когда iPhone был впервые анонсирован, было трудно представить, как устройство с одной кнопкой может использоваться для звонков.

В некотором смысле автоконфигурация Spring Boot выглядит также. Автоматическая конфигурация значительно упрощает разработку приложений Spring. Но после десятилетия установки значений свойств в конфигурации Spring XML и вызова методов setter в экземплярах bean не сразу видно, как установить свойства bean, для которых нет явной конфигурации.

К счастью, Spring Boot предоставляет способ со свойствами конфигурации. Свойства конфигурации - это не что иное, как свойства компонентов в контексте приложения Spring, которые можно задать из нескольких источников свойств, включая системные свойства JVM, аргументы командной строки и переменные среды.

В этой главе вы сделаете шаг назад от реализации новых функций в приложении Taco Cloud, чтобы изучить свойства конфигурации. То, что вы узнаете, несомненно, окажется полезным по мере продвижения вперед в последующих главах. Мы начнем с того, как использовать свойства конфигурации для точной настройки того, что Spring Boot автоматически настраивает.

5.1 Тонкая настройка автоконфигурации

Прежде чем мы слишком глубоко погрузимся в свойства конфигурации, важно установить, что в Spring: есть два разных (но связанных) типа конфигураций:

-Bean wiring - конфигурация, которая объявляет компоненты приложения, которые будут созданы как bean в контексте приложения Spring и как они должны быть внедрены друг в друга.

-Property injection - конфигурация, задающая значения для компонентов в контексте приложения Spring.

И в конфигурации Spring на основе XML и Java конфигурации, эти два типа конфигураций часто объявляются явно в одном и том же месте. В конфигурации Java метод @Bean - аннотированный, вероятно, создаст экземпляр bean, а затем установит значения его свойств. Например, рассмотрим следующий метод @Bean, объявляющий источник данных для встроенной базы данных H2:

@Bean

public DataSource dataSource() {

  return new EmbeddedDataSourceBuilder()

     .setType(H2)

     .addScript("taco\_schema.sql")

    .addScripts("user\_data.sql", "ingredient\_data.sql")

    .build();

}

Здесь методы addScript()  и addScripts() задают некоторые строковые свойства с именами SQL-скриптов, которые должны применяться к базе данных после того, как источник данных готов. Таким образом вы можете настроить bean-компонент DataSource, если вы не используете Spring Boot. Автоконфигурация делает этот метод совершенно ненужным.

Если зависимость (dependency) H2 доступна в пути к классам во время выполнения, Spring Boot автоматически создает соответствующий компонент DataSource в контексте приложения Spring. Bean применяет сценарии SQL schema.sql и data.sql.

Но что, если вы хотите назвать сценарии SQL как-нибудь иначе? Или что, если вам нужно указать более двух сценариев SQL? Именно здесь вступают в силу свойства конфигурации. Но прежде чем вы сможете начать использовать свойства конфигурации, вам необходимо понять, откуда эти свойства берутся.

5.1.1 Понимание абстракции среды Spring

Абстракция среды Spring - это универсальный магазин для любого настраиваемого свойства. Он абстрагирует происхождение свойств, так что bean-ы, нуждающиеся в этих свойствах, могут получать их из среды Spring. Окружающая среда Spring получает их от нескольких источников свойств, включая:

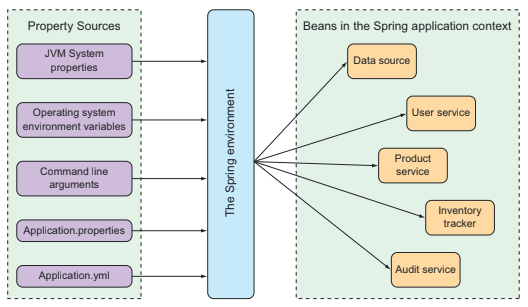
-Свойства системы JVM

-Переменные среды операционной системы

-Аргумент командной строки

-Конфигурационные файлы приложения

Затем он агрегирует эти свойства в один источник, из которого можно производить внедрения Spring bean-ов. На рис. 5.1 показано, как свойства из источников свойств перетекают через абстракцию среды Spring в Spring beans.



**Рисунок 5.1 Spring окружение подтягивает свойства из различных источников, и делает их доступными для bean-ов в контексте приложения.**

Bean-компоненты, которые автоматически настраиваются Spring Boot, настраиваются свойствами, полученными из среды Spring. В качестве простого примера предположим, что вы хотите, чтобы базовый контейнер сервлета приложения прослушивал запросы на каком-либо порту, отличном от порта по умолчанию 8080. Для этого укажите другой порт, установив свойство server.port в src/main/resources/application.properties:

server.port=9090

Лично я предпочитаю использовать YAML при настройке свойств конфигурации. Поэтому, вместо того, чтобы использовать application.properties, я мог бы создал server.port в src / main/resources/application.yml:

server:

   port: 9090

Если вы предпочитаете настраивать это свойство извне, вы также можете указать порт при запуске приложения с помощью аргумента командной строки:

$ java -jar tacocloud-0.0.5-SNAPSHOT.jar --server.port=9090

Если вы хотите, чтобы приложение всегда запускалось на определенном порту, вы можете установить его один раз в качестве переменной среды операционной системы:

$ export SERVER\_PORT=9090

Обратите внимание, что при установке свойств в качестве переменных среды стиль именования немного отличается, чтобы учесть ограничения, накладываемые операционной системой на имена переменных среды. Все нормально. Spring может интерпретировать SERVER\_PORT как server.port без проблем.

Как я уже сказал, существует несколько способов настройки свойств конфигурации. И когда мы перейдем к главе 14, вы увидите еще один способ установки свойств конфигурации на централизованном сервере конфигурации. Фактически, есть несколько сотен свойств конфигурации, которые вы можете использовать для настройки, в том числе и для настройки поведения Spring bean-ов. Вы уже видели несколько: server.port в этой главе и security.user.name и security.user.password в предыдущей главе.

Невозможно изучить все доступные свойства конфигурации в этой главе. Тем не менее, давайте рассмотрим несколько наиболее полезных свойств конфигурации, с которыми вы обычно сталкиваетесь. Мы начнем с нескольких свойств, которые позволяют настроить источник данных с автоматической настройкой.

5.1.2 Конфигурация источника данных

На данный момент приложение Taco Cloud все еще не завершено, но у вас будет еще несколько глав, чтобы позаботиться об этом, прежде чем вы будете готовы развернуть приложение. Таким образом, встроенная база данных H2, которую вы используете в качестве источника данных, идеально подходит для ваших нужд. Но как только вы возьмете приложение в производство, вы, вероятно, захотите рассмотреть более постоянное решение для базы данных.

Хотя вы можете явно настроить свой собственный компонент источника данных, это обычно не требуется. Вместо этого проще настроить URL-адрес и учетные данные для базы данных с помощью свойства конфигурации. Например, если бы вы должны были начать использовать базу данных MySQL, вы могли бы добавить следующие свойства конфигурации к приложению в формате YML:

spring:

   datasource:

     url: jdbc:mysql://localhost/tacocloud

       username: tacodb

       password: tacopassword

Хотя вам нужно будет добавить соответствующий драйвер JDBC в сборку, вам обычно не нужно будет указывать класс драйвера JDBC; Spring Boot может выяснить это из структуры URL базы данных. Но если возникнет проблема, вы можете установить свойство spring.datasource.driver-class-name :

spring:

 datasource:

   url: jdbc:mysql://localhost/tacocloud

   username: tacodb

   password: tacopassword

   driver-class-name: com.mysql.jdbc.Driver

Spring Boot использует эти данные подключения при автоконфигурировании компонента DataSource. Компонент DataSource будет объединен с помощью пула соединений JDBC Tomcat, если он доступен в пути к классам. Если нет, Spring Boot ищет и использует одну из этих реализаций пула соединений в пути к классам:

-HikariCP

-Commons DBCP 2

Хотя это единственные параметры пула соединений, доступные через автоконфигурацию, вы всегда можете явно настроить компонент источника данных для использования любой реализации пула соединений, которую вы хотите.

Ранее в этой главе мы предположили, что можно указать сценарии инициализации базы данных для запуска при запуске приложения. В этом случае,будут полезны свойства spring.datasource.schema и spring.datasource.data:

spring:

 datasource:

 schema:

   - order-schema.sql

   - ingredient-schema.sql

   - taco-schema.sql

   - user-schema.sql

 data:

   - ingredients.sql

Возможно, явная конфигурация источника данных не ваш стиль. Вместо этого, возможно, вы предпочтете настроить источник данных в JNDI и Spring найдет его там. В этом случае настройте источник данных, настроив spring.datasource.jndi-name:

spring:

 datasource:

   jndi-name: java:/comp/env/jdbc/tacoCloudDS

Если вы установите свойство spring.datasource.jndi-name, другие свойства соединения с источником данных (если задано) игнорируются.

5.1.3 Настройка встроенного сервера

Вы уже видели, как установить порт контейнера сервлета, установив server.port. Я не показал вам, что происходит, если server.port 0:

server:

 port: 0

Но базовый сервер-это не просто порт. Одна из наиболее распространенных вещей, которые вам нужно сделать с базовым контейнером, - настроить его для обработки HTTPS-запросов. Чтобы сделать это, первое, что вы должны сделать, это создать хранилище ключей с помощью утилиты командной строки keytool JDK:

$ keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA

Вам будет задано несколько вопросов о вашем имени и организации, большинство из которых не имеют никакого отношения к результату. Но когда вас спросят пароль, помните (а лучше запишите), что вы задаете. Для этого примера я выбрал letmein в качестве пароля.

Затем вам нужно будет установить несколько свойств для включения HTTPS на встроенном сервере. Вы можете указать их все в командной строке, но это было бы ужасно неудобно. Вместо этого вы, вероятно, установите их в файле application.properties или In application.yml формате YML. В In application.yml, свойства могут выглядеть следующим образом:

server:

 port: 8443

 ssl:

   key-store: file:///path/to/mykeys.jks

   key-store-password: letmein

   key-password: letmein

Тут свойство server.port имеет значение 8443, что является общепринятым значением для разработки HTTPS-серверов. Свойство server.ssl.key-store должно быть задано значением пути, где расположен создаваемый файл keystore. Здесь он задан с file:// URL для загрузки его из файловой системы, но если вы упакуете его в файл JAR приложения, вы должны использовать classpath: URL для ссылки на него. И оба свойства server.ssl.key-store-password и задаются значением пароля, который был задан при создании хранилища ключей.

При наличии этих свойств приложение должно прослушивать HTTPS-запросы на порту 8443. В зависимости от используемого браузера может появиться предупреждение о том, что сервер не может подтвердить свою личность. На это можно не обращать внимание при работе с localhost во время разработки.

5.1.4 Конфигурация логирования

Большинство приложений обеспечивают некоторую форму логирования. И даже если ваше приложение ничего не регистрирует напрямую, библиотеки, которые использует ваше приложение, обязательно логируют свою активность.

По умолчанию Spring Boot настраивает ведение журнала через Logback (http://logback.qos.ch) для записи на консоль на информационном уровне. Вероятно, вы уже видели множество записей INFO уровня в журналах приложений при запуске приложения и других примерах.

Для полного контроля над конфигурацией ведения журнала можно создать logback.xml-файл в корневом каталоге пути к классам (в src/main/resources). Вот пример простого logback.xml-файл, который можно использовать:

<configuration>

 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

   <encoder>

     <pattern>

       %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n

    </pattern>

   </encoder>

 </appender>

 <logger name="root" level="INFO"/>

 <root level="INFO">

   <appender-ref ref="STDOUT" />

 </root>

</configuration>

Помимо шаблона, используемого для ведения журнала, эта конфигурация Logback более или менее эквивалентна используемой по умолчанию, которое вы получите, если у вас нет logback.xml файл. Но путем редактирования logback.XML вы можете получить полный контроль над log файлами приложения.

ПРИМЕЧАНИЕ: Особенности настройки logback.xml выходят за рамки этой книги. Обратитесь к документации Logback для получения дополнительной информации.

Наиболее распространенные изменения, которые вы внесете в конфигурацию логирования, - это изменение уровней логирования и, возможно, указание файла, в который должны быть записаны логи. С помощью свойств конфигурации Spring Boot вы можете вносить эти изменения без необходимости создания файла  logback.xml.

Чтобы установить уровень логирования, нужно задать свойства, которые начинаются с logging.level, за которым следует имя регистратора, для которого вы хотите установить уровень ведения журнала. Например, предположим, что вы хотите установить корневой  уровень (root logging level) логирования WARN, а логирование Spring Security в уровень DEBUG. Следующие записи в application.yml позаботятся об этом для вас:

logging:

 level:

   root: WARN

     org:

      springframework:

        security: DEBUG

При необходимости можно свернуть имя пакета Spring Security в одну строку для удобства чтения:

logging:

 level:

   root: WARN

     org.springframework.security: DEBUG

Теперь предположим, что вы хотите записать log-и в файл TacoCloud.log в /var/logs/. Свойства logging.path и logging.file могут помочь в этом:

logging:

 path: /var/logs/

 file: TacoCloud.log

 level:

   root: WARN

   org:

     springframework:

       security: DEBUG

Предполагая, что приложение имеет разрешения на запись в /var/logs/, записи журнала будут записаны в /var/logs/TacoCloud.log. По умолчанию log файлы создаются новые по достижении размера 10 МБ.

5.1.5 Использование специальных значений свойств

При настройке свойств вы не ограничиваетесь объявлением их значений как жестко закодированных строковых и числовых значений. Вместо этого можно получить их значения из других свойств конфигурации.

Например, предположим (по какой-либо причине), что вы хотите установить свойство с именем greeting.welcome на основе значение другого свойства с именем spring.application.name. Для этого при настройке приветствия можно использовать маркеры-заполнители $ {} для задания свойства greeting.welcome:

greeting:

 welcome: ${spring.application.name}

Вы даже можете использовать этот заполнитель как часть другого текста:

greeting:

 welcome: You are using ${spring.application.name}.

Как вы уже видели, настройка собственных компонентов Spring со свойствами конфигурации позволяет легко вводить значения в свойства этих компонентов и настраивать автоконфигурацию. Свойства конфигурации не являются эксклюзивными для bean-компонентов, которые создает Spring. Приложив небольшое усилие, вы можете использовать свойства конфигурации в ваших собственных bean-компонентах. Посмотрим как.

5.2 Создание ваших собственных свойств конфигурации

Как я упоминал ранее, свойства конфигурации - это не что иное, как свойства bean-компонентов, предназначенных для получения конфигураций из абстракции среды Spring. Что я не упомянул, так это то, как эти компоненты предназначены для использования этих конфигураций.

Для поддержки внедрения свойств конфигурации Spring Boot предоставляет аннотацию @ConfigurationProperties. При указании аннотации в любом Spring bean-компонен туказывается, что свойства этого bean-компонента могут быть внедрены из свойств среды Spring.

Чтобы продемонстрировать, как работает @ConfigurationProperties, предположим, что вы добавили следующий метод в OrderController, чтобы вывести список прошлых заказов аутентифицированного пользователя:

@GetMapping

public String ordersForUser(

 @AuthenticationPrincipal User user, Model model) {

   model.addAttribute("orders",

       orderRepo.findByUserOrderByPlacedAtDesc(user));

   return "orderList";

}

Наряду с этим вы также добавили необходимый метод findByUser() в OrderRepository:

List<Order> findByUserOrderByPlacedAtDesc(User user);

Обратите внимание, что этот метод хранилища (репозитория) имеет в названии OrderByPlacedAtDesc. Часть OrderBy указывает свойство, по которому будут упорядочены результаты - в данном случае, свойство placeAt. Desc в конце заставляет упорядочение быть в порядке убывания. Таким образом, список возвращаемых заказов будет отсортирован от самых последних до наименее последних.

Этот метод контроллера может быть полезен после того, как пользователь разместил несколько заказов. Но это может стать немного громоздким для самых заядлых ценителей тако. Несколько заказов, отображаемых в браузере, полезны; бесконечный список из сотен заказов - это просто шум. Допустим, вы хотите ограничить количество отображаемых заказов самыми последними 20 заказами. Вы можете изменить ordersForUser()

@GetMapping

public String ordersForUser(

 @AuthenticationPrincipal User user, Model model) {

   Pageable pageable = PageRequest.of(0, 20);

   model.addAttribute("orders",

     orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

   return "orderList";

}

вместе с соответствующими изменениями в OrderRepository:

List<Order> findByUserOrderByPlacedAtDesc(User user, Pageable pageable);

Вы изменили сигнатуру метода findByUserOrderByPlacedAtDesc(), чтобы принимать на вход Pageable в качестве параметра. Pageable - это способ Spring Data выбрать некоторое подмножество результатов по номеру страницы и размеру страницы. В методе контроллера ordersForUser() вы создали объект PageRequest, который реализовал Pageable для запроса первой страницы (нулевой страницы) с размером страницы 20, чтобы получить до 20 самых последних размещенных заказов для пользователя.

Хотя это работает фантастически, мне немного неловко, что вы жестко закодировали размер страницы. Что, если позже вы решите, что 20 - это слишком много заказов, и вы решите изменить его на 10? Поскольку он жестко запрограммирован, вам придется пересобрать и повторно развернуть приложение.

Вместо того, чтобы жестко задавать размер страницы, вы можете установить его с помощью пользовательского свойства конфигурации. Сначала вам нужно добавить новое свойство с именем pageSize в OrderController, а затем аннотировать OrderController с помощью @ConfigurationProperties, как показано в следующем листинге.

**Листинг 5.1 Включение свойств конфигурации в OrderController**

@Controller

@RequestMapping("/orders")

@SessionAttributes("order")

@ConfigurationProperties(prefix="taco.orders")

public class OrderController {

 private int pageSize = 20;

 public void setPageSize(int pageSize) {

   this.pageSize = pageSize;

 }

 ...

 @GetMapping

 public String ordersForUser(

      @AuthenticationPrincipal User user, Model model) {

    Pageable pageable = PageRequest.of(0, pageSize);

    model.addAttribute("orders",

    orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

    return "orderList";

 }

}

Наиболее значительным изменением, внесенным в листинг 5.1, является добавление аннотации @ConfigurationProperties. Его префиксный атрибут имеет значение taco.orders, что означает, что при установке свойства pageSize необходимо использовать свойство конфигурации с именем taco.orders.pageSize.

Новое свойство pageSize по умолчанию равно 20. Но вы можете легко изменить его на любое желаемое значение, установив свойство taco.orders.pageSize. Например, вы можете установить это свойство в application.yml следующим образом:

taco:

 orders:

   pageSize: 10

Или, если вам нужно сделать быстрые изменения во время работы, вы можете сделать это без необходимости rebuild и redeploy приложения, задав свойство taco.orders.pageSize в качестве переменной среды:

$ export TACO\_ORDERS\_PAGESIZE=10

Любое средство, с помощью которого можно установить свойство конфигурации, можно использовать для настройки размера страницы последних заказов. Далее мы рассмотрим, как устанавливать данные конфигурации в хранилищах свойств (property holders).

5.2.1 Определение хранилища свойств конфигурации (configuration properties holders)

Ничто не указывает на то, что @ConfigurationProperties должен быть установлен на контроллере или любом другом конкретном компоненте. @ConfigurationProperties на самом деле часто размещаются на bean-компонентах, единственная цель которых в приложении - хранить данные конфигурации. Это исключает детали конфигурации из контроллеров и других классов приложений. Это также упрощает совместное использование общих свойств конфигурации несколькими компонентами, которые могут использовать эту информацию.

В случае свойства pageSize в OrderController вы можете перенести его в отдельный класс. Следующий листинг демонстрирует такой класс OrderProps.

**Листинг 5.2. Извлечение pageSize в класс хранилища свойств**

package tacos.web;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.stereotype.Component;

import lombok.Data;

@Component

@ConfigurationProperties(prefix="taco.orders")

@Data

public class OrderProps {

 private int pageSize = 20;

}

Как и в случае с OrderController, для свойства pageSize по умолчанию установлено значение 20, а для OrderProps добавлен @ConfigurationProperties с префиксом taco.orders. Он также помечен @Component, так что сканирование компонентов Spring автоматически обнаружит его и создаст как компонент в контексте приложения Spring. Это важно, так как следующим шагом является внедрение bean-компонента OrderProps в OrderController.

В хранилище конфигурации нет ничего особенного. Это bean, чьи свойства поступают из среды Spring. Они могут быть введены в любой другой компонент, которому нужны эти свойства. Для OrderController это означает удаление свойства pageSize из OrderController и вместо этого внедрение и использование компонента OrderProps:

@Controller

@RequestMapping("/orders")

@SessionAttributes("order")

public class OrderController {

 private OrderRepository orderRepo;

 private OrderProps props;

 public OrderController(OrderRepository orderRepo,

     OrderProps props) {

   this.orderRepo = orderRepo;

   this.props = props;

 }

 ...

 @GetMapping

 public String ordersForUser(

     @AuthenticationPrincipal User user, Model model) {

   Pageable pageable = PageRequest.of(0, props.getPageSize());

   model.addAttribute("orders",

     orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

   return "orderList";

 }

 ...

}

Теперь OrderController больше не отвечает за обработку своих собственных свойств конфигурации. Это делает код OrderController немного аккуратнее и позволяет повторно использовать свойства из OrderProps в любом другом компоненте, который может в них нуждаться. Кроме того, вы группируете свойства конфигурации, которые относятся к заказам в одном месте: класс OrderProps. Если вам нужно добавить, удалить, переименовать или иным образом изменить содержащиеся в нем свойства, вам нужно только произвести эти изменения в OrderProps.

Например, давайте представим, что вы используете свойство pageSize в нескольких других bean-компонентах, когда вдруг решили, что было бы лучше применить некоторую проверку к этому свойству, чтобы ограничить его значения не менее чем 5 и не более 25. Без отдельного bean-компонента вам нужно будет применить аннотации проверки к OrderController, свойству pageSize и во всех других классах, использующих это свойство. Но поскольку вы создали pageSize в OrderProps, вам нужно только внести изменения в OrderProps:

package tacos.web;

import javax.validation.constraints.Max;

import javax.validation.constraints.Min;

import org.springframework.boot.context.properties.ConfigurationProperties;

import org.springframework.stereotype.Component;

import org.springframework.validation.annotation.Validated;

import lombok.Data;

@Component

@ConfigurationProperties(prefix="taco.orders")

@Data

@Validated

public class OrderProps {

   @Min(value=5, message="must be between 5 and 25")

   @Max(value=25, message="must be between 5 and 25")

   private int pageSize = 20;

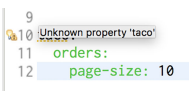
}

//end::validated[]

Хотя вы могли бы так же легко применить аннотации @Validated, @Min и @Max к OrderController (и любым другим bean-компонентам, которые могут быть внедрены с помощью OrderProps), это просто намного больше загромождает OrderController. С помощью компонента-хранилища свойства конфигурации вы собрали спецификацию свойств конфигурации в одном месте, оставив классы, которым эти свойства нужны, относительно чистыми.

5.2.2 Объявление метаданных свойства конфигурации

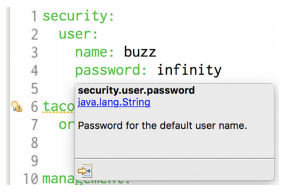
В зависимости от вашей среды IDE вы, возможно, заметили, что запись taco.orders.pageSize в application.yml (или application.properties) имеет предупреждение о чем-то вроде неизвестного свойства «taco». Это предупреждение появляется из-за отсутствия метаданных, касающихся только что созданного свойства конфигурации. На рисунке 5.2 показано, как это выглядит, когда я наведу курсор мыши на тако-часть свойства в Spring Tool Suite.



**Рисунок 5.2 Предупреждение о отсутствующих метаданных свойства конфигурации**

Метаданные свойств конфигурации не являются обязательными и не мешают работе свойств конфигурации. Но метаданные могут быть полезны для предоставления некоторой минимальной документации по свойствам конфигурации, особенно в IDE.

Например, при наведении курсора на свойство security.user.password я вижу то, что показано на рисунке 5.3. Хотя помощь при наведении мыши минимальна, ее может быть достаточно, чтобы понять, для чего используется свойство и как его использовать.



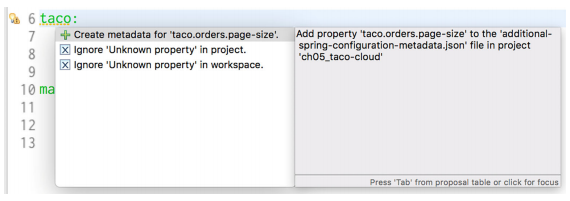
**Рисунок 5.3. Документация при наведению указателя на свойства конфигурации в Spring Tool Suite**

Чтобы помочь тем, кто может использовать свойства конфигурации, которые вы определяете - возможно, даже вы - обычно рекомендуется создать некоторые метаданные к этим свойствам. По крайней мере, это избавляет от этих надоедливых желтых предупреждений в IDE.

Чтобы создать метаданные для пользовательских свойств конфигурации, вам нужно создать файл в META-INF (например, в проекте в каталоге src/main/resources/META-INF) с именем Additional-spring-configuration-metadata.json.

БЫСТРОЕ ИСПРАВЛЕНИЕ ОТСУТСТВУЮЩИХ МЕТАДАННЫХ.

Если вы используете Spring Tool Suite, у вас есть возможность быстрого исправления для создания отсутствующих метаданных свойств. Поместите курсор на строку с предупреждением об отсутствующих метаданных и откройте всплывающее окно быстрого исправления с CMD-1 на Mac или Ctrl-1 в Windows и Linux (см. Рисунок 5.4).



**Рисунок 5.4 Создание метаданных свойства конфигурации с помощью всплывающего быстрого исправления в Spring Tool Suite**

Затем выберите параметр «Create Metadata for ...», чтобы добавить метаданные для свойства (в Additional-spring-configuration-metadata.json, как показано на этом рисунке), и создайте этот файл, если он еще не существует.

Для свойства taco.orders.pageSize вы можете настроить метаданные с помощью следующего JSON:

{

 "properties": [

   {

     "name": "taco.orders.page-size",

    "type": "java.lang.String",

    "description": "Sets the maximum number of orders to display in a list."

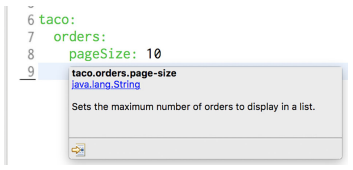
   }

 ]

}

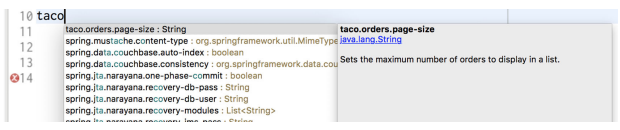
Обратите внимание, что имя свойства, указанное в метаданных, имеет вид taco.orders.page-size. Гибкое именование свойств в Spring Boot допускает некоторые изменения в именах свойств, так что taco.orders.page-size эквивалентен taco.orders.pageSize.

После задания метаданных предупреждения должны исчезнуть. Более того, если вы наведете указатель мыши на свойство taco.orders.pageSize, вы увидите описание, показанное на рисунке 5.5.



**Рисунок 5.5 Справка при наведении курсора на свойства пользовательской конфигурации**

Кроме того, вы получаете справку по автозаполнению из IDE, так же как по стандартным предоставляемым Spring-ом свойствам конфигурации (как показано на рисунке 5.6).



**Рисунок 5.6. Автодополнение при заполнении свойств на основе метаданных.**

Как вы уже видели, свойства конфигурации полезны для настройки как автоматически настраиваемых компонентов, так и деталей, внедряемых в компоненты вашего приложения. Но что, если вам нужно настроить разные свойства для разных сред развертывания? Давайте рассмотрим, как использовать профили Spring для настройки конкретной среды.

5.3 Конфигурация профилей

Когда приложения развертываются в разных средах выполнения, обычно некоторые детали конфигурации различаются. Например, подробности соединения с базой данных, вероятно, не одинаковы в среде разработки, в среде обеспечения качества,и в производственной среде (production). Одним из способов настройки уникальной свойств в каждой среде является использование переменных среды для указания свойств конфигурации вместо их определения в application.properties и application.yml.

Например, во время разработки вы можете опираться на автоматически сконфигурированную встроенную базу данных H2. Но в производственной среде вы можете установить свойства конфигурации базы данных как переменные среды:

% export SPRING\_DATASOURCE\_URL=jdbc:mysql://localhost/tacocloud

% export SPRING\_DATASOURCE\_USERNAME=tacouser

% export SPRING\_DATASOURCE\_PASSWORD=tacopassword

Хотя это сработает, но как-то, несколько громоздко указывать более одного или двух свойств конфигурации в качестве переменных среды. Более того, нет хорошего способа отследить изменения переменных среды или откатить изменения в случае ошибки.

Вместо этого я предпочитаю использовать профили Spring. Профили - это тип условной конфигурации, в которой различные компоненты, классы конфигурации и свойства конфигурации применяются или игнорируются в зависимости от того, какие профили активны во время выполнения.

Например, допустим, что для целей разработки и отладки вы хотите использовать встроенную базу данных H2 и хотите, чтобы уровни ведения журнала для кода Taco Cloud были установлены на DEBUG. Но в продакшине вы хотите использовать внешнюю базу данных MySQL и установить уровень логирования WARN. В ситуации разработки достаточно просто не задавать какие-либо свойства источника данных и получить автоматически сконфигурированную базу данных H2. А что касается уровня логирования в  отладке, вы можете установить для свойства logging.level.tacos для базового пакета tacos значение DEBUG в application.yml:

logging:

 level:

   tacos: DEBUG

Это именно то, что вам нужно для режима разработки. Но если бы вы развернули это приложение в режиме продакшен без каких-либо дальнейших изменений в application.yml, у вас все равно было бы логирование для пакета tacos и встроенная базы данных H2. Вам нужно определить профиль со свойствами, подходящими для продакшена.

5.3.1 Определение свойств профиля

Один из способов определения специфичных для профиля свойств - создать еще один YAML или файл свойств, содержащий только свойства для продакшена. Имя файла должно соответствовать следующему соглашению: application-{имя профиля}.yml или application-{имя профиля}.properties. Затем вы можете указать свойства конфигурации, соответствующие этому профилю. Например, вы можете создать новый файл с именем application-prod.yml, который содержит следующие свойства:

spring:

 datasource: url: jdbc:mysql://localhost/tacocloud

 username: tacouser

 password: tacopassword

 logging:

   level:

     tacos: WARN

Другой способ указать специфичные для профиля свойства работает только с конфигурацией YAML. Он включает размещение специфичных для профиля свойств вместе с непрофилированными свойствами в application.yml, разделенных тремя дефисами и свойством spring.profiles для присвоения имени профилю. При записи свойств для продакшена в application.yml файл будет выглядеть так:

logging:

 level:

   tacos: DEBUG

---

spring:

 profiles: prod

   datasource: url: jdbc:mysql://localhost/tacocloud

   username: tacouser

   password: tacopassword

logging:

 level:

   tacos: WARN

Как вы можете видеть, этот файл application.yml разделен на две секции набором тройных дефисов (---). Во втором разделе указывается значение для spring.profiles, указывающее, что следующие свойства применяются к профилю prod. В первом разделе не указывается никакое значение для spring.profiles. Следовательно, его свойства являются общими для всех профилей или являются значениями по умолчанию, если активный профиль не переопределяет значение этих свойств.

Независимо от того, какие профили активны при запуске приложения, уровень логирвоания для пакета tacos будет установлен на DEBUG с помощью свойства, установленного в профиле по умолчанию. Но если активен профиль с именем prod, то свойство logging.level.tacos будет переопределено как WARN. Аналогично, если профиль prod активен, то свойства источника данных будут установлены для использования внешней базы данных MySQL.

Вы можете определить свойства для любого количества профилей, создав дополнительные файлы YAML или файлы свойств, названные по шаблону application-{имя профиля} .yml или application-{имя профиля} .properties. Или, если хотите, введите еще три черты в application.yml вместе с другим свойством spring.profiles, чтобы указать имя профиля. Затем добавьте все необходимые для профиля свойства.

5.3.2 Активация профилей

Установка специфичных для профиля свойств не принесет пользы, если эти профили не активны. Но как сделать профиль активным? Чтобы сделать профиль активным, достаточно лишь включить его в список имен профилей, заданный свойству spring.profiles.active. Например, вы можете установить его в application.yml следующим образом:

spring:

 profiles:

   active:

     - prod

Но это, пожалуй, худший из возможных способов установить активный профиль. Если вы установите активный профиль в application.yml, то этот профиль станет профилем по умолчанию, и вы не добьетесь ни одного из преимуществ использования профилей для отделения специфичных для продакшена свойств от свойств разработки. Вместо этого я рекомендую вам установить активные профили с переменными окружения. В продакшен среде вы должны установить SPRING\_PROFILES\_ACTIVE следующим образом:

% export SPRING\_PROFILES\_ACTIVE=prod

С этого момента любые приложения, развернутые на этом компьютере, будут иметь активный профиль prod, и соответствующие свойства конфигурации будут иметь приоритет над свойствами в профиле по умолчанию.

Если вы запускаете приложение как исполняемый файл JAR, вы также можете установить активный профиль через аргумент командной строки, например:

% java -jar taco-cloud.jar --spring.profiles.active=prod

Обратите внимание, что имя свойства spring.profiles.active содержит слово множественного числа - profiles. Это означает, что вы можете указать более одного активного профиля. Зададим несколько активных профелей с разделением через запятую, также как можно установить и через переменную окружения:

% export SPRING\_PROFILES\_ACTIVE=prod,audit,ha

В YAML вы бы активные профили в виде списка:

spring:

 profiles:

   active:

     - prod

    - audit

    - ha

Стоит также отметить, что если вы развертываете приложение Spring в Cloud Foundry, для вас автоматически активируется профиль с именем cloud. Если Cloud Foundry является вашей производственной средой, вы обязательно должны указать специфичные для продакшена свойства в профиле cloud.

Открою тайну - профили были бы достаточно бесполезны, если бы могли использоваться только для условной установки свойств конфигурации в приложении Spring. Давайте посмотрим, как объявить bean-компоненты, специфичные для активного профиля.

5.3.3 Создание bean-ов в зависимости от профиля

Иногда полезно предоставить уникальный набор bean-компонентов для разных профилей. Обычно любой компонент, объявленный в классе конфигурации Java, создается независимо от того, какой профиль активен. Но предположим, что есть некоторые bean-компоненты, которые вам нужно создать, только если определенный профиль активен. В этом случае аннотация @Profile может определять bean-компоненты как применимые только к данному профилю.

Например, у вас есть компонент CommandLineRunner, объявленный в TacoCloudApplication, который используется для загрузки встроенной базы данных с данными об ингредиентах при запуске приложения. Это здорово для режима разработки, но было бы ненужным (и нежелательным) в приложении на продакшене. Чтобы предотвратить загрузку данных ингредиента при каждом запуске приложения при продакшен развертывании, вы можете аннотировать метод компонента CommandLineRunner с помощью @Profile следующим образом:

@Bean

@Profile("dev")

public CommandLineRunner dataLoader(IngredientRepository repo,

   UserRepository userRepo, PasswordEncoder encoder) {

 ...

}

Или предположим, что вам нужно создать CommandLineRunner, если активен либо dev-профиль, либо qa-профиль. В этом случае вы можете перечислить профили, для которых должен быть создан компонент:

@Bean

@Profile({"dev", "qa"})

public CommandLineRunner dataLoader(IngredientRepository repo,

   UserRepository userRepo, PasswordEncoder encoder) {

 ...

}

Теперь данные ингредиента будут загружены только если активны профили dev или qa. Это означало бы, что вам нужно активировать профиль разработчика при запуске приложения в среде разработки. Было бы еще удобнее, если бы этот компонент CommandLineRunner создавался всегда, если профиль prod не активен. В этом случае вы можете применить @Profile следующим образом:

@Bean

@Profile("!prod")

public CommandLineRunner dataLoader(IngredientRepository repo,

   UserRepository userRepo, PasswordEncoder encoder) {

 ...

}

Здесь восклицательный знак (!) отменяет имя профиля. По сути, он утверждает, что bean-компонент CommandLineRunner будет создан, если профиль prod не активен.

Также можно использовать @Profile для всего класса, аннотированного @Configuration. Например, предположим, что вы должны были извлечь компонент CommandLineRunner в отдельном классе конфигурации с именем DevelopmentConfig.Для этого достаточно аннотировать DevelopmentConfig с помощью @Profile:

@Profile({"!prod", "!qa"})

@Configuration

public class DevelopmentConfig {

 @Bean

 public CommandLineRunner dataLoader(IngredientRepository repo,

   UserRepository userRepo, PasswordEncoder encoder) {

 ...

 }

}

Здесь bean-компонент CommandLineRunner (как и любые другие bean-компоненты, определенные в DevelopmentConfig) будет создан, только если ни prod, ни qa-профили не активны.

Итог:

* Spring bean-компоненты могут быть аннотированы с помощью @ConfigurationProperties, чтобы включить внедрение значений из одного из нескольких источников свойств.
* Свойства конфигурации можно задавать в аргументах командной строки, переменных среды, системных свойствах JVM, файлах свойств или файлах YAML, а также в других параметрах.
* Свойства конфигурации можно использовать для переопределения параметров автоконфигурации, включая возможность указать URL-адрес источника данных и уровни логирования.
* Профили Spring можно использовать с источниками свойств для условной установки свойств конфигурации на основе активных профилей.

[Spring in Action Covers Spring 5.0 перевод на русский. Глава 6](https://beginner-mercenary.blogspot.com/2019/05/spring-in-action-covers-spring-50-6.html)

6. Создание REST сервисов

**Эта глава охватывает**

* **Определение REST endpoints в  Spring MVC**
* **Включение гиперссылочных REST ресурсов**
* **Автоматические REST endpoints на основе репозитория**

«Веб-браузер мертв. Что теперь?"

Примерно дюжину лет назад я слышал, как кто-то предположил, что веб-браузер приближается к статусу legacy и что что-то другое возьмет верх. Но как такое могло случиться? Что может свергнуть почти вездесущий веб-браузер? Как бы мы использовали растущее количество сайтов и онлайн-сервисов, если бы не веб-браузер? Конечно, это был бред сумасшедшего!

Вернемся в день сегодняшний и станет очевидно, что веб-браузер не исчез. Но он больше не является основным средством доступа к Интернету. Мобильные устройства, планшеты, умные часы и голосовые устройства стали обычным явлением. И даже многие браузерные приложения на самом деле работают с приложениями JavaScript, а не позволяют браузеру быть тупым терминалом для рендеринга контента на сервере.

С таким широким выбором вариантов на стороне клиента многие приложения приняли общий дизайн, в котором пользовательский интерфейс перемещается ближе к клиенту, а сервер предоставляет API, через который все типы клиентов могут взаимодействовать с функциональностью бэкэнда.

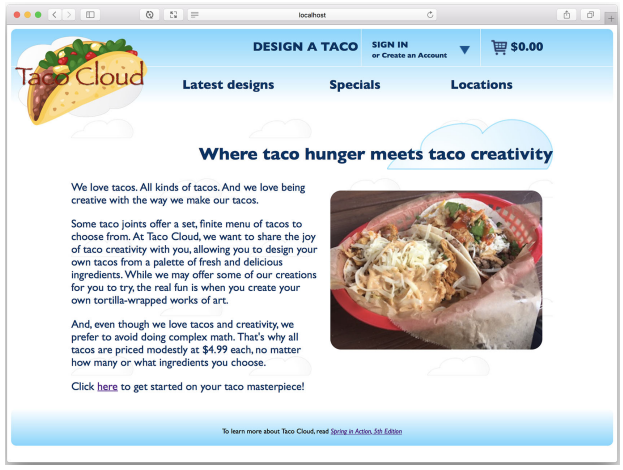
В этой главе вы собираетесь использовать Spring для предоставления REST API для приложения TacoCloud. Вы будете использовать то, что узнали о Spring MVC в главе 2, для создания endpoints  RESTful с контроллерами Spring MVC. Вы также автоматически выставите REST endpoints для Spring Data repository-ев, которые вы определили в главе 4. Наконец, мы рассмотрим способы тестирования и защиты этих endpoint.

Но сначала вы напишите  несколько новых контроллеров Spring MVC, которые предоставляют функциональные возможности бэкэнда с endpoint REST, которые будут использоваться богатым веб-интерфейсом.

6.1 Написание RESTful контроллеров

Надеюсь, вы не возражаете, но пока вы переворачивали страницу и читали введение к этой главе, я взял на себя смелость переосмыслить пользовательский интерфейс для TacoCloud. То, с чем вы работали, было прекрасно для начала, но этого не хватало по части эстетики.

Рисунок 6.1 - это просто пример того, как выглядит Taco Cloud. Довольно шикарно, а?



**Рисунок 6.1 Новая домашняя страница Taco Cloud**

Когда я любовался внешним видом Taco Cloud, я решил создать веб-интерфейс в виде одностраничного приложения с использованием популярной платформы Angular. В конечном счете, этот новый пользовательский интерфейс браузера заменит серверные страницы, созданные вами в главе 2. Но для этого вам потребуется создать REST API, с которым пользовательский интерфейс построенный на Angular (Я решил использовать Angular, но выбор среды интерфейса не должен иметь никакого отношения к написанию бэккенд кода Spring. Не стесняйтесь выбирать Angular, React, Vue.js или любую другую технологию веб-интерфейса, которая подходит вам больше всего) будет связываться, чтобы сохранять и извлекать данные тако.

SPA или не  SPA?

Вы разработали традиционное многостраничное приложение (MPA) с Spring MVC в главе 2, и теперь вы заменяете его одностраничным приложением (SPA) на основе Angular. Но я не утверждаю, что SPA всегда лучший выбор, чем MPA.

Поскольку представление в значительной степени отделено от серверной обработки в SPA, это дает возможность разработать более одного пользовательского интерфейса (такого как собственное мобильное приложение) для одной и той же функциональности сервера. Это также открывает возможность для интеграции с другими приложениями, которые могут использовать API. Но не все приложения требуют такой гибкости, и MPA - это более простой дизайн, если все, что вам нужно, это отображать информацию на веб-странице.

Это не книга по Angular, поэтому код в этой главе будет сосредоточен в основном на бэккенд Spring-коде. Я покажу достаточно Angular-кода, чтобы вы могли понять, как работает клиентская часть. Будьте уверены, что полный набор кода, включая Angular frontend, доступен как часть загружаемого кода для книги и по адресу https://github.com/habuma/spring-in-action-5-samples. Вас также может заинтересовать чтение Angular in Action Джереми Уилкена (Manning, 2018) и Angular Development with TypeScript, второе издание Якова Файна и Антона Моисеева (Manning, 2018).

В двух словах, клиентский код Angular будет взаимодействовать с API, который вы создадите в этой главе посредством HTTP-запросов. В главе 2 вы использовали аннотации @GetMapping и @PostMapping для извлечения и публикации данных на сервере. Те же самые аннотации по-прежнему пригодятся, когда вы определите свой REST API. Кроме того, Spring MVC поддерживает несколько других аннотаций для различных типов HTTP-запросов, как указано в таблице 6.1.

Таблица 6.1. Spring MVC HTTP аннотации обработки запросов (Сопоставление методов HTTP для создания, чтения, обновления и удаления (CRUD) операций не является идеальным соответствием, но на практике именно так они часто используются и как вы будете их использовать в Taco Cloud.)

Аннотация  - HTTP метод  - Стандартное применение

@GetMapping  - HTTP GET requests  - Чтение данных

@PostMapping - HTTP POST requests -  Создание данных

@PutMapping - HTTP PUT requests - Изменение данных

@PatchMapping -  HTTP PATCH requests - Изменение данных

@DeleteMapping - HTTP DELETE requests - Удаление данных

@RequestMapping - Обработка запросов общего назначения; HTTP - метод, указанный как атрибут метода

Чтобы увидеть эти аннотации в действии, вы начнете с создания простой REST endpoint, которая выбирает несколько самых последних созданных тако.

6.1.1 Получение данных с сервера

Одна из самых крутых вещей в Taco Cloud - это то, что он позволяет фанатикам тако создавать свои собственные творения тако и делиться ими со своими коллегами-любителями тако. Для этого в Taco Cloud должна быть возможность отображать список самых последних созданных тако при нажатии на ссылку «Latest Designs».

В коде Angular я определил компонент RecentTacosComponent, который будет отображать самые последние созданные тако. Полный код TypeScript для RecentTacosComponent показан в следующем листинге.

**Листинг 6.1. Angular компонент для отображения последних тако**

import { Component, OnInit, Injectable } from '@angular/core';

import { Http } from '@angular/http';

import { HttpClient } from '@angular/common/http';

@Component({

 selector: 'recent-tacos',

 templateUrl: 'recents.component.html',

 styleUrls: ['./recents.component.css']

})

 @Injectable()

 export class RecentTacosComponent implements OnInit {

   recentTacos: any;

   constructor(private httpClient: HttpClient) { }

   ngOnInit() {

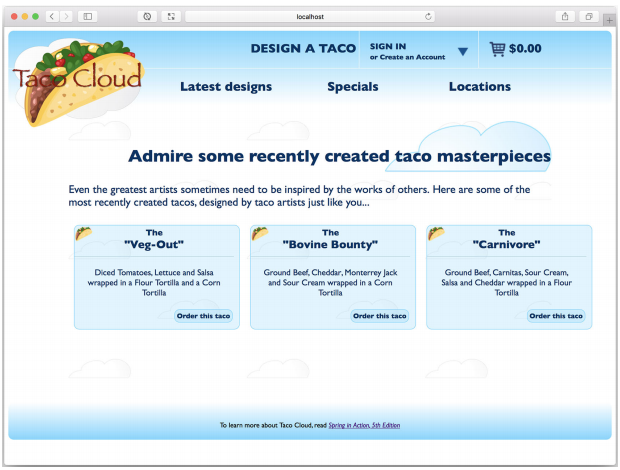
     this.httpClient.get('<http://localhost:8080/design/recent>') //Получает последние тако с сервера

       .subscribe(data => this.recentTacos = data);

 }

}

Обратите ваше внимание на метод ngOnInit(). В этом методе RecentTacosComponent использует внедренный модуль Http для выполнения HTTP-запроса GET к http://localhost:8080/design/latest, ожидая, что ответ будет содержать список дизайнов тако, который будет помещен в переменную модели recentTacos. Визуализация ( recents.component.html) представит данные этой модели в виде HTML, которые будут отображаться в браузере. Конечный результат может выглядеть примерно так, как показано на рисунке 6.2, после создания трех тако.



**Рисунок 6.2 Отображение последних созданных тако**

Недостающий фрагмент этой головоломки - это конечная точка (endpoint), которая обрабатывает запросы GET для /design/recent и отдает списком недавно созданных тако. Вы создадите новый контроллер для обработки такого запроса. Следующий листинг показывает этот контроллер.

**Листинг 6.2. Контроллер RESTful для API запросов дизайнов тако**

package tacos.web.api;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.domain.PageRequest;

import org.springframework.data.domain.Sort;

import org.springframework.hateoas.EntityLinks;

import org.springframework.http.HttpStatus;

import org.springframework.web.bind.annotation.CrossOrigin;

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

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.ResponseStatus;

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

import tacos.Taco;

import tacos.data.TacoRepository;

@RestController

@RequestMapping(path="/design", produces="application/json") //Обрабатывает запросы на /design

@CrossOrigin(origins="\*") //Позволяет перекрестные запросы

public class DesignTacoController {

 private TacoRepository tacoRepo;

@Autowired

 EntityLinks entityLinks;

 public DesignTacoController(TacoRepository tacoRepo) {

   this.tacoRepo = tacoRepo;

 }

 @GetMapping("/recent")

 public Iterable<Taco> recentTacos() { //Формирует и отдает последние дизайны тако

   PageRequest page = PageRequest.of(

     0, 12, Sort.by("createdAt").descending());

   return tacoRepo.findAll(page).getContent();

 }

}

Аннотация @RestController служит двум целям. Во-первых, это аннотация стереотипа, такая как @Controller и @Service, которая отмечает класс для обнаружения при сканировании компонентов. Но наиболее релевантным для обсуждения REST, аннотация @RestController сообщает Spring, что все методы-обработчики в контроллере должны иметь свое возвращаемое значение, записанное непосредственно в тело ответа, а не записываться в модель в представление для визуализации.

В качестве альтернативы, вы могли бы аннотировать DesignTacoController с помощью @Controller, как и с любым контроллером Spring MVC. Но тогда вам также нужно аннотировать все методы-обработчики с помощью @ResponseBody для достижения того же результата. Еще один вариант - вернуть объект ResponseEntity, о котором мы поговорим чуть позже.

Аннотация @RequestMapping на уровне класса работает с аннотацией @GetMapping в методе recentTacos(), чтобы указать, что метод recentTacos() отвечает за обработку запросов GET для /design/recent (именно это необходимо для вашего кода Angular).

Вы заметите, что аннотация @RequestMapping также устанавливает атрибут produces. Это указывает, что любой из методов-обработчиков в DesignTacoController будет обрабатывать запросы только в том случае, если заголовок Accept запроса включает в себя «application/json». Это не только ограничивает ваш API только выдачей результатов в виде JSON, но также позволяет другому контроллеру (возможно, DesignTacoController из главы 2) обрабатывать запросы с одинаковыми путями, если эти запросы не имеют требований к вывода в формате JSON. Несмотря на то, что это ограничивает ваш API-интерфейс только JSON (что подходит для ваших нужд), вы можете установить produces как массив String нескольких типов контента. Например, чтобы разрешить вывод XML, вы можете добавить” text/html " в атрибут produces:

@RequestMapping(path="/design", produces={"application/json", "text/xml"})

То что еще вы, возможно, заметили в листинге 6.2, это то, что класс аннотирован @CrossOrigin. Поскольку Angular-часть приложения будет работать на отдельном хосте и/или порту по API (по крайней мере, на данный момент), веб-браузер не позволит вашему Angular-клиенту использовать API. Это ограничение можно обойти, включив заголовки CORS (Cross-Origin Resource Sharing) в ответы сервера. Spring упрощает применение CORS с аннотацией @CrossOrigin. В данном случае @CrossOrigin позволяет клиентам из любого домена использовать API.

Логика в методе recentTacos() довольно проста. Он создает объект PageRequest, который указывает, что вы хотите получить первую (0-я) страницу с первыми 12 результатами сортировки в порядке убывания по дате создания тако. Короче говоря, вы хотите дюжину самых последних созданных дизайнов тако. PageRequest передается в вызов метода findAll() объекта TacoRepository,  и содержимое этой страницы результатов возвращается клиенту (ответ, как вы видели в листинге 6.1, будет использоваться в качестве данных модели для отображения пользователю).

Теперь предположим, что вы хотите создать endpoint, который извлекает один тако по его идентификатору. Используя переменную-заполнитель в пути метода обработчика и принимая переменную path, вы можете получить ID и использовать его для поиска объекта Taco используя репозиторий:

@GetMapping("/{id}")

public Taco tacoById(@PathVariable("id") Long id) {

 Optional<Taco> optTaco = tacoRepo.findById(id);

   if (optTaco.isPresent()) {

     return optTaco.get();

 }

 return null;

}

Поскольку базовый (корневой) путь контроллера - /design, этот метод контроллера обрабатывает запросы GET для /design/{id}, где часть пути {id} является заполнителем. Фактическое значение в запросе задается параметру id, который сопоставляется с заполнителем {id} с помощью @PathVariable.

Внутри tacoById () параметр id передается методу findById() в репозиторий для получения Taco. findById() возвращает Optional<Taco>, потому что возможна ситуация, что нет тако с переданным ID. Поэтому перед тем как получить результат необходимо проверить, существует ли тако по переданному идентификатору. Если такой тако существует, вы вызываете метод get() для Optional<Taco>, чтобы получить Taco.

Если идентификатор не соответствует ни одному известному tacos, вы возвращаете null. Однако это далеко не идеально. Возвращая значение null, клиент получает ответ с пустым телом и кодом состояния HTTP 200 (OK). Клиент получает ответ, который он не может использовать, но код состояния указывает, что все в порядке. Лучшим подходом было бы вернуть ответ со статусом HTTP 404 (не найден).

Текущая реализация не имеет простого способа, чтобы возвращать код статуса 404 от tacoById(). Но если произвести небольшие изменения, это станет возможным:

@GetMapping("/{id}")

public ResponseEntity<Taco> tacoById(@PathVariable("id") Long id) {

 Optional<Taco> optTaco = tacoRepo.findById(id);

 if (optTaco.isPresent()) {

   return new ResponseEntity<>(optTaco.get(), HttpStatus.OK);

 }

 return new ResponseEntity<>(null, HttpStatus.NOT\_FOUND);

}

Теперь вместо возврата объекта Taco функция tacoById () возвращает ResponseEntity <Taco>. Если тако найдено, вы оборачиваете объект Taco в ResponseEntity с HTTP-статусом OK (что и было ранее). Но если тако не найдено, вы добавляете в ResponseEntity значение null вместе с HTTP-статусом NOT FOUND, чтобы указать, что клиент пытается получить тако, которого не существует.

Начало API Taco Cloud для вашего  Angular-овского (или любого другого, по вашим предпочтениям) клиента положено. Для тестирования разработки вы также можете использовать утилиты командной строки, такие как curl или HTTPie (https://httpie.org/) чтобы лучше понимать API. Например, следующая командная строка показывает, как вы могли бы получить недавно созданные тако с curl:

$ curl localhost:8080/design/recent

Или такой командой, если вы предпочитаете HTTPie:

$ http :8080/design/recent

Но реализация endpoint, которая возвращает информацию, - это только начало. Что делать, если ваш API должен получать данные от клиента? Давайте посмотрим, как вы можете написать методы контроллера, которые обрабатывают входные данные запросов.

6.1.2 Отправка данных на сервер

Сейчас приложение может вернуть дюжину самых последних созданных тако. Но как эти тако были созданы изначально?

Вы еще не удалили ни один код из главы 2, поэтому у вас все еще есть оригинальный DesignTacoController, который отображает форму дизайна Taco и обрабатывает отправку формы. Это отличный способ получить некоторые тестовые данные для тестирования созданного вами API. Но если вы собираетесь преобразовать Taco Cloud в одностраничное приложение, вам нужно будет создать компоненты Angular и соответствующие endpoint-ы, чтобы заменить форму дизайна Taco из главы 2.

Я уже переработал клиентский код для формы дизайна taco, создав новый Angular компонент с именем DesignComponent (в файле с именем design.component.ts). Поскольку в нем должен присутствовать функционал отправки данных формы, DesignComponent имеет метод onSubmit (), который выглядит следующим образом:

onSubmit() {

 this.httpClient.post(

   'http://localhost:8080/design',

   this.model, {

     headers: new HttpHeaders().set('Content-type', 'application/json'),

   }).subscribe(taco => this.cart.addToCart(taco));

 this.router.navigate(['/cart']);

}

В методе onSubmit() вместо get() вызывается метод post(). Это означает, что вместо извлечения данных из API, вы отправляете данные в API. В частности, вы отправляете дизайн тако, который хранится в переменной model, в API endpoint /design с request-ом HTTP POST.

Это означает, что вам нужно будет написать метод в DesignTacoController для обработки этого запроса и сохранения дизайна. Добавив следующий метод postTaco() в DesignTacoController, вы реализуете функционал в контроллере для этой цели:

@PostMapping(consumes="application/json")

@ResponseStatus(HttpStatus.CREATED)

public Taco postTaco(@RequestBody Taco taco) {

 return tacoRepo.save(taco);

}

Поскольку postTaco() будет обрабатывать HTTP POST request , он аннотируется @PostMapping вместо @GetMapping. Вы не указываете здесь атрибут path, поэтому метод postTaco() будет обрабатывать запросы для /design, как указано в классе @RequestMapping у DesignTacoController.

Вы устанавливаете атрибут consumes. Здесь вы используете consumes, чтобы сказать, что метод будет обрабатывать только запросы, Content-type которых соответствует application/json.

Параметр Taco для метода помечается @RequestBody, чтобы указать, что тело запроса должно быть преобразовано в объект Taco и привязано к параметру. Эта аннотация важна, без нее Spring MVC предполагает, что вы хотите, чтобы параметры запроса (либо параметры запроса, либо параметры формы) были связаны с объектом Taco. Но аннотация @RequestBody гарантирует, что вместо этого JSON в теле запроса будет связан с объектом Taco.

Как только postTaco() получает объект Taco, он передает его методу save() в TacoRepository.

Возможно, вы также заметили, что я аннотировал метод postTaco() с помощью @ResponseStatus (HttpStatus.CREATED). При нормальных обстоятельствах (когда не генерируются исключения) все ответы будут иметь код состояния HTTP 200 (ОК), что указывает на успешность запроса. Хотя ответ HTTP 200 всегда приветствуется, он не всегда достаточно описательный. В случае запроса POST HTTP-статус 201 (CREATED) является более информативным. Он сообщает клиенту, что запрос был не только успешным, но в результате был создан ресурс. Всегда целесообразно использовать @ResponseStatus, где это уместно, для передачи клиенту наиболее описательного и точного кода состояния HTTP.

Хотя вы использовали @PostMapping для создания нового ресурса Taco, POST-запросы также можно использовать для обновления ресурсов. Тем не менее, запросы POST обычно используются для создания ресурсов, а запросы PUT и PATCH используются для обновления ресурсов. Давайте посмотрим, как вы можете обновить данные, используя @PutMapping и @PatchMapping.

6.1.3 Обновление данных на сервере

Прежде чем написать какой-либо код контроллера для обработки команд HTTP PUT или PATCH, вам следует уделить время рассмотрению слона в комнате: почему существуют два разных метода HTTP для обновления ресурсов?

Хотя это правда, что PUT часто используется для обновления данных ресурсов, на самом деле это семантическая противоположность GET. В то время как запросы GET предназначены для передачи данных с сервера на клиент, запросы PUT предназначены для отправки данных с клиента на сервер.

В этом смысле PUT действительно предназначен для выполнения операции оптовой замены, а не операции обновления. Напротив, целью HTTP PATCH является выполнение исправления или частичное обновление данных ресурса.

Например, предположим, что вы хотите изменить адрес заказа. Один из способов добиться этого с помощью REST API - обработать запрос PUT следующим образом:

@PutMapping("/{orderId}")

public Order putOrder(@RequestBody Order order) {

 return repo.save(order);

}

Это может сработать, но для этого потребуется, чтобы клиент передал полные данные заказа в запросе PUT. Семантически PUT означает «передать эти данные по этому URL», по сути, заменяя любые данные, которые уже есть. Если какое-либо из свойств заказа будет опущено, значение этого свойства будет заменено на ноль. Даже тако в заказе необходимо будет передать вместе с данными заказа, иначе они будут удалены из заказа.

Если PUT выполняет оптовую замену данных объекта, то как вам следует обрабатывать запросы, чтобы выполнить только частичное обновление? Вот для чего хороши запросы HTTP PATCH и Spring @PatchMapping. Вот как вы можете написать метод контроллера для обработки запроса PATCH для заказа:

@PatchMapping(path="/{orderId}", consumes="application/json")

public Order patchOrder(@PathVariable("orderId") Long orderId,

    @RequestBody Order patch) {

 Order order = repo.findById(orderId).get();

 if (patch.getDeliveryName() != null) {

   order.setDeliveryName(patch.getDeliveryName());

 }

 if (patch.getDeliveryStreet() != null) {

   order.setDeliveryStreet(patch.getDeliveryStreet());

 }

 if (patch.getDeliveryCity() != null) {

   order.setDeliveryCity(patch.getDeliveryCity());

 }

 if (patch.getDeliveryState() != null) {

   order.setDeliveryState(patch.getDeliveryState());

 }

 if (patch.getDeliveryZip() != null) {

   order.setDeliveryZip(patch.getDeliveryState());

 }

 if (patch.getCcNumber() != null) {

   order.setCcNumber(patch.getCcNumber());

 }

 if (patch.getCcExpiration() != null) {

   order.setCcExpiration(patch.getCcExpiration());

 }

 if (patch.getCcCVV() != null) {

   order.setCcCVV(patch.getCcCVV());

 }

 return repo.save(order);

}

Первое, на что следует обратить внимание, это то, что метод patchOrder() имеет аннотацию @PatchMapping вместо @PutMapping, что указывает на то, что он должен обрабатывать запросы HTTP PATCH вместо запросов PUT.

Но одна вещь, которую вы, несомненно, заметили, это то, что метод patchOrder() немного сложнее, чем метод putOrder().  Это потому, что аннотации сопоставления Spring MVC, включая @Pathmapping и @PutMapping, указывают только, какие типы запросов должен обрабатывать метод. Эти аннотации не определяют, как будет обрабатываться запрос. Даже если PATCH семантически подразумевает частичное обновление, вы должны написать код в методе-обработчике, который фактически выполняет такое обновление.

В случае метода putOrder() вы приняли полные данные для заказа и сохранили их, придерживаясь семантики HTTP PUT. Но для того, чтобы patchMapping() придерживался семантики HTTP PATCH, тело метода требует большего интеллекта. Вместо полной замены заказа новыми отправленными данными, он проверяет каждое поле входящего объекта заказа и применяет любые не-null значения к существующему заказу. Этот подход позволяет клиенту отправлять только те свойства, которые должны быть изменены, и позволяет серверу сохранять существующие данные для любых свойств, не указанных клиентом.

Существует более одного подхода к реализации PATCH

Подход исправления, применяемый в методе patchOrder(), имеет несколько ограничений:

* Если null значения предназначены для указания отсутствия изменений, как клиент может указать, что поле должно быть установлено в null?
* Невозможно удалить или добавить подмножество элементов из коллекции. Если клиент хочет добавить или удалить запись из коллекции, он должен отправить полную измененную коллекцию.

На самом деле не существует строгого правила о том, как обрабатывать запросы PATCH или как должны выглядеть входящие данные. Вместо того, чтобы отправлять фактические данные домена, клиент может отправить специфическое для патча описание изменений, которые будут применены. Конечно, обработчик запроса должен был бы быть написан для обработки инструкций патча вместо данных домена.

Обратите внимание, что и в @PutMapping, и в @PatchMapping путь запроса ссылается на ресурс, который необходимо изменить. Это те же самые пути что обрабатываются с помощью методов аннотированных @GetMapping.

Теперь вы видели, как получать и обновлять ресурсы с помощью @GetMapping и @PostMapping. И вы видели два разных способа обновления ресурса с помощью @PutMapping и @PatchMapping. Осталось только обработать запросы на удаление ресурса.

6.1.4 Удаление данных с сервера

Иногда данные просто больше не нужны. В этих случаях клиент должен иметь возможность запросить удаление ресурса с помощью HTTP DELETE request.

Spring MVC @DeleteMapping пригодится для объявления методов, которые обрабатывают запросы на удаление. Например, предположим, вы хотите, чтобы ваш API разрешал удаление ресурса заказа. Следующий метод контроллера должен реализовать этот функционал:

@DeleteMapping("/{orderId}")

@ResponseStatus(code=HttpStatus.NO\_CONTENT)

public void deleteOrder(@PathVariable("orderId") Long orderId) {

 try {

    repo.deleteById(orderId);

 } catch (EmptyResultDataAccessException e) {}

}

К этому моменту принцип построения аннотаций должна быть вам знакома. Вы уже видели @GetMapping, @PostMapping, @PutMapping и @ PatchMapping, каждый из которых указывает, что метод должен обрабатывать HTTP запросы соответствующие им методов. Возможно, вас не удивит, что @DeleteMapping используется для указания того, что метод deleteOrder() отвечает за обработку запросов DELETE для /orders/{orderId}.

Код в методе - это то, что фактически выполняет удаление заказа. В этом случае он принимает идентификатор заказа, предоставленный в качестве переменной пути в URL-адресе, и передает его в метод deleteById () репозитория. Если заказ существует при вызове этого метода, он будет удален. Если заказ не существует, будет выброшено исключение EmptyResultDataAccessException.

Я решил отлавливать EmptyResultDataAccessException и ничего с ними не делать. Я думаю, что если вы попытаетесь удалить ресурс, который не существует, результат будет таким же, как если бы он существовал до удаления. То есть ресурс перестанет существовать. Существовал ли он раньше или нет не имеет значения. Кроме того, я мог бы написать deleteOrder(), чтобы вернуть ResponseEntity, установив тело в null и код состояния HTTP NOT FOUND.

Единственное, на что следует обратить внимание в методе deleteOrder(), это то, что он аннотирован @ResponseStatus, чтобы гарантировать, что HTTP-статус ответа равен 204 (NO CONTENT). Нет необходимости передавать какие-либо данные ресурса обратно клиенту для ресурса, который больше не существует, поэтому ответы на запросы DELETE обычно не имеют тела и, следовательно, должны сообщать код состояния HTTP, чтобы клиент знал, что не стоит ожидать никакого содержимого.

Ваш Taco Cloud API начинает обретать форму. Клиентский код теперь может легко использовать этот API для отображения ингредиентов, принятия заказов и отображения недавно созданных тако. Но есть кое-что, что вы можете сделать, что сделает ваш API еще проще для клиента. Давайте посмотрим, как вы можете добавить hypermedia в Taco Cloud API.

6.2 Добавление hypermedia

API, который вы создали до сих пор, довольно прост, но он работает, пока клиент, который его использует, знает схему URL API. Например, клиент может быть жестко закодирован, чтобы знать, что он может получить список недавно созданных тако, выполнив запрос GET для /design/recent. Аналогично, он должен понимать, что он может добавить идентификатор любого taco из полученного перечня в /design, чтобы получить URL для этого конкретного taco.

Использование жестко закодированных шаблонов URL и манипулирование строками распространено среди клиентского кода API. Но представьте на мгновение, что произойдет, если изменится схема URL API. Жестко закодированный клиентский код будет иметь устаревшее понимание API и, таким образом, будет не корректен. Жесткое кодирование URL-адресов API и использование строковых манипуляций с ними делает клиентский код хрупким.

Гипермедиа как движок состояния приложения, или HATEOAS, является средством создания API с функционалом самоописания, в которых ресурсы, возвращаемые из API, содержат ссылки на связанные ресурсы. Это позволяет клиентам перемещаться по API с минимальным пониманием URL-адресов API. Вместо этого он понимает взаимосвязи между ресурсами, обслуживаемыми API, и использует свое понимание этих взаимосвязей для обнаружения URL-адресов API по мере их прохождения.

Например, предположим, что клиент должен был запросить список недавно разработанных тако. В необработанном виде, без гиперссылок, список последних тако будет получен в клиенте с JSON, который выглядит следующим образом (для краткости все, кроме первого тако в списке, вырезаны):

[

 {

   "id": 4,

   "name": "Veg-Out",

   "createdAt": "2018-01-31T20:15:53.219+0000",

   "ingredients": [

       {"id": "FLTO", "name": "Flour Tortilla", "type": "WRAP"},

       {"id": "COTO", "name": "Corn Tortilla", "type": "WRAP"},

       {"id": "TMTO", "name": "Diced Tomatoes", "type": "VEGGIES"},

       {"id": "LETC", "name": "Lettuce", "type": "VEGGIES"},

       {"id": "SLSA", "name": "Salsa", "type": "SAUCE"}

     ]

   },

   ...

]

Если клиент желает получить или выполнить какую-либо другую HTTP-операцию над конкретным тако, ему необходимо знать (с помощью жесткого кодирования), что он может добавить значение свойства id к URL-адресу, путь которого - /design. Аналогично, если бы он хотел выполнить операцию HTTP над одним из ингредиентов, ему нужно было бы знать, что он может добавить значение свойства id ингредиента к URL-адресу, путь которого равен /ingredients. В любом случае ему также необходимо добавить префикс к этому пути с http: // или https: // и имя хоста API.

Напротив, если API включен с гипермедиа, API будет описывать свои собственные URL-адреса, освобождая клиента от необходимости жестко кодировать эти знания. Тот же список недавно созданных тако мог бы выглядеть как следующий список, если бы были вставлены гиперссылки.

**Листинг 6.3. Список тако-ресурсов с гиперссылками**

{

 "\_embedded": {

   "tacoResourceList": [

     {

      "name": "Veg-Out",

       "createdAt": "2018-01-31T20:15:53.219+0000",

       "ingredients": [

       {

         "name": "Flour Tortilla", "type": "WRAP",

         "\_links": {

         "self": { "href": "http://localhost:8080/ingredients/FLTO" }

       }

     },

     {

       "name": "Corn Tortilla", "type": "WRAP",

       "\_links": {

         "self": { "href": "http://localhost:8080/ingredients/COTO" }

       }

     },

     {

       "name": "Diced Tomatoes", "type": "VEGGIES",

       "\_links": {

         "self": { "href": "http://localhost:8080/ingredients/TMTO" }

       }

     },

     {

       "name": "Lettuce", "type": "VEGGIES",

       "\_links": {

         "self": { "href": "http://localhost:8080/ingredients/LETC" }

       }

     },

     {

       "name": "Salsa", "type": "SAUCE",

       "\_links": {

         "self": { "href": "http://localhost:8080/ingredients/SLSA" }

       }

     }

   ],

   "\_links": {

     "self": { "href": "http://localhost:8080/design/4" }

   }

 },

 ...

 ]

 },

 "\_links": {

   "recents": {

     "href": "http://localhost:8080/design/recent"

   }

 }

}

Этот конкретный вариант HATEOAS известен как HAL (Hypertext Application Language; http://stateless.co/hal\_specification.html), простой и обычно используемый формат для встраивания гиперссылок в JSON ответы.

Хотя этот листинг не такой лаконичный, как раньше, он предоставляет некоторую полезную информацию. Каждый элемент в этом новом списке тако включает в себя свойство с именем \_links, которое содержит гиперссылки для клиента для навигации по API. В этом примере у обоих тако и ингредиентов есть собственные ссылки для ссылки на эти ресурсы, и весь список имеет ссылку recents, которая ссылается на себя.

Если клиентскому приложению необходимо выполнить HTTP-запрос к тако в массиве, его не нужно формировать на основе знания, как будет выглядеть URL ресурса тако. Вместо известно, что нужно запросить собственную ссылку, которая отображается на http: //localhost:8080/design/4. Если клиент хочет иметь дело с конкретным ингредиентом, ему нужно только перейти по ссылке на себя для этого ингредиента.

Проект Spring HATEOAS обеспечивает поддержку гиперссылок в Spring. Он предлагает набор классов и ассемблеров ресурсов, которые можно использовать для добавления ссылок на ресурсы перед их возвратом из контроллера Spring MVC.

Чтобы включить гипермедиа в Taco Cloud API, вам нужно добавить starter Spring HATEOAS в зависимости:

<dependency>

 <groupId>org.springframework.boot</groupId>

 <artifactId>spring-boot-starter-hateoas</artifactId>

</dependency>

Этот стартер не только добавляет Spring HATEOAS в classpath проекта, но также предоставляет автоконфигурацию для включения Spring HATEOAS. Все, что вам нужно сделать, это переработать ваши контроллеры так, чтобы они возвращали типы ресурсов вместо типов доменов.

Вы начнете с добавления гипермедиа-ссылок в список последних тако, возвращаемых GET-запросом в /design/recent.

6.2.1 Добавление гиперлинков

Spring HATEOAS предоставляет два основных типа, которые представляют гиперссылочные ресурсы: Resource и Resources. Тип Resource представляет один ресурс, тогда как Resources представляет собой набор ресурсов. Оба типа способны содержать ссылки на другие ресурсы. При возврате из метода Spring MVC REST контроллера ссылок, они будут включены в JSON (или XML), полученный клиентом.

Чтобы добавить гиперссылки к списку недавно созданных тако, вам нужно будет вернуться к методу recentTacos(), показанному в листинге 6.2. Исходная реализация возвращает List<Taco>, который был прекрасным решением несколько страниц назад, но вам понадобилось, чтобы возвращались  объекты ресурсов вместо этого. В следующем листинге показана новая реализация recentTacos(), которая включает в себя первые шаги по внедрению гиперссылок в списке недавно разработанных тако.

Листинг 6.4. Добавление гиперссылок на ресурсы

@GetMapping("/recent")

public Resources<Resource<Taco>> recentTacos() {

 PageRequest page = PageRequest.of(

   0, 12, Sort.by("createdAt").descending());

 List<Taco> tacos = tacoRepo.findAll(page).getContent();

 Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);

 recentResources.add(

   new Link("http://localhost:8080/design/recent", "recents"));

 return recentResources;

}

В этой новой версии recentTacos(), вы больше не возвращаете непосредственно список тако. Вместо этого вы используете Resources.wrap() для переноса списка тако в качестве экземпляра Resources<Resource <Taco >>, который в конечном итоге возвращается из метода. Но перед возвратом объекта Resources вы добавляете ссылку, имя отношения которой - recents, а URL-адрес которой http://localhost:8080/design/recent. Как следствие, следующий фрагмент JSON включен в ресурс, возвращаемый из запроса API:

"\_links": {

 "recents": {

   "href": "http://localhost:8080/design/recent"

 }

}

Это хорошее начало, но у вас еще есть работа. На данный момент единственная добавленная вами ссылка - на весь перечень; никакие ссылки не добавляются ни к самим ресурсам тако, ни к ингредиентам каждого тако. Вы добавите их в ближайшее время. Но сначала давайте обратимся к жестко закодированному URL, который вы задали для ссылки на recents.

Жесткое кодирование URL-адреса это достаточно плохая идея. Если ваши амбиции Taco Cloud не ограничиваются только тем, что приложение запускается только на вашей собственной машине где разрабатывается ресурс, вам нужен способ не хардкодить localhost:8080. К счастью, Spring HATEOAS предоставляет помощь в виде компоновщиков ссылок.

Наиболее полезным из компоновщиков ссылок Spring HATEOAS является ControllerLinkBuilder. Этот компоновщик ссылок достаточно умен, чтобы знать, что такое имя хоста, без необходимости его жесткого кодирования. Кроме того, он предоставляет удобный API для создания ссылок относительно базового URL-адреса любого контроллера.

Используя ControllerLinkBuilder, вы можете переписать хардкордное задание Link в RecentTacos() следующими строками:

Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);

recentResources.add(

 ControllerLinkBuilder.linkTo(DesignTacoController.class)

   .slash("recent")

   .withRel("recents"));

Вам не только больше не нужно хардкодить имя хоста, вам также не нужно указывать путь /design. Вместо этого вы запрашиваете ссылку на DesignTacoController, базовый путь которого /design. ControllerLinkBuilder использует базовый путь контроллера в качестве основы создаваемого вами объекта Link.

Далее следует вызов одного из моих любимых методов в любом Spring проекте : slash (). Мне нравится этот метод, потому что он так кратко описывает, что именно он собирается делать. Он буквально добавляет косую черту (/) и заданное значение в URL. В результате путь URL-адреса /design/recent.

Наконец, вы указываете имя отношения для ссылки. В этом примере отношение называется recents.

Хотя я большой поклонник метода slash(), у ControllerLinkBuilder есть еще один метод, который может помочь устранить любое жесткое кодирование, связанное с URL-адресами ссылок. Вместо того, чтобы вызывать slash(), вы можете вызвать linkTo(), передав ему в метод контроллер, чтобы ControllerLinkBuilder получал базовый URL как из базового пути контроллера, так и из сопоставленного пути метода. Следующий код написан с использованием метода linkTo():

Resources<Resource<Taco>> recentResources = Resources.wrap(tacos);

recentResources.add(

 linkTo(methodOn(DesignTacoController.class).recentTacos())

   .withRel("recents"));

Здесь я решил статически включить методы linkTo() и methodOn() (оба из ControllerLinkBuilder), чтобы облегчить чтение кода. Метод methodOn() берет класс контроллера и позволяет вам вызвать метод recentTacos(), который перехватывается ControllerLinkBuilder и используется для определения не только базового пути контроллера, но и пути, сопоставленного с recentTacos(). Теперь весь URL-адрес получен из сопоставлений контроллера, и нет никакого хардкода. Великолепно!

6.2.2 Создание ресурсов ассемблеров (assemblers)

Теперь вам нужно добавить ссылки на ресурс тако, содержащийся в списке. Один из вариантов - циклически проходить по каждому из элементов Resource<Taco>, содержащихся в объекте Resources, добавляя ссылку на каждый из них по отдельности. Но это немного утомительно, и вам нужно будет повторять этот код цикла в API везде, где вы возвращаете список ресурсов тако.

Нам нужна другая тактика.

Вместо того чтобы позволить Resources.wrap () создать объект Resource для каждого тако в списке, вы определите служебный класс, который преобразует объекты Taco в новый объект TacoResource. Объект TacoResource будет очень похож на Taco, но он также будет иметь возможность переносить ссылки. Следующий листинг показывает, как может выглядеть TacoResource.

**Листинг 6.5. Тако-ресурс, несущий данные домена и список гиперссылок**

package tacos.web.api;

import java.util.Date;

import java.util.List;

import org.springframework.hateoas.ResourceSupport;

import lombok.Getter;

import tacos.Ingredient;

import tacos.Taco;

public class TacoResource extends ResourceSupport {

 @Getter

 private final String name;

 @Getter

 private final Date createdAt;

 @Getter

 private final List<Ingredient> ingredients;

 public TacoResource(Taco taco) {

   this.name = taco.getName();

   this.createdAt = taco.getCreatedAt();

   this.ingredients = taco.getIngredients();

 }

}

Во многом TacoResource ничем не отличается от доменного типа Taco. У них обоих есть свойства name, createAt и ingredients. Но TacoResource расширяет ResourceSupport для наследования списка объектов Link и методов для управления списком ссылок.

Более того, TacoResource не включает свойство id из Taco. Это потому, что нет необходимости предоставлять какие-либо специфичные для базы данных идентификаторы в API. Самостоятельная ссылка ресурса будет служить идентификатором ресурса с точки зрения клиента API.

**ПРИМЕЧАНИЕ**Домены и ресурсы: отдельные или одинаковые? Некоторые разработчики Spring могут объединить свои доменные типы и ресурсные в один тип, если их типы доменов расширяют ResourceSupport. Так тоже можно, нет правильного или неправильного ответа относительно того, какой путь правильный. Я выбрал создание отдельного типа ресурса, чтобы в Taco не было ненужных загромождений ссылками на ресурсы в тех случаях, когда ссылки не нужны. Кроме того, создав отдельный тип ресурса, я смог легко опустить свойство id, чтобы оно не отображалось в API.

TacoResource имеет единственный конструктор, который принимает Taco и копирует соответствующие свойства из Taco в его собственные свойства. Это облегчает преобразование объекта Taco в TacoResource. Но если вы остановитесь на этом, вам все равно понадобится цикл для преобразования списка объектов Taco в Resources<TacoResource>.

Чтобы помочь в преобразовании объектов Taco в объекты TacoResource, вы также собираетесь создать ассемблер ресурсов. Следующий список - это то, что вам нужно.

**Листинг 6.6. Ассемблер ресурсов, который собирает тако-ресурсы**

package tacos.web.api;

import org.springframework.hateoas.mvc.ResourceAssemblerSupport;

import tacos.Taco;

public class TacoResourceAssembler

   extends ResourceAssemblerSupport<Taco, TacoResource> {

 public TacoResourceAssembler() {

   super(DesignTacoController.class, TacoResource.class);

 }

@Override

 protected TacoResource instantiateResource(Taco taco) {

   return new TacoResource(taco);

 }

 @Override

 public TacoResource toResource(Taco taco) {

   return createResourceWithId(taco.getId(), taco);

 }

}

TacoResourceAssembler имеет конструктор по умолчанию, который сообщает суперклассу (ResourceAssemblerSupport), что он будет использовать DesignTacoController для определения базового пути для любых URL-адресов в ссылках, которые он создает при создании TacoResource.

Метод instantiateResource() переопределяется для создания экземпляра TacoResource с данным Taco. Этот метод необязательный, если TacoResource имеет конструктор по умолчанию. В этом случае, однако, TacoResource  требует для построения Taco, поэтому вы должны переопределить его.

Метод toResource() является единственным методом, строго обязательным при расширении ResourceAssemblerSupport. Здесь вы говорите ему создать объект TacoResource из Taco и автоматически дать ему собственную ссылку с URL-адресом, полученным из свойства id объекта Taco.

На первый взгляд, toResource(), похоже, имеет аналогичное назначение что и instantiateResource(), но они служат несколько иным целям. В то время как instantiateResource() предназначен только для создания экземпляра объекта Resource, метод toResource() предназначен не только для создания объекта Resource, но и для заполнения его ссылками. Под капотом toResource() находиться вызов instantiateResource().

Теперь настройте метод recentTacos(), чтобы использовать TacoResourceAssembler:

@GetMapping("/recent")

public Resources<TacoResource> recentTacos() {

 PageRequest page = PageRequest.of(

     0, 12, Sort.by("createdAt").descending());

 List<Taco> tacos = tacoRepo.findAll(page).getContent();

 List<TacoResource> tacoResources =

   new TacoResourceAssembler().toResources(tacos);

 Resources<TacoResource> recentResources =

   new Resources<TacoResource>(tacoResources);

 recentResources.add(

   linkTo(methodOn(DesignTacoController.class).recentTacos())

       .withRel("recents"));

 return recentResources;

}

Вместо того чтобы возвращать Resources<Resource<Taco >>, recentTacos() теперь возвращает Resources<TacoResource>, чтобы воспользоваться вашим новым типом TacoResource. После извлечения тако из репозитория вы передаете список объектов Taco методу toResources() класса TacoResourceAssembler. Этот удобный метод циклически перебирает все объекты Taco, вызывая метод toResource(), который вы переопределили в TacoResourceAssembler, чтобы создать список объектов TacoResource.

Используюя этот список TacoResource вы затем создаете объект Resources<TacoResource>, а затем заполняете его ссылкой на recents, как и в предыдущей версии recentTacos().

На этом этапе GET-запрос /design/ recent создаст список тако, каждый из которых имеет self ссылку и recents ссылку в самом списке. Но ингредиенты все равно останутся без ссылок. Чтобы решить эту проблему, вы создадите новый ассемблер ресурсов для ингредиентов:

package tacos.web.api;

import org.springframework.hateoas.mvc.ResourceAssemblerSupport;

import tacos.Ingredient;

 class IngredientResourceAssembler extends

       ResourceAssemblerSupport<Ingredient, IngredientResource> {

 public IngredientResourceAssembler() {

   super(IngredientController2.class, IngredientResource.class);

 }

 @Override

 public IngredientResource toResource(Ingredient ingredient) {

   return createResourceWithId(ingredient.getId(), ingredient);

 }

 @Override

 protected IngredientResource instantiateResource(

       Ingredient ingredient) {

   return new IngredientResource(ingredient);

 }

}

Как видите, IngredientResourceAssembler очень похож на TacoResourceAssembler, но работает с объектами Ingredient и IngredientResource вместо объектов Taco и TacoResource.

Говоря о IngredientResource,  он выглядит так:

package tacos.web.api;

import org.springframework.hateoas.ResourceSupport;

import lombok.Getter;

import tacos.Ingredient;

import tacos.Ingredient.Type;

public class IngredientResource extends ResourceSupport {

 @Getter

 private String name;

 @Getter

 private Type type;

 public IngredientResource(Ingredient ingredient) {

   this.name = ingredient.getName();

   this.type = ingredient.getType();

 }

}

Как и в случае с TacoResource, IngredientResource расширяет ResourceSupport и копирует соответствующие свойства из типа домена в свой собственный набор свойств (исключая свойство id).

Осталось лишь немного изменить TacoResource, чтобы он содержал объекты IngredientResource вместо объектов Ingredient:

package tacos.web.api;

import java.util.Date;

import java.util.List;

import org.springframework.hateoas.ResourceSupport;

import lombok.Getter;

import tacos.Taco;

public class TacoResource extends ResourceSupport {

 private static final IngredientResourceAssembler ingredientAssembler = new IngredientResourceAssembler();

 @Getter

 private final String name;

 @Getter

 private final Date createdAt;

 @Getter

 private final List<IngredientResource> ingredients;

 public TacoResource(Taco taco) {

   this.name = taco.getName();

   this.createdAt = taco.getCreatedAt();

   this.ingredients =ingredientAssembler.toResources(taco.getIngredients());

 }

}

Эта новая версия TacoResource создает final статический экземпляр IngredientResourceAssembler и использует его метод toResource() для преобразования списка Ingredient для данного объекта Taco в список IngredientResource.

Ваш недавний список тако теперь полностью снабжен гиперссылками, причем не только на себя (ссылка на recents), но также для всех его записей тако и ингредиентов этих тако. Ответ должен быть очень похож на JSON в листинге 6.3.

Вы можете остановиться здесь и перейти к следующей теме. Но сначала я расскажу о том, что меня беспокоило в листинге 6.3.

6.2.3 Именование встроенных отношений

Если вы внимательнее посмотрите на листинг 6.3, вы заметите, что элементы верхнего уровня выглядят так:

{

 "\_embedded": {

   "tacoResourceList": [

     ...

   ]

 }

}

Прежде всего, позвольте мне обратить ваше внимание на имя tacoResourceList под embedded. Это имя было получено из того факта, что объект Resources был создан из List<TacoResource>. Не то чтобы это было невероятно, но если бы вы реорганизовали имя класса TacoResource в другое, имя поля в результирующем JSON изменится, чтобы соответствовать ему. Это, вероятно, сломало бы любых клиентов, закодированных, чтобы рассчитывать на это имя.

Аннотация @Relation может помочь разорвать связь между именем поля JSON и именами классов типов ресурсов, как это определено в Java. Аннотируя TacoResource с помощью @Relation, вы можете указать, как Spring HATEOAS должен называть поле в результирующем JSON:

@Relation(value="taco", collectionRelation="tacos")

public class TacoResource extends ResourceSupport {

   ...

}

Здесь вы указали, что когда список объектов TacoResource используется в объекте Resources, он должен называться tacos. И хотя вы не используете это в нашем API, один объект TacoResource должен называться в JSON как taco.

В результате JSON, возвращенный из /design/recent, теперь будет выглядеть следующим образом (независимо от того, какой рефакторинг вы выполните в TacoResource):

{

 "\_embedded": {

   "tacos": [

     ...

   ]

 }

}

Spring HATEOAS делает добавление ссылок на ваш API довольно простым и понятным. Тем не менее, он добавил несколько строк кода, которые в противном случае вам не понадобились бы. По этой причине некоторые разработчики могут отказаться от использования HATEOAS в своих API, даже если это означает, что клиентский код может стать некорректным при изменении схемы URL API. Я призываю вас серьезно относиться к HATEOAS, а не лениться, не добавляя гиперссылки в ваши ресурсы.

Но если вы настаиваете на лени, то, возможно, для вас есть беспроигрышный сценарий, если вы используете Spring Data для своих репозиториев. Давайте посмотрим, как Spring Data REST может помочь вам автоматически создавать API на основе репозиториев, созданных с помощью Spring Data в главе 3.

6.3 Включение служб с поддержкой данных

Как вы видели в главе 3, Spring Data выполняет особую магию, автоматически создавая реализации репозитория на основе интерфейсов, которые вы определили в своем коде. Но у Spring Data есть еще одна хитрость, которая может помочь вам сгенерировать API для вашего приложения.

Spring Data REST - это еще один член семейства Spring Data, который автоматически создает API REST для репозиториев, созданных на основе Spring Data. Делая чуть больше, чем просто добавление Spring Data REST в вашу сборку, вы получаете API с операциями для каждого интерфейса репозитория, который вы определили.

Чтобы начать использовать Spring Data REST, добавьте в свою сборку следующую зависимость:

<dependency>

 <groupId>org.springframework.boot</groupId>

 <artifactId>spring-boot-starter-data-rest</artifactId>

</dependency>

Хотите верьте, хотите нет, но это все, что необходимо для предоставления REST API в проекте, который уже использует Spring Data для автоматизации репозиториев. Просто имея стартовый Spring REST в сборке, приложение получает автоконфигурацию, которая позволяет автоматически создавать REST API для любых репозиториев, которые были созданы Spring Data (включая Spring Data JPA, Spring Data Mongo и т. д.).

REST endpoint, которые создает Spring Data REST, по крайней мере так же хороши (и, возможно, даже лучше) тех, которые вы создали сами. Поэтому на данном этапе не стесняйтесь выполнить небольшую работу по удалению всех классов аннотированных @RestController, которые вы создали до этого момента, прежде чем двигаться дальше.

Чтобы опробовать endpoint, предоставляемые Spring Data REST, вы можете запустить приложение и начать тыкать в некоторые URL-адреса. Исходя из набора репозиториев, который вы уже определили для Taco Cloud, вы сможете выполнять запросы GET для тако, ингредиентов, заказов и пользователей.

Например, вы можете получить список всех ингредиентов, сделав запрос GET для /ingredients. Используя curl, вы можете получить что-то похожее на это (сокращенно, чтобы показать только первый ингредиент):

$ curl localhost:8080/ingredients

{

   "\_embedded" : {

       "ingredients" : [ {

           "name" : "Flour Tortilla",

           "type" : "WRAP",

           "\_links" : {

               "self" : {

                   "href" : "http://localhost:8080/ingredients/FLTO"

               },

               "ingredient" : {

                   "href" : "http://localhost:8080/ingredients/FLTO"

               }

           }

       },

       ...

       ]

   },

   "\_links" : {

       "self" : {

           "href" : "http://localhost:8080/ingredients"

       },

       "profile" : {

           "href" : "http://localhost:8080/profile/ingredients"

       }

   }

}

Вот Это Да! Не делая ничего, кроме добавления зависимости к вашей сборке, вы получаете не endpoint для ингредиентов, но и возвращающиеся ресурсы также содержат гиперссылки! Притворяясь клиентом этого API, вы также можете использовать curl, чтобы перейти по ссылке self для записи мучной лепешки (FLTO):

$ curl http://localhost:8080/ingredients/FLTO

{

 "name" : "Flour Tortilla",

 "type" : "WRAP",

 "\_links" : {

   "self" : {

     "href" : "http://localhost:8080/ingredients/FLTO"

   },

   "ingredient" : {

     "href" : "http://localhost:8080/ingredients/FLTO"

   }

 }

}

Чтобы не отвлекаться, мы не будем тратить больше времени в этой книге, копаясь во всех endpoint и опциях, созданных Spring Data REST. Но вы должны знать, что он также поддерживает методы POST, PUT и DELETE для endpoint, которые он создал. Верно: вы можете вызвать POST для /ingredients, чтобы создать новый ингредиент, и DELETE для /ingredients/FLTO, чтобы удалить лепешки из меню.

Одна вещь, которую вы, возможно, захотите сделать, это установить базовый путь для API, чтобы его endpoints были различны и не конфликтовали ни с какими контроллерами, которые вы пишете. (Фактически, если вы не удалите созданный ранее IngredientsController, он будет мешать endpoints /ingredients, предоставляемой Spring Data REST.) Чтобы настроить базовый путь для API, установите свойство spring.data.rest.base-path:

spring:

 data:

   rest:

     base-path: /api

Это установит базовый путь для endpoint Spring Data REST в /api. Следовательно, конечной точкой ингредиентов теперь является /api/ingredients. Теперь проверим новый базовый путь, запросив список тако:

$ curl http://localhost:8080/api/tacos

{

"timestamp": "2018-02-11T16:22:12.381+0000",

"status": 404,

"error": "Not Found",

"message": "No message available",

"path": "/api/tacos"

}

О, Боже!  Это сработало не совсем так, как ожидалось. У вас есть сущность Ingredient и интерфейс IngredientRepository, которые Spring Data REST предоставляет с помощью endpoint /api/ingredients. Итак, если у вас есть сущность Taco и интерфейс TacoRepository, почему Spring Data REST не предоставляет endpoint /api/tacos?

6.3.1 Настройка путей ресурсов и имен отношений

На самом деле Spring Data REST предоставляет вам endpoint для работы с тако. Но насколько бы умным ни был Spring Data REST, он показывает несколько странным, в том как он предоставляет конечную точку tacos.

При создании endpoint для репозиториев Spring Data, Spring Data REST пытается мультиплицировать связанный класс сущностей. Для объекта Ingredient endpoint  является /ingredients. Для сущностей Order и User это /orders и /users. Пока все логично.

Но иногда, например, с “taco”,, оно путается в слове, и множественная версия не совсем верна. Как выяснилось, Spring Data REST назвала“taco” как “tacoes”, поэтому, чтобы сделать запрос на тако, вы должны учесть это и запросить /api/tacoes:

% curl localhost:8080/api/tacoes

{

   "\_embedded" : {

       "tacoes" : [ {

           "name" : "Carnivore",

           "createdAt" : "2018-02-11T17:01:32.999+0000",

           "\_links" : {

               "self" : {

                   "href" : "http://localhost:8080/api/tacoes/2"

               },

               "taco" : {

                   "href" : "http://localhost:8080/api/tacoes/2"

               },

               "ingredients" : {

                   "href" : "http://localhost:8080/api/tacoes/2/ingredients"

               }

           }

       }]

   },

   "page" : {

       "size" : 20,

       "totalElements" : 3,

       "totalPages" : 1,

       "number" : 0

   }

}

Вы можете быть удивлены, откуда я знал, что “taco” будет истолковано как “tacoes”. Оказывается, Spring Data REST также предоставляет домашний ресурс, содержащий ссылки для всех открытых endpoint. Просто сделайте запрос GET к базовому пути API:

$ curl localhost:8080/api

{

  "\_links" : {

     "orders" : {

        "href" : "http://localhost:8080/api/orders"

     },

     "ingredients" : {

        "href" : "http://localhost:8080/api/ingredients"

     },

     "tacoes" : {

        "href" : "http://localhost:8080/api/tacoes{?page,size,sort}",

        "templated" : true

     },

     "users" : {

        "href" : "http://localhost:8080/api/users"

     },

     "profile" : {

        "href" : "http://localhost:8080/api/profile"

     }

  }

}

Как вы можете видеть, здесь есть ссылки для всех ваших сущностей. Все выглядит хорошо, за исключением ссылки tacos, где и имя отношения, и URL имеют нечетное множественное число “taco”.

Хорошей новостью является то, что вам не нужно мириться с этой маленькой причудой Spring Data REST. Добавив простую аннотацию к классу Taco, вы можете настроить как имя отношения, так и этот путь:

@Data

@Entity

@RestResource(rel="tacos", path="tacos")

public class Taco {

 ...

}

Аннотация @RestResource позволяет вам присвоить сущности любое имя и путь отношения. В этом случае вы устанавливаете их обоих на “tacos”. Теперь, когда вы запрашиваете домашний ресурс, вы видите ссылку tacos с правильным множественным числом:

"tacos" : {

 "href" : "http://localhost:8080/api/tacos{?page,size,sort}",

 "templated" : true

},

Это также сортирует путь для endpoint, чтобы вы могли создавать запросы к /api/tacos для работы с ресурсами taco.

Говоря о сортировке, давайте посмотрим, как можно отсортировать результаты с Spring Data REST endpoint.

6.3.2 Paging и sorting

Возможно, вы заметили, что все ссылки на домашнем ресурсе предлагают дополнительные параметры page, size, и sort. По умолчанию запросы возвращающие коллекцию, например /api/tacos, возвращают первую страницу с количеством объектов на ней числом до 20-ти. Но вы можете настроить размер страницы и отображаемую страницу, указав параметры page и size в своем запросе.

Например, чтобы запросить первую страницу тако, где размер страницы равен 5, вы можете выполнить следующий запрос GET (используя curl):

$ curl "localhost:8080/api/tacos?size=5"

Предположим, что общее количество тако более пяти, вы можете запросить вторую страницу тако, добавив параметр page:

$ curl "localhost:8080/api/tacos?size=5&page=1"

Обратите внимание, что параметр страницы начинается с нуля, что означает, что запрос страницы 1 на самом деле запрашивает вторую страницу. (Вы также заметите, что многие запросы командной строки смещаются над амперсандом в запросе, поэтому я привел весь URL в предыдущей команде curl.)

Вы можете использовать манипуляции со строками, чтобы добавить эти параметры в URL, но HATEOAS приходит на помощь, предлагая ссылки на первую, последнюю, следующую и предыдущие страницы в ответе:

"\_links" : {

   "first" : {

       "href" : "http://localhost:8080/api/tacos?page=0&size=5"

   },

   "self" : {

       "href" : "http://localhost:8080/api/tacos"

   },

   "next" : {

       "href" : "http://localhost:8080/api/tacos?page=1&size=5"

   },

   "last" : {

       "href" : "http://localhost:8080/api/tacos?page=2&size=5"

   },

   "profile" : {

       "href" : "http://localhost:8080/api/profile/tacos"

   },

   "recents" : {

       "href" : "http://localhost:8080/api/tacos/recent"

   }

}

С этими ссылками клиент API не должен отслеживать, на какой странице он находится, и объединять параметры с URL. Вместо этого он должен просто знать, что нужно искать одну из этих навигационных ссылок на странице по ее имени и следовать за ней.

Параметр sort позволяет сортировать полученный список по любому свойству объекта. Например, вам нужен способ получить 12 последних созданных тако. Вы можете сделать это, указав следующее сочетание параметров страницы и сортировки:

$ curl "localhost:8080/api/tacos?sort=createdAt,desc&page=0&size=12"

Здесь параметр sort указывает, что должна произвестись сортировка по свойству createdDate в порядке убывания (чтобы самые новые тако были первыми). Параметры page и size указывают, что вы должны увидеть первую страницу с 12 тако.

Это именно то, что нужно UI, чтобы показать самые последние созданные тако. Результат примерно такой же, как /design/recent endpoint, которую вы определили в DesignTacoController ранее в этой главе.

Однако есть небольшая проблема. Код пользовательского интерфейса должен быть жестко запрограммирован, чтобы запросить список тако с этими параметрами. Конечно, это будет работать. Но вы добавляете некоторую хрупкость клиенту, делая его слишком осведомленным о том, как создать запрос API. Было бы здорово, если бы клиент мог найти URL-адрес из списка ссылок. И было бы еще круче, если бы URL был более лаконичным, как /design/recent endpoint, которую вы использовали ранее.

6.3.3 Добавление пользовательских endpoint

Spring Data REST отлично подходит для создания endpoint для выполнения операций CRUD с репозиториями Spring Data. Но иногда вам нужно отойти от CRUD API по умолчанию и создать endpoint, которая доходит до сути проблемы.

Ничто не мешает вам реализовать любой endpoint, который вы хотите, в аннотированном компоненте @RestController, чтобы дополнить то, что Spring Data REST генерирует автоматически. Фактически, вы можете воскресить DesignTacoController, описанный ранее в этой главе, и он все равно будет работать вместе с конечными точками, предоставленными Spring Data REST.

Но когда вы пишете свои собственные API контроллеров, их endpoint кажутся несколько оторванными от endpoint Spring Data REST по нескольким причинам:

* Ваши собственные endpoint  контроллера не отображаются в соответствии с базовым путем Spring Data REST. Вы можете принудительно назначить префиксу их сопоставлений любой базовый путь, который вы хотите, включая базовый путь Spring Data REST, но если базовый путь должен был измениться, вам необходимо отредактировать сопоставления контроллера, чтобы они соответствовали.
* Любые endpoint, которые вы определяете в своих собственных контроллерах, не будут автоматически включены в качестве гиперссылок в ресурсы, возвращаемые endpoint Spring Data REST. Это означает, что клиенты не смогут обнаружить ваши пользовательские endpoint  с именем отношения.

Давайте сначала обратимся к проблеме базового пути. Spring Data REST включает в себя @RepositoryRestController, новую аннотацию для аннотирования классов контроллеров, чьи отображения должны принимать базовый путь, который совпадает с тем, который настроен для endpoint Spring Data REST. Проще говоря, все сопоставления в контроллере, помеченном @RepositoryRestController, будут иметь свой путь с префиксом со значением свойства spring.data.rest.base-path (которое вы настроили как / api).

Вместо того, чтобы воскрешать DesignTacoController, в котором было несколько методов-обработчиков, которые вам не нужны, вы создадите новый контроллер, который содержит только метод recentTacos(). RecentTacosController в следующем листинге помечен аннотацией @RepositoryRestController для принятия базового пути Spring Data REST для его сопоставлений запросов (для request mapping).

**Листинг 6.7. Применение базового пути Spring Data REST к контроллеру**

package tacos.web.api;

import static org.springframework.hateoas.mvc.ControllerLinkBuilder.\*;

import java.util.List;

import org.springframework.data.domain.PageRequest;

import org.springframework.data.domain.Sort;

import org.springframework.data.rest.webmvc.RepositoryRestController;

import org.springframework.hateoas.Resources;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

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

import tacos.Taco;

import tacos.data.TacoRepository;

@RepositoryRestController

public class RecentTacosController {

   private TacoRepository tacoRepo;

   public RecentTacosController(TacoRepository tacoRepo) {

       this.tacoRepo = tacoRepo;

   }

   @GetMapping(path="/tacos/recent", produces="application/hal+json")

   public ResponseEntity<Resources<TacoResource>> recentTacos() {

       PageRequest page = PageRequest.of(

           0, 12, Sort.by("createdAt").descending());

       List<Taco> tacos = tacoRepo.findAll(page).getContent();

       List<TacoResource> tacoResources = new TacoResourceAssembler().toResources(tacos);

       Resources<TacoResource> recentResources = new Resources<TacoResource>(tacoResources);

       recentResources.add(

           linkTo(methodOn(RecentTacosController.class).recentTacos())

       .withRel("recents"));

       return new ResponseEntity<>(recentResources, HttpStatus.OK);

   }

}

Несмотря на то, что @GetMapping сопоставляется с путем /tacos/recent, аннотация @RepositoryRestController на уровне класса гарантирует, что к нему будет добавлен базовый путь Spring Data REST. Метод recentTacos() будет обрабатывать запросы GET для /api/tacos/recent.

Важно отметить, что хотя @RepositoryRestController назван так же, как @RestController, он не обладает той же семантикой, что и @RestController. В частности, он не гарантирует, что значения, возвращаемые из методов-обработчиков, автоматически записываются в тело ответа. Поэтому вам нужно либо аннотировать метод с помощью @ResponseBody, либо возвращать ResponseEntity, который оборачивает данные ответа. Здесь решили вернуть ResponseEntity.

При использовании RecentTacosController в запросах на /api/tacos/recent будет возвращено до 15 самых последних созданных тако без необходимости разбивать на страницы и сортировать параметры в URL-адресе. Но он все еще не появляется в списке гиперссылок при запросе /api/tacos. Давайте исправим это.

6.3.4 Добавление пользовательских гиперссылок в endpoint Spring Data

Если endpoint недавно созданных тако не входит в число гиперссылок, возвращаемых из /api/tacos, как клиент будет знать, как получить самые последние тако? Он должен либо угадать, либо использовать параметры страницы и сортировки. В любом случае, это будет жестко закодировано в клиентском коде, что не идеально.

Однако, объявив ресурс обработчик bean-компонентов (resource processor bean), вы можете добавить ссылки в список ссылок Spring Data REST. Spring Data HATEOAS предлагает ResourceProcessor, интерфейс для управления ресурсами до их возврата через API. Для ваших целей вам нужна реализация ResourceProcessor, которая добавляет recents ссылку на любой ресурс типа PagedResources<Resource<Taco> > (тип, возвращаемый для /api/tacos endpoint). В следующем листинге показан метод объявления bean-компонента, определяющий ResourceProcessor.

**Листинг 6.8. Добавление пользовательских ссылок в REST Spring Data endpoint**

@Bean

public ResourceProcessor<PagedResources<Resource<Taco>>>

       tacoProcessor(EntityLinks links) {

   return new ResourceProcessor<PagedResources<Resource<Taco>>>() {

       @Override

       public PagedResources<Resource<Taco>> process(

               PagedResources<Resource<Taco>> resource) {

           resource.add(

               links.linkFor(Taco.class)

               .slash("recent")

               .withRel("recents"));

           return resource;

       }

   };

}

ResourceProcessor, показанный в листинге 6.8, определяется как анонимный внутренний класс и объявляется как bean-компонент, создаваемый в контексте приложения Spring. Spring HATEOAS обнаружит этот bean-компонент (как и любые другие bean-компоненты типа ResourceProcessor) автоматически и применяет их к соответствующим ресурсам. В этом случае, если из контроллера возвращается PagedResources<Resource<Taco>>, он получит ссылку на самые последние созданные тако. Это включает в себя ответ на запросы для /api/tacos.

Итог

* REST endpoint могут быть созданы с помощью Spring MVC с контроллерами, которые следуют той же модели программирования, что и контроллеры, ориентированные на браузер.
* Методы обработчика контроллера могут быть аннотированы с помощью @ResponseBody или возвращать объекты ResponseEntity для обхода модели, просмотра и записи данных непосредственно в тело ответа.
* Аннотация @RestController упрощает контроллеры REST, устраняя необходимость использования @ResponseBody в методах-обработчиках.
* Spring HATEOAS обеспечивает включение гиперссылок ресурсов, возвращаемых контроллерами Spring MVC.
* Репозитории Spring Data могут автоматически отображаться как API REST с помощью Spring Data REST.

[Spring in Action Covers Spring 5.0 перевод на русский. Глава 7](https://beginner-mercenary.blogspot.com/2019/05/spring-in-action-covers-spring-50-7.html)

REST сервисы

 В этой главе рассматриваются

* Использование RestTemplate для REST API
* Навигация по API гипермедиа с помощью Traverson

Вы когда-нибудь ходили в кино и были единственный человек в театре? Это, безусловно, замечательный опыт, который по сути является приватным просмотром фильма. Вы можете выбрать любое место, которое захотите, поговорить с персонажами на экране и, возможно, даже открыть свой телефон и написать об этом в Твиттере, чтобы никто не рассердился за то, что нарушил их просмотр фильма. И самое приятное то, что никто больше не портит фильм для вас!

Это случалось не часто со мной. Но когда это произошло, я подумал, что бы случилось, если бы я не появился. Они все еще показали бы фильм? Герой все еще спас бы день? Должны ли сотрудники театра убирать театр после окончания фильма?

Фильм без аудитории - это как API без клиента. Он готов принимать и предоставлять данные, но если API никогда не вызывается, действительно ли это API? Как и кот Шредингера, мы не можем знать, активен ли API или возвращает ответы HTTP 404, пока мы не отправим к нему запрос.

В предыдущей главе мы сосредоточились на определении REST endpoint, которые могут использоваться некоторыми клиентами, внешними по отношению к вашему приложению. Хотя движущей силой для разработки такого API было одностраничное приложение Angular, которое служило веб-сайтом Taco Cloud, реальность такова, что клиент может быть любым приложением на любом языке - даже другим приложением Java.

Приложения Spring нередко предоставляют API и отправляют запросы к API другого приложения. Фактически, это становится распространенным в мире микросервисов. Поэтому стоит потратить немного времени на изучение того, как использовать Spring для взаимодействия с REST API.

Приложение Spring может использовать REST API с

* RestTemplate - простой, синхронный REST-клиент, предоставляемый ядром Spring Framework.
* Traverson - синхронный REST-клиент с поддержкой гиперссылок, предоставляемый Spring HATEOAS. Вдохновленный из одноименной библиотеки JavaScript.
* WebClient - реактивный, асинхронный клиент REST, представленный в Spring 5.

Я отложу обсуждение WebClient до тех пор, пока мы не рассмотрим реактивную веб-инфраструктуру Spring в главе 11. Сейчас мы сосредоточимся на двух других REST клиентах, начиная с RestTemplate.

7.1 Использование REST endpoint с RestTemplate

Есть много, что входит во взаимодействие с ресурсом REST с точки зрения клиента-в основном скука и шаблонность. Работая с низкоуровневыми библиотеками HTTP, клиент должен создать экземпляр клиента и объект запроса, выполнить запрос, интерпретировать ответ, сопоставить ответ с объектами домена и обработать любые исключения, которые могут быть брошены по пути. И все это шаблонное повторяется, независимо от того, какой HTTP-запрос отправляется.

Чтобы избежать такого стандартного кода, Spring предоставляет RestTemplate. Так же, как JDBCTemplate обрабатывает уродливые части работы с JDBC, RestTemplate освобождает вас от скуки при использовании ресурсов REST.

RestTemplate предоставляет 41 метод для взаимодействия с ресурсами REST. Вместо того чтобы изучать все методы, которые он предлагает, проще рассмотреть только дюжину уникальных операций, каждая из которых перегружена, и в конечном итоге составляет полный набор в 41 метод. Эти 12 операций описаны в таблице 7.1.

**Таблица 7.1 RestTemplate определяет 12 уникальных операций, каждая из которых перегружена, обеспечивая в общей сложности 41 метод.**

Метод - Описание

delete(...) - Выполняет HTTP-запрос на удаление ресурса по указанному URL-адресу

exchange(...) - Выполняет указанный метод HTTP для URL, возвращая ResponseEntity, содержащий объект, сопоставленный с телом ответа

execute(...) - Выполняет указанный метод HTTP для URL, возвращая объект, сопоставленный с телом ответа

getForEntity(...) - Отправляет HTTP-запрос GET, возвращая ResponseEntity, содержащий объект, сопоставленный с телом ответа.

getForObject(...) - Отправляет HTTP-запрос GET, возвращая объект, сопоставленный с телом ответа.

headForHeaders(...) - Отправляет запрос HTTP HEAD, возвращая заголовки HTTP для указанного URL ресурса

optionsForAllow(...) - Отправляет запрос HTTP OPTIONS, возвращая заголовок Allow для указанного URL

patchForObject(...) - Отправляет запрос HTTP PATCH, возвращая полученный объект, сопоставленный с телом ответа.

postForEntity(...) - помещает POST данные в URL, возвращая ResponseEntity, содержащий объект, сопоставленный с телом ответа

postForLocation(...) - помещает POST данные в URL, возвращая URL вновь созданного ресурса

postForObject(...) - помещает POST данные в URL, возвращая объект, сопоставленный с телом ответа

put(...) - Помещает PUT данные ресурса в указанный URL

За исключением TRACE, RestTemplate имеет по крайней мере один метод для каждого из стандартных методов HTTP. Кроме того, execute () и exchange () предоставляют низкоуровневые методы общего назначения для отправки запросов любым методом HTTP.

Большинство методов в таблице 7.1 перегружены в три формы методов:

* Можно принять спецификацию String URL с параметрами URL, указанными в списке переменных аргументов.
* Можно принять спецификацию String URL с параметрами URL, указанными в Map<String,String>.
* В качестве спецификации URL-адреса принимается java.net.URI без поддержки параметризованных URL-адресов.

Как только вы разберетесь с 12 операциями, предоставляемыми RestTemplate, и с тем, как работает каждая с вариантами форм методов, вы сможете приступить к написанию ресурсоемких REST-клиентов.

Чтобы использовать RestTemplate, вам нужно либо создать экземпляр там, где вам это нужно

RestTemplate rest = new RestTemplate();

или вы можете объявить его как bean и внедрить его там, где вам это нужно:

@Bean

public RestTemplate restTemplate() {

 return new RestTemplate();

}

Давайте рассмотрим операции RestTemplate, рассмотрев те из них, которые поддерживают четыре основных метода HTTP: GET, PUT, DELETE и POST. Начнем с getForObject() и getForEntity() - метод GET.

7.1.1 GET (получение) ресурсов

Предположим, вы хотите получить компонент из Taco Cloud API. Предполагая, что API не поддерживает HATEOAS, вы можете использовать getForObject() для извлечения ингредиента. Например, следующий код использует RestTemplate для извлечения объекта Ingredient по его идентификатору:

public Ingredient getIngredientById(String ingredientId) {

   return rest.getForObject("http://localhost:8080/ingredients/{id}",

       Ingredient.class, ingredientId);

}

Здесь вы используете вариант getForObject(), который принимает String URL и использует список переменных для переменных URL. Параметр ingredientId, передаваемый в getForObject(), используется для заполнения элемента {id} в указанном URL. Хотя в этом примере есть только одна переменная URL, важно знать, что параметры переменных назначаются местозаполнителям в том порядке, в котором они указаны.

Второй параметр в getForObject() - это тип, к которому должен быть приведен ответ. В этом случае данные ответа (вероятно, в формате JSON) должны быть десериализованы в объект Ingredient, который будет возвращен.

Кроме того, вы можете использовать Map, чтобы указать переменные URL:

public Ingredient getIngredientById(String ingredientId) {

   Map<String,String> urlVariables = new HashMap<>();

   urlVariables.put("id", ingredientId);

   return rest.getForObject("http://localhost:8080/ingredients/{id}",

       Ingredient.class, urlVariables);

}

В этом случае значение ingredientId отображается на ключ id. Когда запрос выполняется, заполнитель {id} заменяется записью map-ы, ключом которой является id.

Использование параметра URI является более сложным процессом, требующим создания объекта URI перед вызовом getForObject(). В остальном он похож на оба других варианта:

public Ingredient getIngredientById(String ingredientId) {

   Map<String,String> urlVariables = new HashMap<>();

   urlVariables.put("id", ingredientId);

   URI url = UriComponentsBuilder

       .fromHttpUrl("http://localhost:8080/ingredients/{id}")

       .build(urlVariables);

   return rest.getForObject(url, Ingredient.class);

}

Здесь объект URI определен из спецификации String, а его заполнители заполнены из записей в Map, как и в предыдущем варианте getForObject(). Метод getForObject() - это простой способ извлечения ресурса. Но если клиенту нужно больше, чем содержимое body, вы можете рассмотреть возможность использования getForEntity().

getForEntity() работает почти так же, как getForObject(), но вместо возврата объекта домена, представляющего содержимым body ответа, он возвращает объект ResponseEntity, который оборачивает этот объект домена. ResponseEntity предоставляет доступ к дополнительным деталям ответа, таким как заголовки ответа.

Например, предположим, что в дополнение к данным ингредиента вы хотите посмотреть заголовок Date из ответа. С getForEntity() это становится простым:

public Ingredient getIngredientById(String ingredientId) {

   ResponseEntity<Ingredient> responseEntity = rest.getForEntity("http://localhost:8080/ingredients/{id}",

       Ingredient.class, ingredientId);

   log.info("Fetched time: " +

       responseEntity.getHeaders().getDate());

   return responseEntity.getBody();

}

Метод getForEntity() перегружен теми же параметрами, что и getForObject(), поэтому вы можете предоставить переменные URL-адреса в качестве параметра списка переменных или вызвать getForEntity() с объектом URI.

7.1.2  PUT ресурсов

Для отправки HTTP PUT-запросов, RestTemplate предлагает метод put(). Все три перегруженных варианта put() принимают Object, который должен быть сериализован и отправлен по указанному URL. Что касается самого URL, он может быть указан как объект URI или как String. И как c getForObject() и getForEntity(), переменные URL-адреса могут быть предоставлены либо как список аргументов переменной, либо как Map.

Предположим, что вы хотите заменить ресурс ингредиента данными из нового объекта Ingredient. Следующий код должен сделать это:

public void updateIngredient(Ingredient ingredient) {

   rest.put("http://localhost:8080/ingredients/{id}",

       ingredient,

       ingredient.getId());

}

Здесь URL задан в виде строки и содержит заполнитель, который заменяется свойством id данного объекта Ingredient. Данные для отправки - это сам объект Ingredient. Метод put() возвращает void, поэтому вам ничего не нужно делать для обработки возвращаемого значения.

7.1.3 DELETE ресурсов

Предположим, что Taco Cloud больше не предлагает ингредиент и хочет полностью удалить его в качестве опции. Чтобы это произошло, вы можете вызвать метод delete() из RestTemplate:

public void deleteIngredient(Ingredient ingredient) {

   rest.delete("http://localhost:8080/ingredients/{id}",

       ingredient.getId());

}

В этом примере для delete() передаются только URL-адрес (указанный как String) и значение переменной URL-адреса. Но, как и в случае с другими методами RestTemplate, URL-адрес может быть указан как объект URI или параметры URL-адреса представлены как Map.

7.1.4 POST данных ресурсов

Теперь допустим, что вы добавили новый ингредиент в меню Taco Cloud. Это сделает HTTP POST-запрос к .../ingredients endpoint с данными ингредиентов в теле запроса. RestTemplate имеет три способа отправки запроса POST, каждый из которых имеет одинаковые перегруженные варианты для указания URL. Если вы хотите получить вновь созданный ресурс Ingredient после POST-запроса, вы должны использовать postForObject() следующим образом:

public Ingredient createIngredient(Ingredient ingredient) {

   return rest.postForObject("http://localhost:8080/ingredients",

       ingredient,

       Ingredient.class);

}

Этот вариант метода postForObject() принимает String URL спецификацию, объект, который должен быть отправлен на сервер, и доменный тип, с которым должно быть связано тело ответа. Хотя в этом случае вы не пользуетесь этим но, четвертым параметром может быть Map значения переменной URL-адреса или список переменных параметров для замены в URL.

Если для нужд клиента требуется получить ссылку на расположении только что созданного ресурса, вы можете вызвать postForLocation() :

public URI createIngredient(Ingredient ingredient) {

   return rest.postForLocation("http://localhost:8080/ingredients",ingredient);

}

Обратите внимание, что postForLocation() работает так же, как postForObject(), за исключением того, что он возвращает URI вновь созданного ресурса вместо самого объекта ресурса. Возвращенный URI получен из заголовка Location ответа. В случае, если вам понадобятся как местоположение, так и полезная нагрузка ответа, вы можете вызвать postForEntity():

public Ingredient createIngredient(Ingredient ingredient) {

  ResponseEntity<Ingredient> responseEntity = rest.postForEntity("http://localhost:8080/ingredients",

     ingredient, Ingredient.class);

  log.info("New resource created at " +

     responseEntity.getHeaders().getLocation());

  return responseEntity.getBody();

}

Хотя методы RestTemplate отличаются по своему назначению, они очень похожи в том, как они используются. Это позволяет легко понять RestTemplate и использовать его в клиентском коде.

С другой стороны, если API, который вы используете, включает в свой ответ гиперссылки, RestTemplate не так полезен. Конечно, можно получить более подробные данные ресурса с помощью RestTemplate и работать с содержимым и ссылками, содержащимися в нем, но это не тривиально. Вместо того, чтобы бороться с использованием гипермедиа API с RestTemplate, давайте обратим наше внимание на клиентскую библиотеку, созданную для таких целей - Traverson.

7.2 Навигация REST API с помощью Traverson

Traverson поставляется с Spring Data HATEOAS как готовое решение для использования гипермедиа API в приложениях Spring. Эта библиотека на основе Java основана на аналогичной библиотеке JavaScript с тем же именем (<https://github.com/traverson/traverson>).

Возможно, вы заметили, что имя Трэверсон звучит как «traverse on», что является хорошим способом описать, как оно используется. В этом разделе вы будете использовать API путем обхода API по именам отношений.

Работа с Traverson начинается с создания экземпляра объекта Traverson с базовым API URI:

Traverson traverson = new Traverson(URI.create("http://localhost:8080/api"), MediaTypes.HAL\_JSON);

Здесь я указал Traverson на базовый URL Taco Cloud (работает локально). Это единственный URL, который вам нужно указать Traverson-у. С этого момента вы будете перемещаться по API по именам отношений ссылок. Вы также укажете, что API будет генерировать ответы JSON с гиперссылками в стиле HAL, чтобы Traverson знал, как анализировать входящие данные ресурса. Как и RestTemplate, вы можете создать экземпляр объекта Traverson перед его использованием или объявить его как компонент, который будет введен везде, где это необходимо.

С объектом Traverson вы можете начать использовать API, перейдя по ссылкам. Например, предположим, что вы заинтересованы в получении списка всех ингредиентов. Из раздела 6.3.1 вы знаете, что ссылка ингредиентов имеет свойство href, которое ссылается на ресурс ingredients. Вам нужно перейти по этой ссылке:

ParameterizedTypeReference<Resources<Ingredient>> ingredientType =

 new ParameterizedTypeReference<Resources<Ingredient>>() {};

Resources<Ingredient> ingredientRes =

 traverson

   .follow("ingredients")

   .toObject(ingredientType);

 Collection<Ingredient> ingredients = ingredientRes.getContent();

Вызвав метод follow() для объекта Traverson, вы можете перейти к ресурсу, имя связи которого - ingredients. Теперь, когда клиент перешел к ингредиентам, вам нужно принять содержимое этого ресурса, вызвав toObject().

Метод toObject () требует, чтобы вы указали, в какой тип объекта считывать данные. Это может быть немного сложнее, учитывая, что вам нужно прочитать его как объект Resources<Ingredient>, а стирание типа Java затрудняет предоставление информации о типе для generic типа. Но создание ParameterizedTypeReference помогает в этом.

В качестве аналогии представьте, что вместо REST API это была домашняя страница на веб-сайте. И вместо кода REST клиента представьте, что вы просматриваете эту домашнюю страницу в браузере. Вы видите ссылку на странице с надписью «Ингредиенты» и переходите по этой ссылке, щелкая ее. По прибытии на следующую страницу вы читаете страницу, которая аналогична тому, как Traverson принял содержимое как объект Resources<Ingredient>.

Теперь давайте рассмотрим немного более интересный вариант использования. Допустим, вы хотите получить самые последние созданные тако. Начиная с домашнего ресурса, вы можете перейти к ресурсу с последними тако, например так:

ParameterizedTypeReference<Resources<Taco>> tacoType =

 new ParameterizedTypeReference<Resources<Taco>>() {};

Resources<Taco> tacoRes =

 traverson

   .follow("tacos")

   .follow("recents")

   .toObject(tacoType);

 Collection<Taco> tacos = tacoRes.getContent();

Здесь вы переходите по ссылке tacos, а затем оттуда по ссылке recents. Это приведет вас к интересующему вас ресурсу, поэтому вызов toObject() с соответствующей ParameterizedTypeReference даст вам то, что вы хотите. Метод .follow() можно упростить, перечислив имена отношений через запятую:

Resources<Taco> tacoRes =

 traverson

   .follow("tacos", "recents")

   .toObject(tacoType);

Как видите, Traverson упрощает навигацию по API с поддержкой HATEOAS и использование его ресурсов. Но он не предлагает никаких методов для записи или удаления по этим API. С другой стороны, RestTemplate может писать и удалять ресурсы, но не облегчает навигацию по API.

Если вам нужно не только перемещаться по API, но и обновлять или удалять ресурсы, вам нужно использовать RestTemplate и Traverson вместе. Traverson по-прежнему можно использовать для перехода по ссылке, где будет создан новый ресурс. Затем RestTemplate может быть предоставлена эта ссылка для выполнения POST, PUT, DELETE или любого другого HTTP-запроса.

Например, предположим, что вы хотите добавить новый ингредиент в меню Taco Cloud. Следующий метод addIngredient() объединяет Traverson и RestTemplate для добавления нового ингредиента по API:

private Ingredient addIngredient(Ingredient ingredient) {

 String ingredientsUrl = traverson

   .follow("ingredients")

   .asLink()

   .getHref();

 return rest.postForObject(ingredientsUrl,

   ingredient,

   Ingredient.class);

}

После перехода по ссылке ingredients вы запрашиваете саму ссылку, вызывая asLink(). По этой ссылке вы запрашиваете URL ссылки, вызывая getHref(). Имея URL-адрес, у вас есть все, что нужно для вызова postForObject() в экземпляре RestTemplate и сохранения нового ингредиента.

ИТОГ:

* Клиенты могут использовать RestTemplate для выполнения HTTP-запросов к REST API.
* Traverson позволяет клиентам перемещаться по API с помощью гиперссылок, встроенных в ответы.

[Spring in Action Covers Spring 5.0 перевод на русский. Глава 8](https://beginner-mercenary.blogspot.com/2019/07/spring-in-action-covers-spring-50-8.html)

**8.Отправка сообщений асинхронно**

Эта глава охватывает

* *Асинхронный обмен сообщениями*
* *Messages Отправка сообщений с помощью JMS, RabbitMQ и Kafka*
* *Вытягивание сообщений из broker*
* *Листинер для сообщений*

Сейчас 4:55 вечера пятницы. В нескольких минутах от долгожданного отпуска. У вас достаточно времени, чтобы доехать до аэропорта и успеть на самолет. Но прежде чем собраться и отправиться в путь, вы должны быть уверены, что ваш начальник и коллеги знают о состоянии работы, которую вы выполняли, чтобы в понедельник они могли узнать, где вы остановились. К сожалению, некоторые из ваших коллег уже пропустили выходные, а ваш начальник связывается с вами с митинга. Чем ты занимаешься?

Самый практичный способ сообщить о своем статусе и при этом сесть на самолет - это отправить быстрое электронное письмо своему боссу и коллегам с подробным описанием вашего прогресса и обещанием отправить открытку. Вы не знаете, где они находятся и когда они будут читать электронную почту, но вы знаете, что они в конечном итоге вернутся за свои столы и прочитают ее. Тем временем вы едете в аэропорт.

Синхронное взаимодействие, которое мы видели в REST, имеет свое предназначение. Но это не единственный стиль взаимодействия между приложениями, доступный для разработчиков. Асинхронный обмен сообщениями - это способ косвенной отправки сообщений из одного приложения в другое без ожидания ответа. Эта косвенность обеспечивает более слабую связь и большую масштабируемость между взаимодействующими приложениями.

В этой главе вы будете использовать асинхронный обмен сообщениями для отправки заказов с веб-сайта Taco Cloud в отдельное приложение кухни Taco Cloud, где будут готовиться тако. Мы рассмотрим три варианта, которые Spring предлагает для асинхронного обмена сообщениями: Java Message Service (JMS), RabbitMQ и Advanced Message Queuing Protocol (AMQP), Apache Kafka. В дополнение к базовой отправке и получению сообщений, мы рассмотрим поддержку Spring для POJO, управляемых сообщениями: способ получения сообщений, который напоминает EJB-компоненты, управляемые сообщениями (MDB).

8.1 Отправка сообщений с помощью JMS

JMS - это стандарт Java, который определяет общий API для работы с брокерами (broker) сообщений. Впервые представленный в 2001 году, JMS был подходом для асинхронного обмена сообщениями в Java в течение очень долгого времени. До JMS каждый брокер сообщений имел собственный API, что делало код обмена сообщениями приложения менее переносимым между брокерами. Но с JMS все совместимые реализации могут работать через общий интерфейс почти так же, как JDBC предоставил операциям с реляционными базами данных общий интерфейс.

Spring поддерживает JMS через абстракцию на основе шаблонов, известную как JmsTemplate. Используя JmsTemplate, легко отправлять сообщения по очередям и темам со стороны производителя (producer) и получать эти сообщения на стороне потребителя. Spring также поддерживает понятие сообщение-упраляемые POJO: простые объекты Java, которые реагируют на сообщения, поступающие в очередь или тему асинхронно.

Мы собираемся изучить поддержку Spring JMS, включая JmsTemplate и сообщение-упраляемые POJO. Но прежде чем вы сможете отправлять и получать сообщения, вам нужен брокер сообщений, который готов передавать эти сообщения между производителями и потребителями. Давайте начнем наше исследование Spring JMS, настроив брокер сообщений в Spring.

8.1.1 Настройка JMS

Прежде чем вы сможете использовать JMS, вы должны добавить JMS-клиент в сборку вашего проекта. С Spring Boot это очень просто. Все, что вам нужно сделать, это добавить starter зависимость в сборку. Однако сначала вы должны решить, собираетесь ли вы использовать Apache ActiveMQ или более нового брокера Apache ActiveMQ Artemis.

Если вы используете ActiveMQ, вам нужно добавить следующую зависимость в файл pom.xml вашего проекта:

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-activemq</artifactId>*

*</dependency>*

Если вы решили использовать ActiveMQ Artemis, starter зависимость должна выглядеть следующим образом:

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-artemis</artifactId>*

*</dependency>*

Artemis - это новое воплощение ActiveMQ следующего поколения, которое делает ActiveMQ устаревшим вариантом. Поэтому для Taco Cloud вы выберете Artemis. Но выбор в конечном итоге мало влияет на то, как вы будете писать код, который отправляет и получает сообщения. Единственным существенным отличием будет то, как вы настраиваете Spring для создания соединений с брокером.

По умолчанию Spring предполагает, что ваш брокер Artemis прослушивает localhost на порту 61616. Это хорошо для целей разработки, но как только вы будете готовы отправить свое приложение в продакшен, вам нужно будет установить несколько свойств, которые сообщат Spring, как получить доступ к брокеру. Свойства, которые вы найдете наиболее полезными, перечислены в таблице 8.1.

**Таблица 8.1 Свойства для настройки расположения и учетных данных брокера Artemis**

Свойство - Описание

*spring.artemis.host* - broker host

*spring.artemis.port* - broker’s port

*spring.artemis.user* - user использующийся для доступа к broker (опционально)

*spring.artemis.password* - password использующийся для доступа к broker (опционально)

Например, рассмотрим следующую запись файла application.yml, который может использоваться в параметрах, не относящихся к режиму разработке:

*spring:*

*artemis:*

*host: artemis.tacocloud.com*

*port: 61617*

*user: tacoweb*

*password: l3tm31n*

Это настраивает Spring для создания подключений к брокеру Artemis, который прослушивает artemis.tacocloud.com, порт 61617. Он также устанавливает учетные данные для приложения, которое будет взаимодействовать с этим брокером. Учетные данные не являются обязательными, но они рекомендуются для рабочих развертываний.

Если бы вы использовали ActiveMQ вместо Artemis, вам нужно было бы использовать специфичные для ActiveMQ свойства, перечисленные в таблице 8.2.

**Таблица 8.2 Свойства для настройки расположения и учетных данных брокера ActiveMQ**

Свойство - Описание

*spring.activemq.broker-url* - URL broker-а

*spring.activemq.user*- user использующийся для доступа к broker (опционально)

*spring.activemq.password* - password использующийся для доступа к broker (опционально)

*spring.activemq.in-memory* -  Стоит ли запускать брокер в памяти (по умолчанию: true)

Обратите внимание, что вместо того, чтобы предлагать отдельные свойства для имени хоста и порта брокера (посредника), адрес брокера ActiveMQ указывается с помощью одного свойства, spring.activemq.broker-url. URL должен быть tcp://URL , как показано в следующем фрагменте YAML:

*spring:*

*activemq:*

*broker-url: tcp://activemq.tacocloud.com*

*user: tacoweb*

*password: l3tm31n*

Независимо от того, выбираете ли вы Artemis или ActiveMQ, вам не нужно настраивать эти свойства для разработки, когда брокер работает локально.

Если вы используете ActiveMQ, вам, однако, необходимо установить для свойства spring.activemq.in-memory значение false, чтобы Spring не запускал брокер в памяти. Брокер в памяти может показаться полезным, но он полезен только тогда, когда вы будете использовать сообщения из того же приложения, которое их публикует (что имеет ограниченную полезность).

Вместо использования встроенного брокера вы должны установить и запустить брокера Artemis (или ActiveMQ), прежде чем двигаться дальше. Вместо того, чтобы повторять инструкции по установке здесь, я отсылаю вас к документации брокера:

Artemis—https://activemq.apache.org/artemis/docs/latest/using-server.html

ActiveMQ—http://activemq.apache.org/getting-started.html#GettingStarted-Pre-InstallationRequirements

С JMS-стартером в вашей сборке и брокером, ожидающим пересылки сообщений из одного приложения в другое, вы готовы начать отправку сообщений.

8.1.2 Отправка сообщений с помощью JmsTemplate

С JMS starter зависимостью (Artemis или ActiveMQ) в вашей сборке Spring Boot автоматически настроит JmsTemplate (среди прочего), который вы можете внедрить и использовать для отправки и получения

JmsTemplate является центральным элементом поддержки интеграции Spring JMS. Подобно другим шаблонно-ориентированным компонентам Spring, JmsTemplate устраняет много стандартного кода, который в противном случае потребовался бы для работы с JMS. Без JmsTemplate вам нужно было бы написать код для создания соединения и сеанса с брокером сообщений, а также дополнительный код для обработки любых исключений, которые могут возникнуть в процессе отправки сообщения. JmsTemplate фокусируется на том, что вы действительно хотите сделать: отправить сообщение.

JmsTemplate имеет несколько методов, которые полезны для отправки сообщений, включая следующие:

// Отправка сырых (raw) сообщений

*void send(MessageCreator messageCreator) throws JmsException;*

*void send(Destination destination, MessageCreator messageCreator) throws JmsException;*

*void send(String destinationName, MessageCreator messageCreator) throws JmsException;*

// Отправка сообщений сконвертированных из объектов

*void convertAndSend(Object message) throws JmsException;*

*void convertAndSend(Destination destination, Object message) throws JmsException;*

*void convertAndSend(String destinationName, Object message) throws JmsException;*

// Отправка сообщений, преобразованных из объектов с post-обработкойvoid convertAndSend(Object message,

*MessagePostProcessor postProcessor) throws JmsException;*

*void convertAndSend(Destination destination, Object message,*

*MessagePostProcessor postProcessor) throws JmsException;*

*void convertAndSend(String destinationName, Object message,*

*MessagePostProcessor postProcessor) throws JmsException;*

Как видите, на самом деле есть только два метода, send() и convertAndSend(), каждый из которых переопределен для поддержки различных параметров. И если вы посмотрите поближе, вы заметите, что различные формы convertAndSend() можно разбить на две подкатегории. Чтобы понять, что делают все эти методы, рассмотрим следующий список:

* Три метода send() требуют, чтобы MessageCreator создал объект Message.
* Три метода convertAndSend() принимают Object и автоматически преобразуют этот Object в Message.
* Три метода convertAndSend() автоматически преобразуют Object в Message, но также принимают MessagePostProcessor, позволяющий настроить Message до его отправки.

Кроме того, каждая из этих трех категорий методов состоит из трех переопределяющих методов, которые различаются тем, как указывается место назначения JMS (очередь или тема):

* Один метод не принимает параметр пункта назначения и отправляет сообщение в пункт назначения по умолчанию.
* Один метод принимает объект Destination, который указывает место назначения для сообщения.
* Один метод принимает String, которая указывает место назначения для сообщения по наименованию.

Чтобы эти методы работали, рассмотрим JmsOrderMessagingService в следующем листинге, который использует самую основную форму метода send().

**Листинг 8.1. Отправка заказа с помощью .send() в пункт назначения по умолчанию**

*package tacos.messaging;*

*import javax.jms.JMSException;*

*import javax.jms.Message;*

*import javax.jms.Session;*

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.jms.core.JmsTemplate;*

*import org.springframework.jms.core.MessageCreator;*

*import org.springframework.stereotype.Service;*

*@Service*

*public class JmsOrderMessagingService implements OrderMessagingService {*

*private JmsTemplate jms;*

*@Autowired*

*public JmsOrderMessagingService(JmsTemplate jms) {*

*this.jms = jms;*

*}*

*@Override*

*public void sendOrder(Order order) {*

*jms.send(new MessageCreator() {*

*@Override*

*public Message createMessage(Session session)*

*throws JMSException {*

*return session.createObjectMessage(order);*

*}*

*}*

*);*

*}*

*}*

Метод sendOrder() вызывает jms.send(), передавая анонимную внутреннюю реализацию класса MessageCreator. Эта реализация переопределяет createMessage() для создания нового сообщения объекта из заданного объекта Order.

Я не уверен насчет вас, но я думаю, что код в листинге 8.1, хотя и простой, немного неуклюжий. Церемония, связанная с объявлением анонимного внутреннего класса, усложняет простой вызов метода. Признавая, что MessageCreator является функциональным интерфейсом, вы можете немного привести в порядок метод sendOrder() с помощью лямбды:

*@Override*

*public void sendOrder(Order order) {*

*jms.send(session -> session.createObjectMessage(order));*

*}*

Но обратите внимание, что вызов jms.send() не указывает адресата. Чтобы это работало, вы также должны указать имя получателя по умолчанию в свойстве spring.jms.template.default-destination. Например, вы можете установить свойство в вашем файле application.yml следующим образом:

*spring:*

*jms:*

*template:*

*default-destination: tacocloud.order.queue*

Во многих случаях использование пункта назначения по умолчанию является самым простым выбором. Это позволяет вам указать имя получателя один раз, позволяя коду относиться только к отправке сообщений, независимо от того, куда они отправляются. Но если вам когда-либо понадобится отправить сообщение в пункт назначения, отличный от пункта назначения по умолчанию, вам нужно будет указать этот пункт назначения в качестве параметра send().

Один из способов сделать это - передать объект Destination в качестве первого параметра send(). Самый простой способ сделать это - объявить bean-объект Destination, а затем внедрить его в bean-компонент, выполняющий обмен сообщениями. Например, следующий bean-компонент объявляет Destination очереди Taco Cloud:

*@Bean*

*public Destination orderQueue() {*

*return new ActiveMQQueue("tacocloud.order.queue");*

*}*

Важно отметить, что ActiveMQQueue, использованный здесь, на самом деле от Artemis (из пакета org.apache.activemq.artemis.jms.client). Если вы используете ActiveMQ (не Artemis), есть также класс с именем ActiveMQQueue (из пакета org.apache.activemq.command).

Если этот целевой объект внедряется в JmsOrderMessagingService, вы можете использовать его для указания получателя при вызове send():

*private Destination orderQueue;*

*@Autowired*

*public JmsOrderMessagingService(JmsTemplate jms, Destination orderQueue) {*

*this.jms = jms;*

*this.orderQueue = orderQueue;*

*}*

*...*

*@Override*

*public void sendOrder(Order order) {*

*jms.send(*

*orderQueue,*

*session -> session.createObjectMessage(order));*

*}*

Указание адресата с помощью объекта Destination, подобного этому, дает вам возможность настроить Destination с использованием не только имени пункта назначения. Но на практике вы почти никогда не будете указывать ничего, кроме имени пункта назначения. Часто проще просто отправить имя в качестве первого параметра send():

*@Override*

*public void sendOrder(Order order) {*

*jms.send(*

*"tacocloud.order.queue",*

*session -> session.createObjectMessage(order));*

*}*

Хотя метод send() не особенно сложен в использовании (особенно, когда MessageCreator задается как лямбда-выражение), добавляется небольшая сложность, требующая предоставления MessageCreator. Разве не проще, если вам нужно только указать объект, который должен быть отправлен (и, возможно, пункт назначения)? Это кратко описывает, как работает convertAndSend (). Давайте взглянем.

Конвертация сообщения перед отправкой

Метод convertAndSend() в JmsTemplates упрощает публикацию сообщений, устраняя необходимость предоставлять MessageCreator. Вместо этого вы передаете объект, который должен быть отправлен напрямую, в convertAndSend(), и перед отправкой объект будет преобразован в Message.

Например, следующая переопределение sendOrder() использует convertAndSend() для отправки Order в заданный пункт назначения:

*@Override*

*public void sendOrder(Order order) {*

*jms.convertAndSend("tacocloud.order.queue", order);*

*}*

Как и метод send(), метод convertAndSend () примет значение Destination или String, чтобы указать адресата, или вы можете вообще не указывать адресата, чтобы отправить сообщение в пункт назначения по умолчанию.

Какую бы реализацию convertAndSend() вы ни выбрали, Order, переданный в convertAndSend(), перед отправкой преобразуется в Message. Под капотом это достигается с помощью реализации MessageConverter, которая выполняет грязную работу по преобразованию объектов в Message -ы.

НАСТРОЙКА КОНВЕРТЕРА СООБЩЕНИЙ

MessageConverter - это Spring-определнный интерфейс, который имеет только два метода для реализации:

*public interface MessageConverter {*

*Message toMessage(Object object, Session session)*

*throws JMSException, MessageConversionException;*

*Object fromMessage(Message message)*

*}*

Хотя этот интерфейс достаточно прост для реализации, вам часто не нужно создавать собственную реализацию. Spring уже предлагает несколько реализаций, таких как описанные в таблице 8.3.

**Таблица 8.3. Spring message конвертеры для общих задач преобразования (все в пакете org.springframework.jms.support.converter)**

Message конвертер - Что он делает

*MappingJackson2MessageConverter* - Использует Jackson 2 JSON библиотеку конвертирования сообщений в и из JSON

*MarshallingMessageConverter*- Использует JAXB для конвертирования сообщений в и из XML

*MessagingMessageConverter*- Конвертирует Message из абстракции в и из Message используя базовый MessageConverter для полезной нагрузки и JmsHeaderMapper для сопоставления заголовков JMS в и из стандартных заголовков сообщений

*SimpleMessageConverter* - Преобразует String в и из TextMessage, байтовые массивы в и из BytesMessage, Maps в и из MapMessage, и Serializable объекты в и из ObjectMessage.

SimpleMessageConverter является значением по умолчанию, но требует, чтобы отправляемый объект реализовывал Serializable. Это может быть хорошей идеей, но вы можете предпочесть использовать один из других конвертеров сообщений, например MappingJackson2MessageConverter, чтобы избежать этого ограничения.

Чтобы применить другой конвертер сообщений, все, что вам нужно сделать, - это объявить экземпляр выбранного конвертера как bean-компонент. Например, следующее объявление компонента позволит использовать MappingJackson2MessageConverter вместо SimpleMessageConverter:

*@Bean*

*public MappingJackson2MessageConverter messageConverter() {*

*MappingJackson2MessageConverter messageConverter =*

*new MappingJackson2MessageConverter();*

*messageConverter.setTypeIdPropertyName("\_typeId");*

*return messageConverter;*

*}*

Обратите внимание, что вы вызвали setTypeIdPropertyName() для MappingJackson2MessageConverter, прежде чем возвращать его. Это очень важно, так как позволяет получателю знать, в какой тип конвертировать входящее сообщение. По умолчанию он будет содержать полное имя класса конвертируемого типа. Но это несколько негибко, требуя, чтобы получатель также имел тот же тип, с тем же полностью определенным именем класса.

Чтобы обеспечить большую гибкость, вы можете сопоставить синтетическое имя типа с реальным типом, вызвав setTypeIdMappings() в конвертере сообщений. Например, следующее изменение метода bean-объекта конвертера сообщений отображает синтетический идентификатор типа order в класс Order:

*@Bean*

*public MappingJackson2MessageConverter messageConverter() {*

*MappingJackson2MessageConverter messageConverter =*

*new MappingJackson2MessageConverter();*

*messageConverter.setTypeIdPropertyName("\_typeId");*

*Map<String, Class<?>> typeIdMappings = new HashMap<String, Class<?>>();*

*typeIdMappings.put("order", Order.class);*

*messageConverter.setTypeIdMappings(typeIdMappings);*

*return messageConverter;*

*}*

Вместо того, чтобы полное имя класса отправлялось в свойстве \_typeId сообщения, будет отправлено значение order. В принимающем приложении будет сконфигурирован аналогичный конвертер сообщений, отображающий order в собственном понимании order-а. Эта реализация order может быть в другом пакете, иметь другое имя и даже иметь подмножество свойств Order отправителя.

POST-ОБРАБОТКА СООБЩЕНИЙ

Предположим, что в дополнение к своему прибыльному веб-бизнесу Taco Cloud решила открыть несколько магазинов в строительном секторе. Учитывая, что любой из их ресторанов также может быть центром для веб-бизнеса, им нужен способ сообщить источник заказа на кухню в ресторан. Это позволит кухонному персоналу использовать другой процесс для заказов в магазине, нежели для заказов через Интернет.

Было бы разумно добавить новое свойство источника в объект Order для переноса этой информации, установв его как WEB для заказов, размещенных в Интернете, и STORE для заказов, размещенных в магазинах. Но для этого потребуется изменить как класс Order на веб-сайте, так и класс Order для кухонного приложения, когда на самом деле это информация, которая требуется только для составителей тако.

Более простым решением было бы добавить пользовательский заголовок к сообщению для установки источника заказа. Если бы вы использовали метод send() для отправки тако-заказов, это можно легко сделать, вызвав setStringProperty() для объекта Message:

*jms.send("tacocloud.order.queue",*

*session -> {*

*Message message = session.createObjectMessage(order);*

*message.setStringProperty("X\_ORDER\_SOURCE", "WEB");*

*});*

Проблема в том, что вы не используете send(). При выборе использования convertAndSend() объект Message создается в обертке, и у вас нет к нему доступа.

К счастью, есть способ настроить Message, созданное в обертке, до его отправки. Передав MessagePostProcessor в качестве последнего параметра для convertAndSend(), вы можете делать с сообщением все, что захотите, после его создания. Следующий код по-прежнему использует convertAndSend(), но использует MessagePostProcessor для добавления заголовка X\_ORDER\_SOURCE до отправки сообщения:

*jms.convertAndSend("tacocloud.order.queue", order, new MessagePostProcessor() {*

*@Override*

*public Message postProcessMessage(Message message) throws JMSException {*

*message.setStringProperty("X\_ORDER\_SOURCE", "WEB");*

*return message;*

*}*

*});*

Возможно, вы заметили, что MessagePostProcessor - это функциональный интерфейс. Это означает, что вы можете немного упростить его, заменив анонимный внутренний класс лямбда-выражением:

*jms.convertAndSend("tacocloud.order.queue", order,*

*message -> {*

*message.setStringProperty("X\_ORDER\_SOURCE", "WEB");*

*return message;*

*});*

Хотя вам нужен только этот конкретный MessagePostProcessor для этого одного вызова метода convertAndSend(), вы можете использовать один и тот же MessagePostProcessor для нескольких различных вызовов convertAndSend(). В этих случаях, возможно, ссылка на метод является лучшим выбором, чем лямбда, во избежание ненужного дублирования кода:

*@GetMapping("/convertAndSend/order")*

*public String convertAndSendOrder() {*

*Order order = buildOrder();*

*jms.convertAndSend("tacocloud.order.queue", order,*

*this::addOrderSource);*

*return "Convert and sent order";*

*}*

*private Message addOrderSource(Message message) throws JMSException {*

*message.setStringProperty("X\_ORDER\_SOURCE", "WEB");*

*return message;*

*}*

Теперь вы видели несколько способов отправки сообщений. Но не стоит отправлять сообщение, если никто его не получит. Давайте посмотрим, как вы можете получать сообщения с помощью Spring и JMS.

8.1.3 Получение JMS сообщений

Когда дело доходит до получения сообщений, у вас есть выбор pull (получения) модель, где ваш код запрашивает сообщение и ожидает его получение, или push (проталкивания) модель, в которой сообщения передаются в ваш код по мере их поступления.

JmsTemplate предлагает несколько методов для получения сообщений, но все они используют pull модель. Вы вызываете один из этих методов для запроса сообщения, и поток блокируется до тех пор, пока сообщение не станет доступным (что может быть немедленно или может занять некоторое время).

С другой стороны, у вас также есть возможность использовать push-модель, в которой вы определяете слушатель (listener) сообщений, который вызывается каждый раз, когда сообщение доступно.

Оба варианта подходят для различных вариантов использования. Общепринято, что push-модель - лучший выбор, так как она не блокирует поток. Но в некоторых случаях использование слушателя  (listener) может быть перегружено, если сообщения приходят слишком быстро. Pull (извлечения) модель позволяет потребителю заявить, что он готов обработать новое сообщение.

Давайте рассмотрим оба способа получения сообщений. Начнем с pull модели, предлагаемой JmsTemplate.

**ПОЛУЧЕНИЕ С JmsTemplate**

JmsTemplate предлагает несколько методов для методов извлечения из брокера, включая следующие:

*Message receive() throws JmsException;*

*Message receive(Destination destination) throws JmsException;*

*Message receive(String destinationName) throws JmsException;*

*Object receiveAndConvert() throws JmsException;*

*Object receiveAndConvert(Destination destination) throws JmsException;*

*Object receiveAndConvert(String destinationName) throws JmsException;*

Как видите, эти шесть методов отражение методов send() и convertAndSend () из JmsTemplate. Методы receive() получают необработанное Message,  а методы receiveAndConvert() используют сконфигурированный конвертер сообщений для преобразования сообщений в типы доменов. И для каждого из них вы можете указать либо Destination, либо String, содержащую имя пункта назначения, или вы можете получить сообщение из пункта назначения по умолчанию.

Чтобы увидеть их в действии, вы напишите некоторый код, который извлекает Order из пункта назначения tacocloud.order.queue. В следующем листинге показан OrderReceiver, компонент службы, который получает данные заказа с помощью JmsTemplate.receive().

**Листинг 8.2. Получение заказов из очереди**

*package tacos.kitchen.messaging.jms;*

*import javax.jms.Message;*

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.jms.core.JmsTemplate;*

*import org.springframework.jms.support.converter.MessageConverter;*

*import org.springframework.stereotype.Component;*

*@Component*

*public class JmsOrderReceiver implements OrderReceiver {*

*private JmsTemplate jms;*

*private MessageConverter converter;*

*@Autowired*

*public JmsOrderReceiver(JmsTemplate jms, MessageConverter converter) {*

*this.jms = jms;*

*this.converter = converter;*

*}*

*public Order receiveOrder() {*

*Message message = jms.receive("tacocloud.order.queue");*

*return (Order) converter.fromMessage(message);*

*}*

*}*

Здесь вы использовали String, чтобы указать пункт назначения для получения заказа. Метод receive() возвращает не конвертированный Message. Нам нужен Order, который находится внутри, Message, поэтому следующее, что происходит, - это использование конвертера сообщения для его преобразования. Свойство type ID в сообщении поможет конвертеру преобразовать его в Order, но ответ будет в виде Object, который придется  привести к нужному типу перед тем, как вы сможете его вернуть.

Получение необработанного объекта Message может быть полезно в некоторых случаях, когда вам необходимо проверить свойства и заголовки сообщения. Но часто вам нужна только полезная нагрузка. Преобразование этой полезной нагрузки в тип домена является двухэтапным процессом и требует, чтобы преобразователь сообщений был внедрен в компонент. Когда вы заботитесь только о полезной нагрузке сообщения, использовать receiveAndConvert() будет намного проще. В следующем листинге показано, как JmsOrderReceiver может быть переработан для использования receiveAndConvert () вместо receive().

**Листинг 8.3 Получение преобразованного объекта Order**

*package tacos.kitchen.messaging.jms;*

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.jms.core.JmsTemplate;*

*import org.springframework.stereotype.Component;*

*@Component*

*public class JmsOrderReceiver implements OrderReceiver {*

*private JmsTemplate jms;*

*@Autowired*

*public JmsOrderReceiver(JmsTemplate jms) {*

*this.jms = jms;*

*}*

*public Order receiveOrder() {*

*return (Order) jms.receiveAndConvert("tacocloud.order.queue");*

*}*

*}*

Эта новая версия JmsOrderReceiver имеет метод receieveOrder (), который был сокращен до одной строки. И вам больше не нужно внедрять MessageConverter, потому что все преобразования сообщений будут выполняться под капотом в receiveAndConvert ().

Прежде чем двигаться дальше, давайте рассмотрим, как receiveOrder() может быть использован в кухонном приложении Taco Cloud. Повар на одной из кухонь Taco Cloud может нажать кнопку или предпринять какие-либо действия, чтобы указать, что они готовы начать создание тако. В этот момент будет вызван receiveOrder(), и  вызов метода receive() или receiveAndConvert() будет заблокирован. Больше ничего не произойдет, пока сообщение о заказе не будет готово. Как только заказ поступит, он будет возвращен функцией receiveOrder(), и его информация будет использоваться для отображения деталей заказа, чтобы повар мог приступить к работе. Это кажется естественным выбором для модели тянуть (pull).

Теперь давайте посмотрим, как работает push-модель, объявив JMS-слушатель.

**ОБЪЯВЛЕНИЕ ЛИСТЕНЕРА СООБЩЕНИЙ**

В отличие от модели извлечения (pull), где для получения сообщения требовался явный вызов метода receive() или receiveAndConvert(), листенер сообщений является пассивным компонентом, который простаивает до получения сообщения.

Чтобы создать листенер сообщений, который реагирует на сообщения JMS, вы просто должны аннотировать метод в компоненте с помощью @JmsListener. В следующем листинге показан новый компонент OrderListener, который пассивно прослушивает сообщения, а не активно запрашивает их.

**Листинг 8.4. Компонент OrderListener, который прослушивает заказы**

*package tacos.kitchen.messaging.jms.listener;*

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.jms.annotation.JmsListener;*

*import org.springframework.stereotype.Component;*

*@Component*

*public class OrderListener {*

*private KitchenUI ui;*

*@Autowired*

*public OrderListener(KitchenUI ui) {*

*this.ui = ui;*

*}*

*@JmsListener(destination = "tacocloud.order.queue")*

*public void receiveOrder(Order order) {*

*ui.displayOrder(order);*

*}*

*}*

Метод receiveOrder () аннотирован JmsListener для «прослушивания» сообщений в месте назначения tacocloud.order.queue. Он не имеет отношения к JmsTemplate и не вызывается явно кодом вашего приложения. Вместо этого, каркасный код в Spring ожидает поступления сообщений в указанный пункт назначения, и когда они приходят, метод receiveOrder() вызывается автоматически с полезной нагрузкой Order в качестве параметра.

Во многих отношениях аннотация @JmsListener похожа на одну из аннотаций сопоставления запросов Spring MVC, например, @GetMapping или @PostMapping. В Spring MVC методы, аннотированные одним из методов отображения запросов, реагируют на запросы по указанному пути. Точно так же методы, аннотированные @JmsListener, реагируют на сообщения, поступающие в пункт назначения.

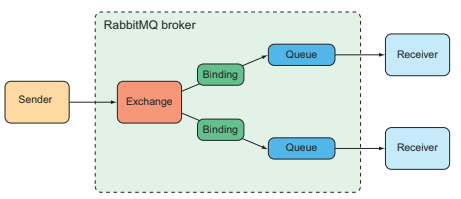
Слушатели сообщений часто советуются как лучший выбор, потому что они не блокируют и могут быстро обрабатывать несколько сообщений. Однако в контексте приложения Taco Cloud они, возможно, не лучший выбор. Повара являются существенным узким местом в системе и могут не иметь возможности готовить тако так быстро, как поступают заказы. Повар может наполовину выполнить заказ, когда на экране отображается новый заказ. Пользовательский интерфейс кухни должен был бы буферизовать заказы по мере их поступления, чтобы не перегружать кухонный персонал.

Нельзя сказать, что слушатели сообщений плохие. Напротив, они идеально подходят для быстрой обработки сообщений. Но когда обработчики сообщений должны иметь возможность запрашивать бОльшую часть сообщений в свое время, модель извлечения, предлагаемая JmsTemplate, кажется более подходящей.

Поскольку JMS определяется стандартной спецификацией Java и поддерживается многими реализациями брокера сообщений, это обычный выбор для обмена сообщениями в Java. Но у JMS есть несколько недостатков, не последним из которых является то, что в качестве спецификации Java его использование ограничено приложениями Java. Новые опции обмена сообщениями, такие как RabbitMQ и Kafka, устраняют эти недостатки и доступны для других языков и платформ, помимо JVM. Давайте отложим JMS и посмотрим, как вы могли бы реализовать обмен сообщениями о заказах тако с RabbitMQ.

8.2 Работа с RabbitMQ и AMQP

Как, возможно, самая известная реализация AMQP, RabbitMQ предлагает более продвинутую стратегию маршрутизации сообщений, чем JMS. Принимая во внимание, что сообщения JMS адресуются с именем пункта назначения, из которого получатель получит их, сообщения AMQP адресуются с именем обмена и ключом маршрутизации, которые отделены от очереди, которую прослушивает получатель. Эта связь между обменом и очередями показана на рисунке 8.1.



**Рис. 8.1. Сообщения, отправляемые на обмен RabbitMQ, направляются в одну или несколько очередей на основе ключей маршрутизации и привязок.**

Когда сообщение приходит к брокеру RabbitMQ, оно отправляется на тот обмен (exchange), для которого оно было адресовано. Exchange отвечает за маршрутизацию сообщения в одну или несколько очередей в зависимости от типа exchange, привязки между exchange и очередями (queue) и значения ключа маршрутизации сообщения.

Существует несколько видов обмена, включая следующие:

*Default*— обмен, который автоматически создается брокером. Он направляет сообщения в очереди, чье имя совпадает с ключом маршрутизации сообщения. Все очереди по умолчанию привязаны к default exchange.

*Direct*- направляет сообщения в очередь, ключ привязки которой совпадает с ключом маршрутизации сообщения.

*Topic*- направляет сообщение в одну или несколько очередей, в которых ключ привязки (который может содержать символы подстановки) соответствует ключу маршрутизации сообщения.

*Fanout*- маршрутизирует сообщения во все связанные очереди без учета ключей привязки или ключей маршрутизации.

*Headers*- аналогичны обмену topic, за исключением того, что маршрутизация основана на значениях заголовков сообщений, а не на ключах маршрутизации.

*Dead letter* - универсальное сообщение для всех сообщений, которые невозможно доставить (это означает, что они не соответствуют какой-либо определенной привязке обмена к очереди).

Простейшие формы обмена это Default и Fanout, так как они примерно соответствуют очереди и теме JMS. Но другие типы обмена позволяют задать более гибкие схемы маршрутизации.

Самая важная вещь для понимания - это то, что сообщения отправляются на обменники (exchanges) с ключами маршрутизации, и они берутся из очередей. То, как они попадают из обмена в очередь, зависит от определений привязки и того, что лучше всего подходит для ваших сценариев использования.

Какой тип обмена вы используете и как вы определяете привязки между обменами и очередями, мало влияет на то, как сообщения отправляются и принимаются в ваших приложениях Spring. Поэтому мы сосредоточимся на том, как написать код, который отправляет и получает сообщения с помощью Rabbit.

П р и м е ч а н и е - Более подробное обсуждение того, как лучше связать очереди с обменниками , см. RabbitMQ in Action Alvaro Videla and Jason J.W. Williams (Manning, 2012).

8.2.1 Добавление RabbitMQ в Spring

Прежде чем вы сможете начать отправку и получение сообщений RabbitMQ с помощью Spring, вам нужно добавить в свою сборку зависимость AMQP-стартера Spring Boot вместо Artemis или ActiveMQ-стартера, которые вы добавили в предыдущем разделе:

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-amqp</artifactId>*

*</dependency>*

Добавление стартера AMQP в вашу сборку вызовет автоконфигурацию, которая создаст фабрику соединений AMQP и компоненты RabbitTemplate, а также другие вспомогательные компоненты. Просто добавьте эту зависимость, чтобы начать отправку и получение сообщений от брокера RabbitMQ с помощью Spring. Но есть несколько полезных свойств, о которых вы захотите узнать, перечисленных в таблице 8.4.

**Таблица 8.4 Свойства для настройки местоположения и учетных данных RabbitMQ брокера**

*spring.rabbitmq.addresses*- Разделенный запятыми список адресов брокера RabbitMQ

*spring.rabbitmq.host* - Хост брокера (по умолчанию localhost)

*spring.rabbitmq.port*- Порт брокера (по умолчанию 5672)

*spring.rabbitmq.username*- Имя пользователя для доступа к брокеру (необязательно)

*spring.rabbitmq.password*- Пароль пользователя для доступа к брокеру (необязательно)

В целях разработки у вас, вероятно, будет брокер RabbitMQ, для которого не требуется проверка подлинности на вашем локальном компьютере с слушателем порта 5672. Эти свойства, скорее всего, не будут особенно полезны, пока вы еще в разработке, но они без сомнения, окажутся полезными, когда ваши приложения будут запущены в продакшен.

Например, предположим, что при переходе в продакшен ваш брокер RabbitMQ находится на сервере с именем rabbit.tacocloud.com, прослушивает порт 5673 и требует учетных данных. В этом случае следующая конфигурация в вашем файле application.yml установит эти свойства, когда профиль prod активен:

*spring:*

*profiles: prod*

*rabbitmq:*

*host: rabbit.tacocloud.com*

*port: 5673*

*username: tacoweb*

*password: l3tm31n*

Теперь, когда RabbitMQ настроен в вашем приложении, пришло время начать отправку сообщений с RabbitTemplate.

8.2.2 Отправка сообщений с RabbitTemplate

В основе поддержки обмена сообщениями RabbitMQ от Spring лежит RabbitTemplate. RabbitTemplate похож на JmsTemplate, предлагая аналогичный набор методов. Однако, как вы увидите, есть некоторые тонкие различия, которые соответствуют уникальному способу работы RabbitMQ.

Что касается отправки сообщений с помощью RabbitTemplate, методы send() и convertAndSend() аналогичны методам с такими же именами из JmsTemplate. Но в отличие от методов JmsTemplate, которые направляют сообщения только в определенную очередь или тему, методы RabbitTemplate отправляют сообщения в  с точки зрения обмена и ключей маршрутизации. Вот несколько наиболее важных методов отправки сообщений с RabbitTemplate (эти методы определены в AmqpTemplate, интерфейсе, реализованном RabbitTemplate.):

// Отправка необработанных сообщений

*void send(Message message) throws AmqpException;*

*void send(String routingKey, Message message) throws AmqpException;*

*void send(String exchange, String routingKey, Message message) throws AmqpException;*

// Отправлять сообщения, преобразованные из объектов

*void convertAndSend(Object message) throws AmqpException;*

*void convertAndSend(String routingKey, Object message) throws AmqpException;*

*void convertAndSend(String exchange, String routingKey, Object message) throws AmqpException;*

// Отправка сообщений, преобразованных из объектов с последующей обработкой (post-processing)

*void convertAndSend(Object message, MessagePostProcessor mPP) throws AmqpException;*

*void convertAndSend(String routingKey, Object message, MessagePostProcessor messagePostProcessor) throws AmqpException;*

*void convertAndSend(String exchange, String routingKey, Object message, MessagePostProcessor messagePostProcessor) throws AmqpException;*

Как видите, эти методы следуют шаблону, аналогичному их близнецам в JmsTemplate. Первые три метода send() отправляют необработанный объект Message. Следующие три метода convertAndSend() принимают объект, который будет преобразован в Message за кулисами перед отправкой. Последние три метода convertAndSend() похожи на предыдущие три, но они принимают MessagePostProcessor, который можно использовать для манипулирования объектом Message до его отправки брокеру.

Эти методы отличаются от своих аналогов JmsTemplate тем, что они принимают значения String, чтобы указать ключ обмена и маршрутизации, а не имя назначения (или Destination назначения). Методы, которые не принимают exchange, будут отправлять свои сообщения в exchange по умолчанию. Аналогично, методы, которые не принимают ключ маршрутизации, будут маршрутизировать свои сообщения с ключом маршрутизации по умолчанию.

Давайте включим RabbitTemplate для работы с отправкой тако-заказов. Один из способов сделать это - использовать метод send(), как показано в листинге 8.5. Но прежде чем вы сможете вызвать send(), вам нужно преобразовать объект Order в a Message. Это может быть утомительной работой, если бы не тот факт, что RabbitTemplate делает свой конвертер сообщений доступным с помощью метода getMessageConverter().

**Листинг 8.5 Отправка сообщений с RabbitTemplate.send()**

*package tacos.messaging;*

*import org.springframework.amqp.core.Message;*

*import org.springframework.amqp.core.MessageProperties;*

*import org.springframework.amqp.rabbit.core.RabbitTemplate;*

*import org.springframework.amqp.support.converter.MessageConverter;*

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.stereotype.Service;*

*import tacos.Order;*

*@Service*

*public class RabbitOrderMessagingService implements OrderMessagingService {*

*private RabbitTemplate rabbit;*

*@Autowired*

*public RabbitOrderMessagingService(RabbitTemplate rabbit) {*

*this.rabbit = rabbit;*

*}*

*public void sendOrder(Order order) {*

*MessageConverter converter = rabbit.getMessageConverter();*

*MessageProperties props = new MessageProperties();*

*Message message = converter.toMessage(order, props);*

*rabbit.send("tacocloud.order", message);*

*}*

*}*

Если у вас есть MessageConverter, конвертировать Order в Message становится просто. Вы должны задать любые свойства сообщения с MessageProperties, но если вам не нужно устанавливать какие-либо такие свойства, то подойдет экземпляр MessageProperties по умолчанию. Затем все, что осталось - это вызвать send(), передав ключ обмена и маршрутизации (оба из которых являются необязательными) вместе с сообщением. В этом примере вы указываете только ключ маршрутизации - tacocloud.order - вместе с сообщением, поэтому будет использоваться exchange (обмен) по умолчанию.

Говоря об default exchange, именем обмена по умолчанию является «» (пустая строка), что соответствует default exchange, автоматически создаваемому брокером RabbitMQ. Точно так же ключом маршрутизации по умолчанию является «» (маршрутизация которого зависит от рассматриваемого обмена и привязок (exchange and binding)). Вы можете переопределить эти значения по умолчанию, установив свойства spring.rabbitmq.template.exchange и spring.rabbitmq.template.routing-key:

*spring:*

*rabbitmq:*

*template:*

*exchange: tacocloud.orders*

*routing-key: kitchens.central*

В этом случае все сообщения, отправленные без указания exchange, будут автоматически отправлены на exchange, имя которого - tacocloud.orders. Если ключ маршрутизации также не указан в вызове send() или convertAndSend(), у сообщений будет ключ маршрутизации kitchens.central.

Создать объект Message из конвертера сообщений достаточно просто, но еще проще использовать convertAndSend(), чтобы RabbitTemplate обрабатывал всю работу по преобразованию за вас:

*public void sendOrder(Order order) {*

*rabbit.convertAndSend("tacocloud.order", order);*

*}*

**НАСТРОЙКА ПРЕОБРАЗОВАТЕЛЯ СООБЩЕНИЙ**

По умолчанию преобразование сообщений выполняется с помощью SimpleMessageConverter, который может преобразовывать простые типы (например, String) и объекты Serializable в объекты Message. Но Spring предлагает несколько конвертеров сообщений для RabbitTemplate, включая следующие:

*Jackson2JsonMessageConverter*—Ковертирует объекты В и ИЗ JSON используя  Jackson 2 JSON processor

*MarshallingMessageConverter*—Конвертирует используя Spring Marshaller и Unmarshaller

*SerializerMessageConverter*—Ковертирует  String aи нативные объекты любого типа используя Spring-овый Serializer и Deserializer абстракции

*SimpleMessageConverter*- преобразует строки, байтовые массивы и Serializable типы.

*ContentTypeDelegatingMessageConverter*- делегирует другому MessageConverter на основе заголовка contentType.

*MessagingMessageConverter*- делегирует базовый MessageConverter для преобразования сообщений и AmqpHeaderConverter для заголовков.

Если вам нужно изменить конвертер сообщений, все, что вам нужно сделать, это настроить bean-компонент типа MessageConverter. Например, для преобразования сообщений на основе JSON вы можете настроить конвертер Jackson2JsonMessageConver следующим образом:

*@Bean*

*public MessageConverter messageConverter() {*

*return new Jackson2JsonMessageConverter();*

*}*

Автоконфигурация Spring Boot обнаружит этот bean-компонент и вставит его в RabbitTemplate вместо конвертера сообщений по умолчанию.

**НАСТРОЙКА СВОЙСТВ СООБЩЕНИЙ**

Как и в случае с JMS, вам может потребоваться установить некоторые заголовки в отправляемых вами сообщениях. Например, допустим, вам нужно отправить X\_ORDER\_SOURCE для всех заказов, представленных через веб-сайт Taco Cloud. При создании ваших собственных объектов Message вы можете установить заголовок через экземпляр MessageProperties, который вы передаете конвертеру сообщений. Возвращаясь к методу sendOrder() из листинга 8.5, вам понадобится всего одна дополнительная строка кода для установки заголовка:

*public void sendOrder(Order order) {*

*MessageConverter converter = rabbit.getMessageConverter();*

*MessageProperties props = new MessageProperties();*

*props.setHeader("X\_ORDER\_SOURCE", "WEB");*

*Message message = converter.toMessage(order, props);*

*rabbit.send("tacocloud.order", message);*

*}*

Однако при использовании convertAndSend() у вас нет быстрого доступа к объекту MessageProperties. В этом вам может помочь MessagePostProcessor:

*@Override*

*public void sendOrder(Order order) {*

*rabbit.convertAndSend("tacocloud.order.queue", order,*

*new MessagePostProcessor() {*

*@Override*

*public Message postProcessMessage(Message message)*

*throws AmqpException {*

*MessageProperties props = message.getMessageProperties();*

*props.setHeader("X\_ORDER\_SOURCE", "WEB");*

*return message;*

*}*

*});*

*}*

Здесь вы предоставляете convertAndSend() анонимной внутренней реализацией MessagePostProcessor. В методе postProcessMessage() вы извлекаете свойства MessageProperties из сообщения и затем вызываете setHeader(), чтобы установить заголовок X\_ORDER\_SOURCE.

Теперь, когда вы увидели, как отправлять сообщения с помощью RabbitTemplate, давайте переключимся на код, который получает сообщения из очереди RabbitMQ.

8.2.3 Получение сообщения от RabbitMQ

Вы видели, что отправка сообщений с помощью RabbitTemplate не сильно отличается от отправки сообщений с помощью JmsTemplate. И, как выясннится, получение сообщений из очереди RabbitMQ не сильно отличается от JMS.

Как и в случае с JMS, у вас есть два варианта:

-Получение сообщений из очереди с помощью RabbitTemplate

-Отправка сообщений в методе, аннотированным @RabbitListener.

Давайте начнем с рассмотрения pull-овского  метода RabbitTemplate.receive(), основанного на извлечении.

**ПОЛУЧЕНИЕ СООБЩЕНИЙ С RABBITTEMPLATE**

RabbitTemplate поставляется с несколькими методами для извлечения сообщений из очереди. Несколько наиболее полезных из них перечислены здесь:

// Получение сообщений

*Message receive() throws AmqpException;*

*Message receive(String queueName) throws AmqpException;*

*Message receive(long timeoutMillis) throws AmqpException;*

*Message receive(String queueName, long timeoutMillis) throws AmqpException;*

// Получать объекты, преобразованные из сообщений

*Object receiveAndConvert() throws AmqpException;*

*Object receiveAndConvert(String queueName) throws AmqpException;*

*Object receiveAndConvert(long timeoutMillis) throws AmqpException;*

*Object receiveAndConvert(String queueName, long timeoutMillis) throws*

*AmqpException;*

// Получать типобезопасные объекты, преобразованные из сообщений

*<T> T receiveAndConvert(ParameterizedTypeReference<T> type) throws*

*AmqpException;*

*<T> T receiveAndConvert(String queueName, ParameterizedTypeReference<T> type)*

*throws AmqpException;*

*<T> T receiveAndConvert(long timeoutMillis, ParameterizedTypeReference<T>*

*type) throws AmqpException;*

*<T> T receiveAndConvert(String queueName, long timeoutMillis,*

*ParameterizedTypeReference<T> type) throws AmqpException;*

Эти методы являются зеркальными изображениями методов send() и convertAndSend(), описанных ранее. Принимая во внимание, что send() используется для отправки необработанных объектов Message, а receive() получает необработанные объекты Message из очереди. Аналогично, receiveAndConvert() получает сообщения и использует конвертер сообщений для преобразования их в объекты домена перед их возвратом.

Но есть несколько очевидных различий в сигнатурах методов. Во-первых, ни один из этих методов не принимает ключ обмена или маршрутизации в качестве параметра. Это связано с тем, что обмены и ключи маршрутизации используются для маршрутизации сообщений в очереди, но как только сообщения находятся в очереди, их следующим пунктом назначения является потребитель, который вытаскивает их из очереди. Приложения-потребители не должны заниматься обменом или маршрутизацией ключей. Очередь - это единственное, что нужно знать приложениям-потребителям.

Вы также заметите, что многие методы принимают long параметр, указывающий тайм-аут для получения сообщений. По умолчанию время ожидания приема составляет 0 миллисекунд. То есть вызов метода receive() будет возвращен немедленно, потенциально с null значением, если сообщения недоступны. Это явное отличие от того, как ведут себя методы receive() в JmsTemplate. Передав значение тайм-аута, методы  receive() и receiveAndConvert() блокируются до прибытия сообщения или до истечения времени ожидания. Но даже при ненулевом таймауте ваш код должен быть готов к null возврату.

Давайте посмотрим, как вы можете применить это на практике. Следующий листинг показывает новую реализацию OrderReceiver на основе Rabbit, которая использует RabbitTemplate для получения заказов.

**Листинг 8.6. Получение заказов от RabbitMQ с помощью RabbitTemplate**

*package tacos.kitchen.messaging.rabbit;*

*import org.springframework.amqp.core.Message;*

*import org.springframework.amqp.rabbit.core.RabbitTemplate;*

*import org.springframework.amqp.support.converter.MessageConverter;*

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.stereotype.Component;*

*@Component*

*public class RabbitOrderReceiver {*

*private RabbitTemplate rabbit;*

*private MessageConverter converter;*

*@Autowired*

*public RabbitOrderReceiver(RabbitTemplate rabbit) {*

*this.rabbit = rabbit;*

*this.converter = rabbit.getMessageConverter();*

*}*

*public Order receiveOrder() {*

*Message message = rabbit.receive("tacocloud.orders");*

*return message != null ? (Order) converter.fromMessage(message) : null;*

*}*

*}*

Метод receiveOrder() - это место, где происходит все действие. Он вызывает метод receive() на внедренном шаблоне RabbitTemplate для получения заказа из очереди tacocloud.orders. Он не предоставляет значения времени ожидания, поэтому вы можете только предполагать, что вызов немедленно возвращается либо с Message, либо с null значением. Если Message возвращается, вы используете MessageConverter из шаблона RabbitTemplate для преобразования Message в Order. Иначе, если receive() возвращает null, вы возвращаете null.

В зависимости от варианта использования вы можете установить небольшую задержку. Например, на верхнем дисплее кухни Taco Cloud вы можете подождать некоторое время, если нет заказов. Допустим, вы решили подождать до 30 секунд, прежде чем сдаться. Метод receiveOrder() может быть изменен для передачи 30 000 миллисекундной задержки для receive():

*public Order receiveOrder() {*

*Message message = rabbit.receive("tacocloud.order.queue", 30000);*

*return message != null ? (Order) converter.fromMessage(message) : null;*

*}*

Если вы похожи на меня, то, увидев такой захардкоженный элемент, вы почувствуете дискомфорт. Вы можете подумать, что было бы неплохо создать аннотированный класс @ConfigurationProperties, чтобы можно было настроить это время ожидания с помощью свойства конфигурации Spring Boot. Я бы согласился с вами, если бы не тот факт, что Spring Boot уже предлагает такое свойство конфигурации. Если вы хотите установить время ожидания с помощью конфигурации, просто удалите значение времени ожидания в вызове receive() и установите его в своей конфигурации с помощью свойства spring.rabbitmq.template.receive-timeout:

*spring:*

*rabbitmq:*

*template:*

*receive-timeout: 30000*

Вернемся к методу receiveOrder(), обратите внимание, что вам пришлось использовать конвертер сообщений из RabbitTemplate для преобразования входящего объекта Message в объект Order. Но если RabbitTemplate использует конвертер сообщений, почему он не может выполнить преобразование для вас? Именно для этого и предназначен метод receiveAndConvert(). Используя receiveAndConvert(), вы можете переписать receiveOrder() следующим образом:

*public Order receiveOrder() {*

*return (Order) rabbit.receiveAndConvert("tacocloud.order.queue");*

*}*

Это намного проще, не так ли? Единственная тревожная вещь, которую я вижу, это приведение Object к Order. Однако есть альтернатива. Вместо этого вы можете передать ParameterizedTypeReference в receiveAndConvert() для непосредственного получения объекта Order:

*public Order receiveOrder() {*

*return rabbit.receiveAndConvert("tacocloud.order.queue", new ParameterizedTypeReference<Order>() {});*

*}*

Спорный вопрос, чем это лучше, но это более типобезопасный подход. Единственное требование для использования ParameterizedTypeReference с receiveAndConvert() состоит в том, что конвертер сообщений должен быть реализацией SmartMessageConverter; Jackson2JsonMessageConverter - единственная готовая реализация на выбор.

Модель pull, предлагаемая JmsTemplate, подходит для многих случаев использования, но часто лучше иметь код, который прослушивает сообщения и который вызывается при поступлении сообщений. Давайте посмотрим, как вы можете создавать управляемые сообщениями bean-компоненты, которые отвечают на сообщения RabbitMQ.

**ОБРАБОТКА СООБЩЕНИЙ RABBITMQ СО СЛУШАТЕЛЯМИ**

Для управляемых сообщениями компонентов RabbitMQ Spring предлагает RabbitListener, аналог RabbitMQ для JmsListener. Чтобы указать, что метод должен вызываться при поступлении сообщения в очередь RabbitMQ, аннотируйте метод компонента с помощью @RabbitTemplate.

Например, в следующем листинге показана реализация OrderReceiver в RabbitMQ, аннотированная для прослушивания сообщений заказа, а не для их запроса с помощью RabbitTemplate.

**Листинг 8.7 объявление метода в качестве листенера сообщений RabbitMQ**

*package tacos.kitchen.messaging.rabbit.listener;*

*import org.springframework.amqp.rabbit.annotation.RabbitListener;*

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.stereotype.Component;*

*@Component*

*public class OrderListener {*

*private KitchenUI ui;*

*@Autowired*

*public OrderListener(KitchenUI ui) {*

*this.ui = ui;*

*}*

*@RabbitListener(queues = "tacocloud.order.queue")*

*public void receiveOrder(Order order) {*

*ui.displayOrder(order);*

*}*

*}*

Вы без сомнения заметите, что это очень похоже на код из листинга 8.4. Действительно, единственное, что изменилось, это аннотация слушателя - вместо @JmsListener используется @RabbitListener. Каким бы замечательным ни был @RabbitListener, это почти дублирование кода оставляет мне мало что сказать о @RabbitListener, чего я еще не говорил о @JmsListener. Они оба отлично подходят для написания кода, который отвечает на сообщения, отправляемые им соответствующими брокерами - брокером JMS для @JmsListener и брокером RabbitMQ для @RabbitListener.

Хотя вы можете чувствовать отсутствие энтузиазма в отношении @RabbitListener в предыдущем абзаце, будьте уверены, что я не специально. По правде говоря, тот факт, что @RabbitListener работает так же, как @JmsListener, на самом деле очень интересен! Это означает, что вам не нужно изучать совершенно другую модель программирования при работе с RabbitMQ против Artemis или ActiveMQ. То же самое справедливо для сходства между RabbitTemplate и JmsTemplate.

Завершая эту главу, рассмотрим еще один вариант обмена сообщениями, поддерживаемый Spring: Apache Kafka.

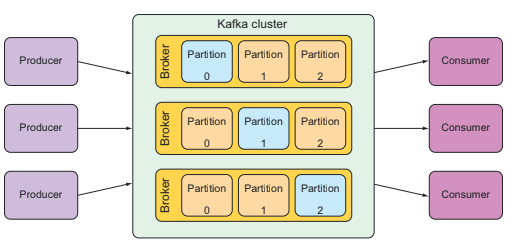
8.3 Обмен сообщениями с Kafka

Apache Kafka - это новейший вариант обмена сообщениями, который мы рассмотрим в этой главе. На первый взгляд, Kafka - это брокер сообщений, такой же, как ActiveMQ, Artemis или Rabbit. Но у Кафки есть несколько уникальных козырей в рукавах.

Kafka предназначена для работы в кластере, обеспечивая отличную масштабируемость.  И разделяя свои темы по всем экземплярам в кластере добивается особой устойчивости. Принимая во внимание, что RabbitMQ имеет дело главным образом с очередями на слушателях, Kafka использует темы только для обмена сообщениями в pub/sub - сообщениях.

Темы Кафки распространяются на всех брокеров в кластере. Каждый узел в кластере выступает в качестве лидера для одной или нескольких тем, отвечая за данные этой темы и реплицируя их на другие узлы в кластере.

Далее, каждая тема может быть разделена на несколько разделов. В этом случае каждый узел в кластере является лидером для одного или нескольких разделов темы, но не для всей темы. Ответственность за тему распределена по всем узлам. Рисунок 8.2 иллюстрирует, как это работает.



**Рис. 8.2. Кластер Kafka состоит из нескольких брокеров, каждый из которых выступает в роли лидера по разделам тем.**

Kafka обладает уникальной архитектурой, и я призываю вас прочитать больше об этом в Kafka in Action Дилана Скотта (Manning, 2017). Для наших целей мы сосредоточимся на том, как отправлять сообщения и получать их от Kafka с помощью Spring.

8.3.1 Настройка Spring для обмена сообщениями Kafka

Чтобы начать использовать Kafka для обмена сообщениями, вам нужно добавить соответствующие зависимости в вашу сборку. Однако, в отличие от параметров JMS и RabbitMQ, для Kafka нет стартового Spring Boot. Но не бойтесь; вам понадобится только одна зависимость:

*<dependency>*

*<groupId>org.springframework.kafka</groupId>*

*<artifactId>spring-kafka</artifactId>*

*</dependency>*

Эта зависимость вносит все, что нужно для Кафки в проект. Более того, её присутствие инициирует автоконфигурацию Spring Boot для Kafka, которая, помимо прочего, организует KafkaTemplate в контексте приложения Spring. Все, что вам нужно сделать, это заинжектить KafkaTemplate и приступить к отправке и получению сообщений.

Однако прежде чем приступить к отправке и получению сообщений, вы должны знать о некоторых свойствах, которые пригодятся при работе с Kafka. В частности, KafkaTemplate по умолчанию работает с брокером Kafka на локальном хосте, прослушивая порт 9092. Это нормально, если брокер Kafka запускается локально при разработке приложения, но когда придет время перейти к продакшену, вам потребуется настроить другой хост и порт.

Свойство spring.kafka.bootstrap-servers задает расположение одного или нескольких серверов Kafka, используемых для установления начального соединения с кластером Kafka. Например, если один из серверов Kafka в кластере работает kafka.tacocloud.com и прослушивая порт 9092, вы можете настроить его местоположение в YAML следующим образом:

*spring:*

*kafka:*

*bootstrap-servers:*

*- kafka.tacocloud.com:9092*

Но обратите внимание spring.kafka.bootstrap-servers - является множественным числом и принимает список. Таким образом, вы можете предоставить ему несколько серверов Kafka в кластере:

*spring:*

*kafka:*

*bootstrap-servers:*

*- kafka.tacocloud.com:9092*

*- kafka.tacocloud.com:9093*

*- kafka.tacocloud.com:9094*

С настройкой Kafka в вашем проекте вы готовы отправлять и получать сообщения. Начнем с отправки объектов Order в Kafka с помощью KafkaTemplate.

8.3.2 Отправка сообщений с помощью KafkaTemplate

Во многих отношениях KafkaTemplate похож на своих аналогов в JMS и RabbitMQ. В то же время, это совсем другое. Это становится очевидным, когда мы рассмотрим методы отправки сообщений:

*ListenableFuture<SendResult<K, V>> send(String topic, V data);*

*ListenableFuture<SendResult<K, V>> send(String topic, K key, V data);*

*ListenableFuture<SendResult<K, V>> send(String topic, Integer partition, K key, V data);*

*ListenableFuture<SendResult<K, V>> send(String topic, Integer partition, Long timestamp, K key, V data);*

*ListenableFuture<SendResult<K, V>> send(ProducerRecord<K, V> record);*

*ListenableFuture<SendResult<K, V>> send(Message<?> message);*

*ListenableFuture<SendResult<K, V>> sendDefault(V data);*

*ListenableFuture<SendResult<K, V>> sendDefault(K key, V data);*

*ListenableFuture<SendResult<K, V>> sendDefault(Integer partition, K key, V data);*

*ListenableFuture<SendResult<K, V>> sendDefault(Integer partition, Long timestamp, K key, V data);*

Первое, что вы могли заметить, это то, что нет методов convertAndSend(). Это потому, что KafkaTemplate написана с помощью дженериков и может работать с типами доменов непосредственно при отправке сообщений. В некотором смысле, все методы send() выполняют функцию convertAndSend ().

Вы также могли заметить, что есть несколько параметров send() и sendDefaul (), которые сильно отличаются от тех, которые вы использовали с JMS и Rabbit. При отправке сообщений в Kafka вы можете указать следующие параметры, которые будут определять способ отправки сообщения:

-*topic*- тема для отправки сообщения (требуется для send())

-*partition*- раздел для записи topic (необязательно)

-*key*- ключ для отправки на запись (необязательно)

-*timestamp*- время (необязательно; по умолчанию используется System.currentTimeMillis())

-*полезные данные* (полезная нагрузка) (обязательно)

Тема и полезные данные являются двумя наиболее важными параметрами. Разделы и ключи мало влияют на то, как вы используете KafkaTemplate, кроме дополнительной информации, предоставляемой в качестве параметров для send() и sendDefault(). Для наших целей мы собираемся сосредоточиться на отправке полезной нагрузки сообщения в заданную тему и не беспокоиться о разделах и ключах.

Для метода send() вы также можете выбрать отправку ProducerRecord, который является немногим больше, чем тип, который содержит все предыдущие параметры в одном объекте. Вы также можете отправить объект Message, но для этого потребуется преобразовать ваши доменные объекты в Message. Как правило, проще использовать один из других методов, чем создавать и отправлять объект ProducerRecord или Message.

Используя KafkaTemplate и его метод send(), вы можете написать реализацию OrderMessagingService на основе Kafka. Следующий листинг показывает, как может выглядеть такая реализация.

**Листинг 8.8. Отправка заказов с помощью KafkaTemplate**

*package tacos.messaging;*

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.kafka.core.KafkaTemplate;*

*import org.springframework.stereotype.Service;*

*@Service*

*public class KafkaOrderMessagingService implements OrderMessagingService {*

*private KafkaTemplate<String, Order> kafkaTemplate;*

*@Autowired*

*public KafkaOrderMessagingService(*

*KafkaTemplate<String, Order> kafkaTemplate) {*

*this.kafkaTemplate = kafkaTemplate;*

*}*

*@Override*

*public void sendOrder(Order order) {*

*kafkaTemplate.send("tacocloud.orders.topic", order);*

*}*

*}*

В этой новой реализации OrderMessagingService метод sendOrder() использует метод send() внедренного KafkaTemplate для отправки Order в тему с именем tacocloud.orders.topic. За исключением слова «Кафка», разбросанного по коду, оно не сильно отличается от кода, который вы написали для JMS и Rabbit.

Если задать тему по умолчанию, можно немного упростить метод sendOrder(). Во-первых, установить тему по умолчанию в tacocloud.orders.topic, установив свойство spring.kafka.template.default-topic:

*spring:*

*kafka:*

*template:*

*default-topic: tacocloud.orders.topic*

Затем в методе sendOrder() вы можете вызвать sendDefault() вместо send() и не указывать имя темы:

*@Override*

*public void sendOrder(Order order) {*

*kafkaTemplate.sendDefault(order);*

*}*

Теперь, когда ваш код для отправки сообщений был написан, давайте обратим наше внимание на написание кода, который будет получать эти сообщения от Kafka.

8.3.3 Написание Kafka листинеров

Помимо уникальных сигнатур методов для send() и sendDefault(), KafkaTemplate отличается от JmsTemplate и RabbitTemplate тем, что не предлагает никаких методов для получения сообщений. Это означает, что единственный способ использовать сообщения из темы Kafka с помощью Spring - написать листинер сообщений.

Для Kafka листинеры сообщений определяются как методы, аннотированные @KafkaListener. Аннотация @KafkaListener примерно аналогична @JmsListener и @RabbitListener и используется практически одинаково. В следующем листинге показано, как может выглядеть получатель заказа на основе слушателя, если он написан для Kafka.

**Листинг 8.9. Получение заказов с помощью @KafkaListener**

*package tacos.kitchen.messaging.kafka.listener;*

*import org.springframework.beans.factory.annotation.Autowired;*

*import org.springframework.kafka.annotation.KafkaListener;*

*import org.springframework.stereotype.Component;*

*import tacos.Order;*

*import tacos.kitchen.KitchenUI;*

*@Component*

*public class OrderListener {*

*private KitchenUI ui;*

*@Autowired*

*public OrderListener(KitchenUI ui) {*

*this.ui = ui;*

*}*

*@KafkaListener(topics="tacocloud.orders.topic")*

*public void handle(Order order) {*

*ui.displayOrder(order);*

*}*

*}*

Метод handle() имеет аннотацию @KafkaListener, чтобы указать, что его следует вызывать при поступлении сообщения в теме с именем tacocloud.orders.topic. Как написано в листинге 8.9, для handle() предоставляется только Order (полезная нагрузка). Но если вам нужны дополнительные метаданные из сообщения, он также может принять объект ConsumerRecord или Message.

Например, следующая реализация handle() принимает ConsumerRecord, чтобы вы могли регистрировать раздел и метку времени сообщения:

*@KafkaListener(topics="tacocloud.orders.topic")*

*public void handle(Order order, ConsumerRecord<Order> record) {*

*log.info("Received from partition {} with timestamp {}",*

*record.partition(), record.timestamp());*

*ui.displayOrder(order);*

*}*

Точно так же вы можете запросить Message вместо ConsumerRecord и достичь того же:

*@KafkaListener(topics="tacocloud.orders.topic")*

*public void handle(Order order, Message<Order> message) {*

*MessageHeaders headers = message.getHeaders();*

*log.info("Received from partition {} with timestamp {}",*

*headers.get(KafkaHeaders.RECEIVED\_PARTITION\_ID)*

*headers.get(KafkaHeaders.RECEIVED\_TIMESTAMP));*

*ui.displayOrder(order);*

*}*

Стоит отметить, что полезная нагрузка сообщения также доступна через ConsumerRecord.value() или Message.getPayload(). Это означает, что вы можете запросить Order через эти объекты вместо того, чтобы запрашивать его напрямую как параметр handle().

ИТОГ

-Асинхронный обмен сообщениями обеспечивает уровень перенаправления между взаимодействующими приложениями, что обеспечивает более слабую связь и большую масштабируемость.

-Spring поддерживает асинхронный обмен сообщениями с JMS, RabbitMQ или Apache Kafka.

-Приложения могут использовать основанные на шаблонах клиенты (JmsTemplate, RabbitTemplate или KafkaTemplate) для отправки сообщений через брокер сообщений.

-Приложения могут использовать сообщения в модели на основе запросов, используя те же клиенты на основе шаблонов.

-Сообщения также можно передавать потребителям, применяя аннотации листенеров сообщений(@JmsListener, @RabbitListener или @KafkaListener) к методам bean.

# [Spring in Action Covers Spring 5.0 перевод на русский. Глава 9](https://beginner-mercenary.blogspot.com/2019/08/spring-in-action-covers-spring-50-9.html)

## 9. Integrating Spring

### Эта глава охватывает

**-Обработка данных в режиме реального времени**

**-Определение интеграционных потоков**

**-Использование определения JAVA DSL Spring Integration**

**-Интеграция с электронной почтой, файловыми системами и другими внешними системами**

Одна из самых неприятных вещей, с которыми я сталкиваюсь во время путешествия - это длительный перелет и плохое или несуществующее интернет-соединение в полете. Мне нравится использовать свое свободное время, чтобы выполнить некоторую работу, включая написание страниц этой книги. Если нет сетевого подключения, я в невыгодном положении, если мне нужно получить библиотеку или найти Java Doc, и я не могу выполнить большую часть работы. Я научился брать с собой книгу для чтения в таких случаям.

Точно так же, как нам необходимо подключиться к Интернету для продуктивной работы, многие приложения должны подключаться к внешним системам для выполнения своей работы. Приложению может потребоваться читать или отправлять электронные письма, взаимодействовать с внешним API или использовать данные, содержащиеся в базе данных. И поскольку данные поступают из этих внешних систем или записываются в эти внешние системы, приложению может потребоваться каким-то образом обрабатывать данные, чтобы перевести их в собственный домен приложения или из него.

В этой главе вы узнаете, как использовать общие шаблоны интеграции Spring Integration. Spring Integration - это готовая к использованию реализация многих шаблонов интеграции, которые каталогизированы в Enterprise Integration Patterns by Gregor Hohpe and Bobby Woolf (Addison-Wesley, 2003). Каждый шаблон реализован как компонент, через который сообщения передают данные в конвейер. Используя конфигурацию Spring, вы можете собрать эти компоненты в конвейер, по которому проходят потоки данных. Давайте начнем с определения простого потока интеграции, который познакомит  вас со многие функциями и характеристиками работы с Spring Integration.

### 9.1 Объявление простого потока интеграции

Вообще говоря, Spring Integration позволяет создавать интеграционные потоки, через которые приложение может получать или отправлять данные на некоторый ресурс, внешний по отношению к самому приложению. Одним из таких ресурсов, с которым приложение может интегрироваться, является файловая система. Поэтому среди многих компонентов Spring Integration есть канальные адаптеры для чтения и записи файлов.

Чтобы получить удовольствие от Spring Integration, вы создадите поток интеграции, который записывает данные в файловую систему. Для начала вам нужно добавить Spring Integration в ваш проект. Для Maven необходимы следующие зависимости:

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-integration</artifactId>

</dependency>

<dependency>

    <groupId>org.springframework.integration</groupId>

    <artifactId>spring-integration-file</artifactId>

</dependency>

Первая зависимость - это стартер Spring Boot для Spring Integration. Эта зависимость важна для разработки потока Spring Integration, независимо от того, с чем он будет интегрироваться. Как и все starter зависимости Spring Boot, он доступен в виде флажка в форме Initializr.

Вторая зависимость - для файла Spring Integration endpoint модуля. Этот модуль является одним из более двух десятков endpoint модулей, используемых для интеграции с внешними системами. Мы поговорим подробнее о endpoint модулях в разделе 9.2.9. Но на данный момент известно, что endpoint модуль файлов предлагает возможность загружать файлы из файловой системы в поток интеграции и / или записывать данные из потока на файловую систему.

Затем необходимо создать способ для приложения, чтобы отправить данные в поток интеграции, так что он может быть записан в файл. Для этого вы создадите интерфейс шлюза, например, показанный ниже.

**Листинг 9.1 Интерфейс шлюза сообщений для преобразования вызовов методов в сообщения**

package sia5;

import org.springframework.integration.annotation.MessagingGateway;

import org.springframework.integration.file.FileHeaders;

import org.springframework.messaging.handler.annotation.Header;

@MessagingGateway(defaultRequestChannel="textInChannel") //Объявляет шлюз сообщений

public interface FileWriterGateway {

    void writeToFile(@Header(FileHeaders.FILENAME) String filename,String data); //Пишет в файл

}

Хотя это простой Java интерфейс, о FileWriterGateway можно многое сказать. Первое, что вы заметите, это то, что он аннотирован @MessagingGateway. Эта аннотация указывает Spring Integration на создание реализации этого интерфейса должно поисходить во время выполнения - подобно тому, как Spring Data автоматически генерирует реализации интерфейсов репозитория. Другие части кода будут использовать этот интерфейс, когда им нужно будет переписать файл.

Атрибут defaultRequestChannel @MessagingGateway указывает, что любые сообщения, полученные в результате вызова методов интерфейса, должны быть отправлены в данный канал сообщений. В этом случае вы заявляете, что любые сообщения, возникающие в результате вызова writeToFile(), должны отправляться на канал, имя которого textInChannel.

Что касается метода writeToFile(), он принимает имя файла в виде String и String, содержащий текст, который должен быть записан в файл. Что примечательно в сигнатуре этого метода, так это то, что параметр имени файла аннотирован @Header. В этом случае аннотация @Header указывает, что значение, переданное в filename, должно быть помещено в заголовок сообщения (указанный как FileHeaders.FILENAME, который разрешается в file\_name), а не в полезную нагрузку сообщения. Значение параметра data, помещается в полезную нагрузку сообщения.

Теперь, когда у вас есть шлюз сообщений, вам нужно настроить интеграционный поток. Несмотря на то, что начальная зависимость Spring Integration, которую вы добавили в свою сборку, обеспечивает необходимую автоконфигурацию для Spring Integration, вы все равно должны написать дополнительные конфигурации, чтобы определить потоки, отвечающие потребностям приложения. Вот три варианта конфигурации для объявления потоков интеграции:

-XML конфигурация

-Java конфигурация

- Java конфигурация с DSL

Мы рассмотрим все три из этих стилей конфигурации для Spring Integration, начиная с устаревшей конфигурации XML.

### 9.1.1 Определение потоков интеграции через XML

Хотя я избегал использования конфигурации XML в этой книге, Spring Integration имеет долгую историю интеграционных потоков, определенных в XML. Поэтому я считаю, что мне стоит показать хотя бы один пример потока интеграции, определенного через XML. В следующем списке показано, как настроить пример потока через XML.

**Листинг 9.2 Определение потоков интеграции через 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:int="http://www.springframework.org/schema/integration"

     xmlns:int-file="http://www.springframework.org/schema/integration/file"

     xsi:schemaLocation="http://www.springframework.org/schema/beans

     http://www.springframework.org/schema/beans/spring-beans.xsd

     http://www.springframework.org/schema/integration

     http://www.springframework.org/schema/integration/spring-integration.xsd

     http://www.springframework.org/schema/integration/file

     http://www.springframework.org/schema/integration/file/springintegration-file.xsd">

<int:channel id="textInChannel" /> //Объявление textInChannel

<int:transformer id="upperCase"

     input-channel="textInChannel"

     output-channel="fileWriterChannel"

     expression="payload.toUpperCase()" /> // Преобразование текста

<int:channel id="fileWriterChannel" />  //Объявление fileWriterChanne

<int-file:outbound-channel-adapter id="writer"

      channel="fileWriterChannel"

      directory="/tmp/sia5/files"

      mode="APPEND"

      append-new-line="true" />  //Записывает текст в файл

</beans>

Детализация XML в листинге 9.2:

* Настаиваем канал с именем textInChannel. Это тот же канал что установлен для запроса FileWriterGateway. При вызове метода writeToFile() в FileWriterGateway, результирующее сообщение публикуется на этом канале.
* Настроили преобразователь (transformer), который получает сообщения от textInChannel. Он использует выражение Spring Expression Language (SpEL) для вызова toUpperCase() в полезной нагрузке сообщения. Результат операции верхнего регистра затем публикуется в fileWriterChannel.
* Настроили  канал с именем fileWriterChannel. Этот канал служит проводником, который соединяет transformer с адаптером исходящего канала.
* Наконец, настроили адаптер исходящего канала, используя пространство имен int-file. Это пространство имен XML предоставляется модулем Spring Integration для записи файлов. Он настроен так что получает сообщения от fileWriterChannel и записывает полезные данные сообщения в файл, имя которого указано в заголовке file\_name сообщения в каталоге, указанном в атрибуте каталога. Если файл уже существует, в него будут добавлены новые данные, а не перезаписан.

Этот поток показан на рис. 9.1 с использованием графических элементов, стилизованных под Enterprise Integration Patterns.



Шлюз записи файлов - Текст в канале - Uppercase transformer - Канал записи файлов - Адаптер исходящего канала файла

**Рисунок 9.1 Поток интеграции записи файлов**

Если вы хотите использовать конфигурацию XML в приложении Spring Boot, вам необходимо импортировать XML как ресурс в приложение Spring. Самый простой способ сделать это - использовать аннотацию Spring @ImportResource в одном из классов конфигурации вашего приложения:

@Configuration

@ImportResource("classpath:/filewriter-config.xml")

public class FileWriterIntegrationConfig { ... }

Хотя конфигурация на основе XML хорошо послужила Spring Integration, большинство разработчиков опасаются использовать XML. (И, как я уже сказал, я избегаю конфигурации через XML в этой книге.) Давайте отложим эти угловые скобки и обратим наше внимание на стиль конфигурации через Java в Spring Integration.

#### 9.1.2 Настройка потоков интеграции через Java

Большинство современных приложений Spring отказались от конфигурации XML в пользу конфигурации Java. Фактически в приложениях Spring Boot конфигурация Java является естественным стилем, дополняющим автоконфигурацию. Поэтому, если вы добавляете поток интеграции в приложение Spring Boot, имеет смысл определить поток посредством Java.

В качестве примера того, как написать поток интеграции используя Java конфигурацию, взгляните на следующий листинг. Это показывает тот же процесс интеграции записи файлов, что и раньше, но на этот раз он написан на Java.

package sia5;

import java.io.File; import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.integration.annotation.ServiceActivator;

import org.springframework.integration.annotation.Transformer;

import org.springframework.integration.file.FileWritingMessageHandler;

import org.springframework.integration.file.support.FileExistsMode;

 import org.springframework.integration.transformer.GenericTransformer;

@Configuration

public class FileWriterIntegrationConfig {

    @Bean

    @Transformer(inputChannel="textInChannel", //Объявляем transformer

                outputChannel="fileWriterChannel")

    public GenericTransformer<String, String> upperCaseTransformer() {

        return text -> text.toUpperCase();

    }

    @Bean

    @ServiceActivator(inputChannel="fileWriterChannel")

    public FileWritingMessageHandler fileWriter() { //Объявляем запись в файл

        FileWritingMessageHandler handler =

                    new FileWritingMessageHandler(new File("/tmp/sia5/files"));

        handler.setExpectReply(false);

        handler.setFileExistsMode(FileExistsMode.APPEND);

        handler.setAppendNewLine(true);

        return handler;

    }

}

В Java конфигурации вы объявляете два bean-компонента: преобразователь (transformer) и обработчик сообщения для записи файла. Трансформатор является универсальным трансформатором. Поскольку GenericTransformer является функциональным интерфейсом, вы можете предоставить его реализацию в виде лямбды, которая вызывает toUpperCase() в тексте сообщения. Преобразователь bean аннотируется @Transformer, определяющим его как преобразователь в потоке интеграции, который принимает сообщения в канале с именем textInChannel и записывает сообщения в канал с именем fileWriterChannel.

Что касается bean для записи файлов, он помечается @ServiceActivator, чтобы указать, что он будет принимать сообщения от fileWriterChannel и передавать эти сообщения сервису, определенному экземпляром FileWritingMessageHandler. FileWritingMessageHandler - это обработчик сообщений, который записывает полезную нагрузку сообщения в файл в указанном каталоге, используя имя файла, указанное в заголовке file\_name сообщения. Как и в примере с XML, FileWritingMessageHandler настроен на добавление в файл новой записи.

Уникальной особенностью конфигурации bean-компонента FileWritingMessageHandler является то, что существует вызов метода setExpectReply(false), который указывает, что активатору службы не следует ожидать ответный канал (канал, через который значение может быть возвращено вышестоящим компонентам в потоке).  Если вы не вызываете setExpectReply(), для записи файла bean по умолчанию имеет значение true, и, хотя функция по-прежнему работает должным образом, вы увидите несколько ошибок, указывающих, что канал ответа не настроен.

Вы также заметите, что вам не нужно явно объявлять каналы. Каналы textInChannel и fileWriterChannel будут созданы автоматически, если бинов с этими именами не существует. Но если вам нужен больший контроль над конфигурацией каналов, вы можете явно создать их как bean-компоненты:

@Bean

public MessageChannel textInChannel() {

    return new DirectChannel();

}

...

@Bean

public MessageChannel fileWriterChannel() {

    return new DirectChannel();

}

Конфигурация посредством Java, возможно, проще для чтения и немного короче, и, безусловно, соответствует конфигурации “только Java”, о которой я рассказываю в этой книге. Но это можно сделать еще более упорядоченным с помощью стиля конфигурации Spring Integration Java DSL (предметно-ориентированного языка).

### 9.1.3 Использование конфигурации DSL Spring Integration

Давайте еще раз попробуем настроить поток интеграции записи файлов. На этот раз вы все равно настроите его через Java, но вы будете использовать Spring Integration Java DSL. Вместо объявления отдельного bean для каждого компонента в потоке, вы объявите один bean, который определяет весь поток.

**Листинг 9.4. Предоставление свободного API для проектирования потоков интеграции**

package sia5;

import java.io.File;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.integration.dsl.IntegrationFlow;

import org.springframework.integration.dsl.IntegrationFlows;

import org.springframework.integration.dsl.channel.MessageChannels;

import org.springframework.integration.file.dsl.Files;

import org.springframework.integration.file.support.FileExistsMode;

@Configuration

public class FileWriterIntegrationConfig {

    @Bean

    public IntegrationFlow fileWriterFlow() {

        return IntegrationFlows

            .from(MessageChannels.direct("textInChannel")) //Входящий канал

            .<String, String>transform(t -> t.toUpperCase()) //Объявление transformer

            .handle(Files                      //Обрабатывает запись в файл

            .outboundAdapter(new File("/tmp/sia5/files"))

            .fileExistsMode(FileExistsMode.APPEND)

            .appendNewLine(true))

            .get();

    }

}

Эта новая конфигурация настолько кратка, насколько это возможно, захватывая весь поток одним bean

методом. Класс IntegrationFlows инициирует builder API, из которого можно объявить поток.

В листинге 9.4 вы начинаете с получения сообщений из канала с именем textInChannel, которые затем направляются в transformer, который преобразует в верхний регистр текст полезной нагрузки сообщения. После transformer сообщения обрабатываются адаптером исходящего канала, созданным из типа Files, предоставленного в файловом модуле Spring Integration. Наконец, вызов get() создает IntegrationFlow для возврата. Вкратце, этот одиночный bean метод создает тот же поток интеграции, что и примеры конфигурации XML и Java.

Заметьте, что, как и в примере Java конфигурации, вам не нужно явно объявлять bean-компоненты канала. Несмотря на то, что вы ссылаетесь на textInChannel, он автоматически создается Spring Integration, потому что не существует никакого bean-компонента канала с таким именем. Но вы можете явно объявить bean-компонент канала, если хотите.

Что касается канала, соединяющего преобразователь (transformer) с адаптером исходящего канала, вы даже не ссылаетесь на него по имени. Если необходимо явно настроить канал, вы можете ссылаться на него по имени в определении потока с помощью вызова channel():

@Bean

public IntegrationFlow fileWriterFlow() {

    return IntegrationFlows

        .from(MessageChannels.direct("textInChannel"))

        .<String, String>transform(t -> t.toUpperCase())

        .channel(MessageChannels.direct("fileWriterChannel"))

        .handle(Files

        .outboundAdapter(new File("/tmp/sia5/files"))

        .fileExistsMode(FileExistsMode.APPEND)

        .appendNewLine(true))

        .get();

}

При работе с Java DSL от Spring Integration (как и с любым свободно распространяемым API) нужно помнить, что для обеспечения удобочитаемости вы должны использовать пробелы. В приведенном здесь примере я старался делать отступы для обозначения блоков связанного кода. Для более длинных и сложных потоков вы можете даже рассмотреть выделение частей потока в отдельные методы или подпотоки для лучшей читаемости.

Теперь, когда вы увидели простой поток, определенный с использованием трех разных стилей конфигурации, давайте вернемся назад и взглянем на общую картину Spring Integration.

### 9.2. Обзор представления Spring Integration

Spring Integration охватывает множество сценариев интеграции. Попытка включить все это в одну главу - это все равно, что поместить слона в конверт. Вместо всестороннего рассмотрения Spring Integration, я представлю фотографию слона Spring Integration, чтобы дать вам некоторое представление о том, как он работает. Затем вы создадите еще один поток интеграции, который добавит функциональность в приложение Taco Cloud.

Поток интеграции состоит из одного или нескольких компонентов. Прежде чем писать больше кода, мы кратко рассмотрим роль каждого из этих компонентов в процессе интеграции:

-Channels (Каналы) —Передача сообщений от одного элемента к другому..

-Filters (Фильтры)—Позволяет сообщениям проходить через поток при соблюдении определенных условий.

-Transformers (Трансформаторы)—Меняют значение сообщения и/или преобразуют полезные нагрузки сообщений из одного типа в другой.

-Routers (Маршрутизаторы)—Направляет сообщения на один из нескольких каналов, обычно на основе заголовков сообщений

-Splitters (Разделить)—Разделяет входящие сообщения на два или более сообщений, каждое из которых отправляется на разные каналы.

-Aggregators (Агрегаторы)— Противоположность разделителям, объединяет несколько сообщений, поступающих из отдельных каналов, в одно сообщение.

-Service activators (Активаторы службы)—Передает сообщение какому-либо методу Java для обработки, а затем опубликует возвращаемое значение в выходном канале.

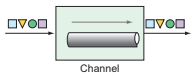
-Channel adapters (Канальные адаптеры)—Подключает канал к какой-либо внешней системе или транспорту. Может принимать входные данные или записывать данные во внешнюю систему.

-Gateways (Шлюзы)—Передает данные в интеграционный поток через интерфейс.

Вы уже видели некоторые из этих компонентов в действии, когда вы описали процесс интеграции записи файлов. Интерфейс FileWriterGateway был gateway, через который приложение отправляло текст для записи в файл. Вы также определили transformer для преобразования данного текста в верхний регистр; затем вы объявили сервисный gateway, который выполнял задачу записи текста в файл. И у потока было два канала, textInChannel и fileWriterChannel, которые связывали другие компоненты друг с другом. Теперь краткий обзор компонентов потока интеграции, как и было обещано.

### 9.2.1 Каналы сообщений

Каналы сообщений - это средства, с помощью которых сообщения проходят через интеграционный конвейер (рисунок 9.2). Это трубы, которые соединяют между собой все остальные части Spring Integration.



**Рисунок 9.2 Каналы сообщений - это каналы, по которым потоки данных проходят между другими компонентами в потоке интеграции.**

Spring Integration предоставляет несколько реализаций канала, включая следующие:

-PublishSubscribeChannel—Сообщения, опубликованные в PublishSubscribeChannel, передаются одному или нескольким потребителям. Если есть несколько потребителей, все они получают сообщение.

-QueueChannel—Сообщения, опубликованные в QueueChannel, сохраняются в очереди до тех пор, пока пользователь не извлечет их в порядке поступления (FIFO). Если есть несколько потребителей, только один из них получает сообщение.

-PriorityChannel—Как и QueueChannel, но вместо поведения FIFO сообщения извлекаются потребителями на основе заголовка приоритета сообщения.

-RendezvousChannel—Как и QueueChannel, за исключением того, что отправитель блокирует канал до тех пор, пока потребитель не получит сообщение, эффективно синхронизируя отправителя с потребителем.

-DirectChannel - Подобно PublishSubscribeChannel, но отправляет сообщение одному потребителю, вызывая потребителя в том же потоке, что и отправитель. Это позволяет транзакциям распространяться по каналу.

-ExecutorChannel - аналогичен DirectChannel, но отправка сообщения происходит через TaskExecutor, происходящий в отдельном потоке от отправителя. Этот тип канала не поддерживает транзакции, которые охватывают канал.

-FluxMessageChannel - канал сообщений Reactive Streams Publisher, основанный на Flux в Project Reactor. (Мы поговорим подробнее о Reactive Streams, Reactor и Flux в главе 10.)

Как в конфигурации Java, так и в стилях Java DSL входные каналы создаются автоматически с DirectChannel по умолчанию. Но если вы хотите использовать другую реализацию канала, вам необходимо явно объявить канал как bean-компонент и сослаться на него в потоке интеграции. Например, чтобы объявить PublishSubscribeChannel, вы должны объявить следующий метод @Bean:

@Bean

public MessageChannel orderChannel() {

    return new PublishSubscribeChannel();

}

Тогда вы будете ссылаться на этот канал по имени в определении потока интеграции. Например, если канал использовался bean-компонентом-активатором службы, вы бы ссылались на него в атрибуте inputChannel @ServiceActivator:

@ServiceActivator(inputChannel="orderChannel")

Или, если вы используете стиль конфигурации JAVA DSL, вы должны ссылаться на него с помощью вызова channel():

@Beanpublic

IntegrationFlow orderFlow() {

    return IntegrationFlows

        …

        .channel("orderChannel")

        …

        .get();

}

Важно отметить, что если вы используете QueueChannel, потребители должны быть настроены с помощью средства опроса. Например, предположим, что вы объявили QueueChannel bean следующим образом:

@Bean

public MessageChannel orderChannel() {

    return new QueueChannel();

}

Вам нужно будет убедиться, что потребитель настроен на опрос канала для сообщений. В случае службы активатора аннотация @ServiceActivator может выглядеть следующим образом:

@ServiceActivator(inputChannel="orderChannel", poller=@Poller(fixedRate="1000"))

В этом примере служба активатора опрашивает канал с именем orderChannelevery каждую 1 секунду (или 1000 мс).

### 9.2.2 Фильтры

Фильтры могут быть размещены в середине конвейера интеграции, чтобы разрешить или запретить переход сообщений к следующему шагу в потоке (рис.9.3).

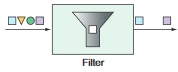


Рисунок 9.3 Фильтры, основанные на некоторых критериях, разрешают или запрещают передачу сообщений в конвейере.

Например, предположим, что сообщения, содержащие целочисленные значения, публикуются через канал с именем numberChannel, но вы хотите, чтобы на канал с именем evenNumberChannel передавались только четные числа. В этом случае вы можете объявить фильтр с аннотацией @Filter следующим образом:

@Filter(inputChannel="numberChannel", outputChannel="evenNumberChannel")

public boolean evenNumberFilter(Integer number) {

    return number % 2 == 0;

}

Кроме того, если вы используете стиль конфигурации JAVA DSL для определения потока интеграции, вы можете сделать вызов filter() следующим образом:

@Bean

public IntegrationFlow evenNumberFlow(AtomicInteger integerSource) {

    return IntegrationFlows

    …

    .<Integer>filter((p) -> p % 2 == 0)

    …

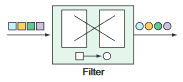
    .get();

}

В этом случае для реализации фильтра используется лямбда. Но, по правде говоря, метод filter() принимает GenericSelector в качестве аргумента. Это означает, что вы можете реализовать интерфейс GenericSelector вместо этого, если ваша фильтрация слишком сложна для простой лямбды.

### 9.2.3 Трансформаторы

Трансформаторы выполняют некоторые операции над сообщениями, обычно приводя к другому сообщению и, возможно, с другим типом полезной нагрузки (см. рис.9.4). Преобразование может быть чем-то простым, например, выполнение математических операций над числом или манипулирование строковым значением.  Или преобразование может быть более сложным,например, использование строкового значения, представляющего ISBN, для поиска и возврата сведений о соответствующей книге.



**Рисунок 9.4 Трансформаторы преобразуют сообщения по мере их прохождения через поток интеграции.**

Например, предположим, что целочисленные значения публикуются на канале с именем numberChannel, и вы хотите преобразовать эти числа в String, содержащую Римский числовой эквивалент.  В этом случае вы можете объявить bean GenericTransformer и аннотировать его с помощью @Transformer следующим образом:

@Bean

@Transformer(inputChannel="numberChannel", outputChannel="romanNumberChannel")

public GenericTransformer<Integer, String> romanNumTransformer() {

    return RomanNumbers::toRoman;

}

Аннотация @Transformer определяет этот компонент как компонент transformer, который получает целочисленные значения из канала с именем numberChannel и использует статический метод с именем toRoman() для выполнения преобразования. (Метод toRoman () статически определен в классе с именем RomanNumbers и ссылается здесь со ссылкой на метод) Результат будет опубликован на канале с именем romanNumberChannel.

В стиле конфигурации Java DSL еще проще с вызовом transform(), передавая ссылку на метод toRoman():

@Bean

public IntegrationFlow transformerFlow() {

    return IntegrationFlows

        …

        .transform(RomanNumbers::toRoman)

        …

        .get();

}

Хотя вы использовали ссылку на метод в обоих примерах кода преобразователя, знайте, что преобразователь также можно указать как лямбду. Или, если преобразователь достаточно сложен, чтобы потребовался отдельный класс Java, вы можете внедрить его как bean-компонент в конфигурацию потока и передать ссылку на метод transform():

@Bean

public RomanNumberTransformer romanNumberTransformer() {

    return new RomanNumberTransformer();

}

@Bean

public IntegrationFlow transformerFlow( RomanNumberTransformer romanNumberTransformer) {

    return IntegrationFlows

        …

        .transform(romanNumberTransformer)

        …

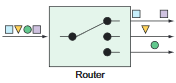
        .get();

}

Здесь вы объявляете bean-компонент типа RomanNumberTransformer, который сам является реализацией Spring Integration Transformer или интерфейса GenericTransformer. Bean внедряется в метод transformerFlow() и передается в метод transform() при определении потока интеграции.

### 9.2.4 Маршрутизаторы

Маршрутизаторы, основанные на некоторых критериях маршрутизации, позволяют разветвляться в потоке интеграции,направляя сообщения в разные каналы (см. рисунок 9.5).



**Рис. 9.5 Маршрутизаторы направляют сообщения в различные каналы на основе некоторых критериев, применяемых к сообщениям.**

Например, предположим, что у вас есть канал с именем numberChannel, через который проходят целочисленные значения. Допустим, вы хотите направить все сообщения с четными номерами в канал с именем evenChannel, а сообщения с нечетными номерами направляются в канал с именем oddChannel. Чтобы создать такую маршрутизацию в потоке интеграции, вы можете объявить bean-компонент типа AbstractMessageRouter и аннотировать bean-компонент с помощью @Router:

@Bean

@Router(inputChannel="numberChannel")

public AbstractMessageRouter evenOddRouter() {

    return new AbstractMessageRouter() {

        @Override

        protected Collection<MessageChannel>

                determineTargetChannels(Message<?> message) {

            Integer number = (Integer) message.getPayload();

            if (number % 2 == 0) {

                return Collections.singleton(evenChannel());

            }

            return Collections.singleton(oddChannel());

        }

    };

}

@Bean

public MessageChannel evenChannel() {

    return new DirectChannel();

}

@Bean

public MessageChannel oddChannel() {

    return new DirectChannel();

}

Объявленный здесь bean AbstractMessageRouter принимает сообщения от входного канала с именем numberChannel.  Реализация, определенная как анонимный внутренний класс, проверяет полезную нагрузку сообщения и, если это четное число, возвращает канал с именем evenChannel (объявленный как bean). В противном случае число в полезной нагрузке канала должно быть нечетным; в этом случае возвращается канал с именем oddChannel (также созданный как bean).

В Java DSL маршрутизаторы объявляются путем вызова route() в ходе определения потока, как показано ниже:

@Bean

public IntegrationFlow numberRoutingFlow(AtomicInteger source) {

    return IntegrationFlows

        …

        .<Integer, String>route(n -> n%2==0 ? "EVEN":"ODD", mapping -> mapping

        .subFlowMapping("EVEN", sf -> sf .<Integer, Integer>transform(n -> n \* 10)

        .handle((i,h) -> { ... })

        )

        .subFlowMapping("ODD", sf -> sf .transform(RomanNumbers::toRoman) .handle((i,h) -> { ... })

        )

    )

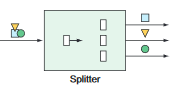
    .get();

}

Хотя по-прежнему можно объявить AbstractMessageRouter и передать его в router(), в этом примере используется лямбда для определения того, является ли полезная нагрузка сообщения четной или нечетной. Если она четная, то возвращается строковое значение EVEN. Если нечетная, то возвращается ODD.Эти значения затем используются для определения того, какое под-сопоставление будет обрабатывать сообщение.

### 9.2.5 Разделители

Иногда в потоке интеграции может быть полезно разделить сообщение на несколько сообщений, которые будут обрабатываться независимо. Разделители, как показано на рис.9.6, будут разделять и обрабатывать эти сообщения для вас.



**Рис. 9.6 Hазделители разбивают сообщения на два или более отдельных сообщения, которые могут обрабатываться отдельными подпотоками.**

Splitter-ы полезны во многих случаях, но есть два основных варианта использования, для которых вы можете использовать сплиттер:

-Полезная нагрузка сообщения содержит коллекцию элементов того же типа, которые вы хотите обработать как отдельные полезные нагрузки сообщений. Например, сообщение, содержащее список продуктов, может быть разделено на несколько сообщений с полезной нагрузкой по одному продукту в каждом

-Полезная нагрузка сообщения содержит информацию, которая, хотя и связана, может быть разделена на два или более сообщений разных типов. Например, заказ на покупку может содержать информацию о поставках, выставлении счетов и номенклатуре. Сведения о доставке могут обрабатываться одним подпотоком, выставление счетов-другим, а номенклатура - еще одним.  В этом случае за разделителем обычно следует маршрутизатор, который направляет сообщения по типу полезной нагрузки, чтобы гарантировать, что данные обрабатываются правильным подпотоком.

При разделении полезной нагрузки сообщения на два или более сообщений разных типов обычно достаточно определить POJO,  который извлекает отдельные части входящей полезной нагрузки и возвращает их как элементы коллекции.

Например, предположим, что вы хотите разделить сообщение, содержащее заказ на покупку, на два сообщения: одно содержит платежную информацию, а другое-список элементов списка. Следующий OrderSplitter сделает работу:

public class OrderSplitter {

    public Collection<Object> splitOrderIntoParts(PurchaseOrder po)

        ArrayList<Object> parts = new ArrayList<>();

        parts.add(po.getBillingInfo());

        parts.add(po.getLineItems());

        return parts;

    }

}

Затем вы можете объявить bean OrderSplitter как часть потока интеграции, аннотируя его с помощью @Splitter следующим образом:

@Bean

@Splitter(inputChannel="poChannel", outputChannel="splitOrderChannel")

public OrderSplitter orderSplitter() {

    return new OrderSplitter();

}

Здесь заказы на поставку поступают на канал с именем poChannel и делятся на OrderSplitter. Затем каждый элемент в возвращенной коллекции публикуется как отдельное сообщение в потоке интеграции в канал с именем splitOrderChannel. На этом этапе потока вы можете объявить PayloadTypeRouter для маршрутизации платежной информации и позиций в их собственный подпоток:

@Bean

@Router(inputChannel="splitOrderChannel")

public MessageRouter splitOrderRouter() {

    PayloadTypeRouter router = new PayloadTypeRouter();

    router.setChannelMapping(

        BillingInfo.class.getName(), "billingInfoChannel");

    router.setChannelMapping(

        List.class.getName(), "lineItemsChannel");

    return router;

}

Как следует из его названия, PayloadTypeRouter направляет сообщения на разные каналы на основе их типа полезной нагрузки. Здесь сконфигурировано так что, сообщения, чья полезная нагрузка имеет тип BillingInfo, направляются в канал с именем billingInfoChannel для дальнейшей обработки. Что касается позиций, они находятся в коллекции java.util.List; следовательно, вы сопоставили полезные нагрузки типа List для направления на канал с именем lineItemsChannel.

В настоящее время поток разделяется на два подпотока: один, через который проходят объекты BillingInfo, и другой, через который проходит List<LineItem>. Но что, если вы хотите разбить его дальше так, чтобы вместо работы со списком LineItems вы обрабатывали каждый LineItem отдельно? Все, что вам нужно сделать, чтобы разделить список элементов строки на несколько сообщений, по одному для каждой позиции, - это написать метод (не bean-компонент), который аннотируется с помощью @Splitter и возвращает коллекцию LineItems, возможно, что-то вроде этого:

@Splitter(inputChannel="lineItemsChannel", outputChannel="lineItemChannel")

public List<LineItem> lineItemSplitter(List<LineItem> lineItems) {

    return lineItems;

}

Когда сообщение, несущее полезную нагрузку List<LineItem>, поступает в канал с именем lineItemsChannel, оно передается методу lineItemSplitter(). Согласно правилам разделителя, метод должен возвращать коллекцию элементов, подлежащих разделению. В этом случае у вас уже есть коллекция LineItems, поэтому вы просто возвращаете коллекцию напрямую. В результате каждый LineItem в коллекции публикуется в своем собственном сообщении для канала с именем lineItemChannel.

Если вы предпочитаете использовать Java DSL для объявления конфигурации сплиттера/маршрутизатора, вы можете сделать это с помощью вызовов функций split() и route():

return IntegrationFlows

    …

    .split(orderSplitter())

    .<Object, String> route(

        p -> {

            if (p.getClass().isAssignableFrom(BillingInfo.class)) {

                return "BILLING\_INFO";

            } else {

                return "LINE\_ITEMS";

            }

        }, mapping -> mapping

            .subFlowMapping("BILLING\_INFO", sf -> sf

                .<BillingInfo> handle((billingInfo, h) -> {

                    …

                }))

            .subFlowMapping("LINE\_ITEMS", sf -> sf

                .split()

                .<LineItem> handle((lineItem, h) -> {

                    …

                }))

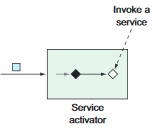
        )

    .get();

Форма определения потока в DSL определенно более краткая, но и более трудная для понимания. Для разделения порядка используется тот же OrderSplitter, что и в примере конфигурации Java. После разделения заказ разбивается по типу на два отдельных подпотока.

### 9.2.6 Активаторы службы

Активаторы службы получают сообщения из входного канала и отправляют эти сообщения в реализацию MessageHandler, как показано на рисунке 9.7.



**Рис. 9.7. Активаторы службы вызывают некоторую службу посредством MessageHandler при получении сообщения.**

Spring Integration предлагает несколько реализаций MessageHandler «из коробки» (даже PayloadTypeRouter - это реализация MessageHandler), но вам часто потребуется предоставить некоторую пользовательскую реализацию, которая будет действовать как активатор службы. Например, в следующем коде показано, как объявить bean MessageHandler, настроенный как активатор службы:

@Bean

@ServiceActivator(inputChannel="someChannel")

public MessageHandler sysoutHandler() {

    return message -> {

        System.out.println("Message payload: " + message.getPayload());

    };

}

Bean аннотируется @ServiceActivator, чтобы обозначить его как активатор службы, который обрабатывает сообщения из канала с именем someChannel. Что касается MessageHandler, он реализован через лямбду. Несмотря на то, что это простой MessageHandler, при получении сообщения он передает свою полезную нагрузку в стандартный поток вывода.

Кроме того, можно объявить активатор службы, который обрабатывает данные во входящем сообщении перед возвратом новой полезной нагрузки. В этом случае компонент должен быть GenericHandler, а не MessageHandler:

@Bean

@ServiceActivator(inputChannel="orderChannel", outputChannel="completeOrder")

public GenericHandler<Order> orderHandler( OrderRepository orderRepo) {

    return (payload, headers) -> {

        return orderRepo.save(payload);

    };

}

В этом случае активатор службы является GenericHandler, который ожидает сообщения с полезной нагрузкой типа Order.  Когда заказ поступает, он сохраняется через репозиторий; полученный сохраненный заказ возвращается для отправки в выходной канал, имя которого completeChannel.

Вы могли заметить, что GenericHandler предоставляет не только полезную нагрузку, но и заголовки сообщений (даже если пример не использует эти заголовки в любом случае).  Вы также можете использовать активаторы служб в стиле конфигурации JAVA DSL, передавая MessageHandler или GenericHandler для метода handle () в определении потока:

public IntegrationFlow someFlow() {

    return IntegrationFlows

        …

        .handle(msg -> {

            System.out.println("Message payload: " + msg.getPayload());

        })

        .get();

}

В этом случае MessageHandler задается как лямбда, но вы также можете предоставить его как ссылку на метод или даже как экземпляр класса, который реализует интерфейс  MessageHandler. Если вы даете ему ссылку на лямбду или метод, имейте в виду, что он принимает сообщение в качестве параметра.

Аналогично, handle() может быть  написан для принятия GenericHandler, если активатор службы не предназначен для завершения потока. Применяя ранее описанный активатор службы сохранения заказов, вы можете настроить поток с помощью Java DSL следующим образом:

public IntegrationFlow orderFlow(OrderRepository orderRepo) {

    return IntegrationFlows

        …

        .<Order>handle((payload, headers) -> {

            return orderRepo.save(payload);

        })

        …

        .get();

}

При работе с GenericHandler, ссылка на лямбду или метод принимает полезную нагрузку и заголовки сообщений в качестве параметров.  Кроме того, если вы решите использовать GenericHandler в конце потока, вам нужно будет вернуть null, иначе вы получите ошибки, указывающие, что нет указанного выходного канала.

### 9.2.7 Шлюзы

Шлюзы-это средства, с помощью которых приложение может передавать данные в поток интеграции и, при необходимости, получать ответ, который является результатом потока. Реализованные с помощью Spring Integration шлюзы реализуются как интерфейсы, которые приложение может вызывать для отправки сообщений в поток интеграции (рис.9.8).

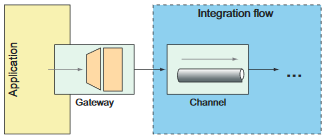


Рисунок 9.8 Сервисные шлюзы - это интерфейсы, через которые приложение может отправлять сообщения в интеграционный поток.

Вы уже видели пример шлюза сообщений со шлюзом FileWriterGateway. FileWriterGateway был односторонним шлюзом с методом, принимающим String для записи в файл, возвращая void.  Примерно так же легко написать двусторонний шлюз. При написании интерфейса шлюза убедитесь, что метод возвращает некоторое значение для публикации в потоке интеграции.

В качестве примера представьте себе шлюз, который находится в простом интеграционном потоке, который принимает String и переводит данную строку в верхний регистр.  Интерфейс шлюза может выглядеть примерно так:

package com.example.demo;

import org.springframework.integration.annotation.MessagingGateway;

import org.springframework.stereotype.Component;

@Component

@MessagingGateway(defaultRequestChannel="inChannel",

        defaultReplyChannel="outChannel")

public interface UpperCaseGateway {

    String uppercase(String in);

}

Что удивительно в этом интерфейсе, так это то, что его не нужно реализовывать. Spring Integration автоматически предоставляет реализацию во время выполнения, которая отправляет и получает данные по указанным каналам.

При вызове метода uppercase() данная String публикуется в потоке интеграции в канал с именем inChannel.  И, независимо от того, как определяется поток или что он делает, когда данные поступают в канал с именем outChannel, он возвращается из метода верхнего uppercase().

Что касается потока интеграции в верхнем регистре, то это упрощенный процесс интеграции с единственным шагом преобразования String в верхний регистр. Опишем в конфигурации Java DSL:

@Bean

public IntegrationFlow uppercaseFlow() {

    return IntegrationFlows

        .from("inChannel")

        .<String, String> transform(s -> s.toUpperCase())

        .channel("outChannel")

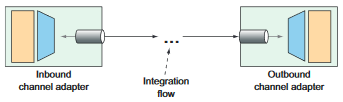
        .get();

}

Здесь описано, поток начинается с данных, поступающих в канал с именем inChannel. Затем полезная нагрузка сообщения преобразуется преобразователем, который здесь определен как лямбда-выражение, для выполнения операции преобразования в верхний регистр. Полученное сообщение затем публикуется в канал с именем outChannel, который вы объявили в качестве канала ответа для интерфейса UpperCaseGateway.

### 9.2.8 Канальные адаптеры

Канальные адаптеры представляют точки входа и выхода потока интеграции. Данные входят в поток интеграции через адаптер входящего канала и выходят из потока интеграции через адаптер исходящего канала. Это показано на рисунке 9.9.



**Рис. 9.9. Адаптеры канала - это точки входа и выхода потока интеграции.**

Адаптеры входящего канала могут принимать различные формы в зависимости от источника данных, вводимых в поток. Например, вы можете объявить адаптер входящего канала, который вводит инкрементные числа из AtomicInteger в поток. При использовании Java конфигурации это может выглядеть так:

@Bean

@InboundChannelAdapter(

    poller=@Poller(fixedRate="1000"), channel="numberChannel")

public MessageSource<Integer> numberSource(AtomicInteger source) {

    return () -> {

        return new GenericMessage<>(source.getAndIncrement());

    };

}

Этот @Bean метод объявляет компонент адаптера входящего канала, который в соответствии с аннотацией @InboundChannelAdapter отправляет число из введенного AtomicInteger в канал с именем numberChannel каждые 1 секунду (или 1000 мс).

Принимая во внимание, что @InboundChannelAdapter указывает адаптер входящего канала при использовании конфигурации Java, метод from() - это то, как это делается при использовании Java DSL для определения потока интеграции. В следующем фрагменте определения потока показан аналогичный адаптер входящего канала, определенный в Java DSL:

@Bean

public IntegrationFlow someFlow(AtomicInteger integerSource) {

    return IntegrationFlows

        .from(integerSource, "getAndIncrement", c -> c.poller(Pollers.fixedRate(1000)))

        …

        .get();

}

Часто адаптеры каналов предоставляются одним из множества модулей конечной точки Spring Integration. Предположим, например, что вам нужен адаптер входящего канала, который отслеживает указанный каталог и отправляет любые файлы, которые записываются в этот каталог, как сообщения в канал с именем file-channel. Следующая конфигурация Java использует FileReadingMessageSource из модуля Spring Integration’s file endpoint для достижения этой цели:

@Bean

@InboundChannelAdapter(channel="file-channel", poller=@Poller(fixedDelay="1000"))

public MessageSource<File> fileReadingMessageSource() {

    FileReadingMessageSource sourceReader = new FileReadingMessageSource();

    sourceReader.setDirectory(new File(INPUT\_DIR));

    sourceReader.setFilter(new SimplePatternFileListFilter(FILE\_PATTERN));

    return sourceReader;

}

При записи эквивалентного адаптера входящего канала для чтения файлов в Java DSL метод inboundAdapter() из класса Files выполняет то же самое. Адаптер внешнего канала - это конец строки для процесса интеграции, передающий окончательное сообщение приложению или какой-либо другой системе:

@Bean

public IntegrationFlow fileReaderFlow() {

    return IntegrationFlows

        .from(Files.inboundAdapter(new File(INPUT\_DIR))

            .patternFilter(FILE\_PATTERN))

        .get();

}

Активаторы сервисов, реализованные как обработчики сообщений, часто служат для адаптера исходящего канала, особенно когда данные должны быть переданы самому приложению. Мы уже обсуждали сервисы активаторы, поэтому нет смысла повторять это обсуждение.

Однако стоит отметить, что модули Spring Integration endpoint предоставляют полезные обработчики сообщений для нескольких распространенных случаев использования. Вы видели пример такого адаптера исходящего канала, FileWritingMessageHandler, в листинге 9.3. Говоря о модулях Spring Integration endpoint, давайте кратко рассмотрим, какие готовые к использованию модули конечных точек интеграции доступны.

### 9.2.9 Модули конечных точек (endpoint)

Замечательно, что Spring Integration позволяет вам создавать свои собственные адаптеры канала. Но еще лучше то, что Spring Integration предоставляет более двух десятков модулей конечных точек, содержащих адаптеры каналов - как входящие, так и исходящие - для интеграции с различными общими внешними системами, в том числе перечисленными в таблице 9.1.

**Таблица 9.1. Spring Integration предоставляет более двух десятков моделей конечных точек для интеграции с внешними системами.**

***Модуль : Идентификатор артефакта зависимости (идентификатор группы: org.springframework.integration)***

**AMQP**: spring-integration-amqp

**Spring application events** : spring-integration-event

**RSS and Atom** : spring-integration-feed

**Filesystem** : spring-integration-file

**FTP/FTPS** : spring-integration-ftp

**GemFire**: spring-integration-gemfire

**HTTP**: spring-integration-http

**JDBC**: spring-integration-jdbc

**JPA**: spring-integration-jpa

**JMS**: spring-integration-jms

**Email**: spring-integration-mail

**MongoDB**: spring-integration-mongodb

**MQTT**: spring-integration-mqtt

**Redis**: spring-integration-redis

**RMI**: spring-integration-rmi

**SFTP**: spring-integration-sftp

**STOMP**: spring-integration-stomp

**Stream**: spring-integration-stream

**Syslog**: spring-integration-syslog

**TCP/UDP** : spring-integration-ip

**Twitter**: spring-integration-twitter

**Web Services** : spring-integration-ws

**WebFlux**: spring-integration-webflux

**WebSocket**: spring-integration-websocket

**XMPP**: spring-integration-xmpp

**ZooKeeper**: spring-integration-zookeeper

Из таблицы 9.1 ясно, что Spring Integration предоставляет обширный набор компонентов для удовлетворения многих потребностей интеграции. Большинству приложений никогда не потребуется даже часть того, что предлагает Spring Integration. Но хорошо знать, что Spring Integration может, на случай если вам вдруг что-то из этого перечня понадобится.

Более того, было бы невозможно охватить все адаптеры каналов, предоставляемые модулями, перечисленными в таблице 9.1, в этой главе. Вы уже видели примеры, которые используют модуль файловой системы для записи в файловую систему. И вы скоро будете использовать модуль электронной почты для чтения электронных писем.

Каждый из модулей конечных точек предлагает канальные адаптеры, которые могут быть либо объявлены как bean-ы при использовании Java конфигурации, либо ссылаться на статические методы при использовании Java DSL конфигурации. Я рекомендую вам изучить любые другие endpoint модули, которые вас интересуют больше всего. Вы обнаружите, что они довольно последовательны в том, как они используются. Но сейчас давайте обратим наше внимание на endpoint модуль электронной почты, чтобы узнать, как вы можете использовать его в приложении Taco Cloud.

**9.3 Создание интеграционного потока электронной почты**

Вы решили, что Taco Cloud должна позволить своим клиентам отправлять свои тако-дизайны и размещать заказы по электронной почте. Вы рассылаете листовки и размещаете рекламу в газетах, предлагая всем отправлять свои заказы тако по электронной почте. Это огромный успех! К сожалению, это слишком успешно. На адрес электронной почты поступает так много писем, что вам нужно нанять отдельного человека, чтобы читать все письма и отправлять детали заказа в систему заказов.

В этом разделе вы реализуете интеграционный поток, который опрашивает почтовый ящик Taco Cloud на наличие электронных писем с заказами, анализирует электронные письма на предмет деталей заказа и отправляет заказы в Taco Cloud для обработки. Короче говоря, поток интеграции, который вам понадобится, будет использовать адаптер входящего канала из модуля конечной точки электронной почты для загрузки электронной почты из папки входящих сообщений Taco Cloud в поток интеграции.

Следующим шагом в процессе интеграции будет разбор электронных писем в объекты заказов, которые передаются другому обработчику для отправки заказов в REST API Taco Cloud, где они будут обрабатываться так же, как и любой заказ. Для начала давайте определим простой класс свойств конфигурации, чтобы отразить особенности обработки электронной почты Taco Cloud:

@Data

@ConfigurationProperties(prefix="tacocloud.email")

@Component

public class EmailProperties {

    private String username;

    private String password;

    private String host;

    private String mailbox;

    private long pollRate = 30000;

    public String getImapUrl() {

        return String.format("imaps://%s:%s@%s/%s",

            this.username, this.password, this.host, this.mailbox);

    }

}

Как вы можете видеть, EmailProperties содержит свойства, которые используются для создания URL-адреса IMAP. Этот поток использует этот URL-адрес для подключения к серверу электронной почты Taco Cloud и опроса электронных писем. К числу свойств относятся имя пользователя и пароль пользователя электронной почты, а также имя хоста сервера IMAP, почтовый ящик для опроса и скорость, с которой опрашивается почтовый ящик (по умолчанию каждые 30 секунд).

Класс EmailProperties аннотируется на уровне класса @ConfigurationProperties с атрибутом prefix, установленным в tacocloud.email. Это означает, что вы можете настроить детали использования электронной почты в файле application.yml следующим образом:

tacocloud:

    email:

        host: imap.tacocloud.com

        mailbox: INBOX

        username: taco-in-flow

        password: 1L0v3T4c0s

        poll-rate: 10000

Теперь давайте воспользуемся EmailProperties для настройки потока интеграции. Поток, который вы создадите, будет немного похож на рисунок 9.10.



1)Email (IMAP)адаптер входящего канала

3)Mail-to-ordertransformer

5)Отправить заказ на адаптер исходящего канала

**Рисунок 9.10 Поток интеграции для приема заказов тако по электронной почте**

У вас есть два варианта определения этого потока:

-Определить внутри приложения Taco Cloud - в конце потока сервис-активатор вызовет репозитории, которые вы определили, чтобы создать заказ тако.

-Определите как отдельное приложение.- В конце потока активатор службы отправит запрос POST в Taco Cloud API для отправки заказа тако.

Что вы выберете, мало влияет на сам поток, помимо того, как реализован сервисный активатор. Но поскольку вам потребуются некоторые типы, которые представляют собой тако, заказы и ингредиенты, которые несколько отличаются от тех, которые вы уже определили в основном приложении Taco Cloud, продолжите определение определять интеграционный поток в отдельном приложении, чтобы избежать путаница с существующими типами доменов.

У вас также есть выбор определения потока с использованием XML конфигурации, Java конфигурации или Java DSL. Мне нравится элегантность Java DSL, так что и вы вслед за мной будете ее использовать. Не стесняйтесь писать поток, используя один из других стилей конфигурации, если вам требуются какие-то дополнительные возможности. А пока давайте взглянем на Java DSL конфигурацию для потока электронной почты заказа тако, как показано ниже.

**Листинг 9.5. Определение потока интеграции для приема электронных писем и отправки их в качестве заказов**

package tacos.email;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.integration.dsl.IntegrationFlow;

import org.springframework.integration.dsl.IntegrationFlows;

import org.springframework.integration.dsl.Pollers;

@Configuration

public class TacoOrderEmailIntegrationConfig {

    @Bean

    public IntegrationFlow tacoOrderEmailFlow(

        EmailProperties emailProps,

        EmailToOrderTransformer emailToOrderTransformer,

        OrderSubmitMessageHandler orderSubmitHandler) {

            return IntegrationFlows

                .from(Mail.imapInboundAdapter(emailProps.getImapUrl()),

                    e -> e.poller(

                        Pollers.fixedDelay(emailProps.getPollRate())))

                .transform(emailToOrderTransformer)

                .handle(orderSubmitHandler)

                .get();

    }

}

Поток электронной почты заказа тако, определенный в методе tacoOrderEmailFlow(), состоит из трех отдельных компонентов:

-Адаптер входящего канала электронной почты IMAP - этот адаптер канала создается с помощью URL-адреса IMP, созданного методом getImapUrl() объекта EmailProperties, и опрашивает с задержкой, установленной в свойстве pollRate объекта EmailProperties. Входящие письма передаются по каналу, соединяющему его с трансформатором.

-Трансформатор, который преобразует электронную почту в объект заказа - Преобразователь реализуется в EmailToOrderTransformer, который внедряется в метод tacoOrderEmailFlow(). Заказы, полученные в результате преобразования, передаются конечному компоненту через другой канал.

-Обработчик (выступающий в качестве адаптера исходящего канала)- Обработчик принимает объект заказа и отправляет его по REST API Taco Cloud.

Вызов Mail.imapInboundAdapter() стал возможен благодаря включению endpoint модуля электронной почты в качестве зависимости в сборку проекта. Зависимость Maven выглядит так:

<dependency>

    <groupId>org.springframework.integration</groupId>

    <artifactId>spring-integration-file</artifactId>

</dependency>

Класс EmailToOrderTransformer является реализацией интерфейса Spring Integration Transformer посредством расширения AbstractMailMessageTransformer (показано в следующем листинге).

**Листинг 9.6. Преобразование входящих писем в тако-заказы с использованием интеграционного преобразователя**

@Component

public class EmailToOrderTransformer

        extends AbstractMailMessageTransformer<Order> {

    @Override

    protected AbstractIntegrationMessageBuilder<Order> doTransform(Message mailMessage)

            throws Exception {

        Order tacoOrder = processPayload(mailMessage);

        return MessageBuilder.withPayload(tacoOrder);

    }

    …

}

AbstractMailMessageTransformer - удобный базовый класс для обработки сообщений, чья полезная нагрузка - это электронная почта. Он заботится о извлечении информации электронной почты из входящего сообщения в объект Message, который передается в метод doTransform().

В методе doTransform() вы передаете Message private методу с именем processPayload() для анализа электронной почты на получения объекта Order. Несмотря на то, что этот объект Order не похож на объект Order, используемый в основном приложении TacoCloud; это немного проще:

package tacos.email;

import java.util.ArrayList;

import java.util.List;

import lombok.Data;

@Data

public class Order {

    private final String email;

    private List<Taco> tacos = new ArrayList<>();

    public void addTaco(Taco taco) {

        this.tacos.add(taco);

    }

}

Вместо того, чтобы содержать всю информацию о доставке и выставлении счетов клиентам, этот класс Order несет только электронную почту клиента, полученную из входящей электронной почты.

Парсинг электронных писем в тако-заказы является нетривиальной задачей. На самом деле, даже простенькая реализация включает в себя несколько десятков строк кода. И эти несколько десятков строк кода ничего не дадут для дальнейшего обсуждения Spring Integration и того, как реализовать трансформер. Поэтому, чтобы сэкономить место, я опускаю детали метода processPayload().

Последнее, что делает EmailToOrderTransformer, это возвращает MessageBuilder с полезной нагрузкой, содержащей объект Order. Сообщение, сгенерированное MessageBuilder, отправляется последнему компоненту в потоке интеграции: обработчику сообщений, который отправляет заказ в Taco Cloud API. OrderSubmitMessageHandler, показанный в следующем листинге, реализует GenericHandler в Spring Integration для обработки сообщений с полезной нагрузкой Order.

**Листинг 9.7. Отправка заказов в Taco Cloud API через обработчик сообщений**

package tacos.email;

import java.util.Map;

import org.springframework.integration.handler.GenericHandler;

import org.springframework.stereotype.Component;

import org.springframework.web.client.RestTemplate;

@Component

public class OrderSubmitMessageHandler implements GenericHandler<Order> {

    private RestTemplate rest;

    private ApiProperties apiProps;

    public OrderSubmitMessageHandler(

            ApiProperties apiProps, RestTemplate rest) {

        this.apiProps = apiProps;

        this.rest = rest;

    }

    @Override

    public Object handle(Order order, Map<String, Object> headers) {

        rest.postForObject(apiProps.getUrl(), order, String.class);

        return null;

    }

}

Чтобы удовлетворить требования интерфейса GenericHandler, OrderSubmitMessageHandler переопределяет метод handle(). Этот метод получает входящий объект Order и использует внедренный RestTemplate для отправки Order посредством POST запроса на URL-адрес, полученный из объекте ApiProperties. Наконец, метод handle() возвращает null, чтобы указать, что этот обработчик отмечает конец потока.

ApiProperties используется, чтобы избежать жесткого кодирования URL-адреса при вызове postForObject(). Это файл свойств конфигурации, который выглядит следующим образом:

@Data

@ConfigurationProperties(prefix="tacocloud.api")

@Componentpublic class ApiProperties {

    private String url;

}

А в application.yml URL-адрес для Taco Cloud API может быть настроен следующим образом:

tacocloud:

    api:

        url: [http://api.tacocloud.com](http://api.tacocloud.com/)

Чтобы сделать RestTemplate доступным в проекте, чтобы его можно было внедрить в OrderSubmitMessageHandler, необходимо добавить веб-стартер Spring Boot в сборку проекта:

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-web</artifactId>

</dependency>

В то время как это делает RestTemplate доступным в classpath, оно также запускает автоконфигурирование для Spring MVC. В качестве автономного потока Spring Integration приложению не требуется Spring MVC или даже встроенный Tomcat, который обеспечивает автоконфигурация. Поэтому вам следует отключить автоконфигурирование Spring MVC с помощью следующей записи в application.yml:

spring:

    main:

        web-application-type: none

Свойство spring.main.web-application-type может быть установлено в servlet, reactive или none. Когда Spring MVC находится в classpath, автоконфигурация устанавливает его значение в servlet. Но здесь вы переопределяете его на none, чтобы Spring MVC и Tomcat не были автоматически сконфигурированы. (Мы поговорим подробнее о том, что означает, что приложение является reactive веб-приложением в главе 11.)

### ИТОГ:

-Spring Integration позволяет определять потоки, через которые данные могут обрабатываться при их поступлении в приложение или на выходе из него.

-Интеграционные потоки могут быть определены в XML, Java или с использованием краткого стиля конфигурации Java DSL.

-Шлюзы сообщений и адаптеры каналов действуют как точки входа и выхода интеграционного потока.

-Сообщения могут быть преобразованы, разделены, агрегированы, направлены и обработаны активаторами службы в ходе потока.

-Каналы сообщений соединяют компоненты потока интеграции.

[Spring in Action Covers Spring 5.0 перевод на русский. Глава 10](https://beginner-mercenary.blogspot.com/2019/10/spring-in-action-covers-spring-50-10.html)

**РАЗДЕЛ 3**

**Реактивный Spring**

В разделе 3, мы рассмотрим новую замечательную поддержку реактивного программирования в Spring. В главе 10 обсуждаются основы реактивного программирования с Project Reactor, библиотекой реактивного программирования, которая лежит в основе реактивных функций Spring 5. Затем мы рассмотрим некоторые из наиболее полезных реактивных операций Reactor. В главе 11 мы вернемся к разработке REST API, представив Spring WebFlux, новую веб-инфраструктуру, которая позаимствовала многое из Spring MVC и предлагает новую реактивную модель для веб-разработки. В главе 12 завершается третья часть, в которой описывается постоянное сохранение реактивных данных с помощью Spring Data для чтения и записи данных в базы данных Cassandra и Mongo.

**10. Знакомство с Reactor**

*В этой главе рассматривается*

*-Понимание реактивного программирования*

*-Project Reacto*

*-Реактивная работа с данными*

Вы когда-нибудь имели подписку на газету или журнал? Интернет определенно уменьшил число подписчиков традиционных публикаций, но было время, когда подписка на газеты была одним из лучших способов быть в курсе событий дня. Вы можете рассчитывать на свежую доставку информации о текущих событиях каждое утро, чтобы читать во время завтрака или по дороге на работу.

Теперь предположим, что если после оплаты вашей подписки прошло несколько дней, и издания не были доставлены. Проходит еще несколько дней, и вы звоните в отдел продаж, чтобы узнать, почему вы еще не получили ежедневную газету. Представьте свой сюрприз, если они объяснят: «Вы заплатили за целый год газет. Год еще не закончен. Вы обязательно получите их все, как только будет готово полное годовое издание газет».

К счастью, это совсем не то, как работают подписки. Газеты имеют определенную периодичность. Они доставляются как можно быстрее после публикации, чтобы их можно было прочитать, пока их содержание еще свежо. Более того, когда вы читаете последний выпуск, газетные репортеры пишут новые истории для будущих изданий, и прессы запускаются для выпуска следующего выпуска - все параллельно.

Поскольку мы разрабатываем код приложения, мы можем написать два стиля кода: императивный и реактивный:

* Императивный код очень похож на эту абсурдную гипотетическую подписку на газету. Это последовательный набор задач, каждая из которых выполняется по одной, каждая после предыдущей. Данные обрабатываются в большом количестве и не могут быть переданы следующей задаче, пока предыдущая задача не завершила свою работу с массивом данных.
* Реактивный код очень похож на настоящую подписку на газету. Набор задач определен для обработки данных, но эти задачи могут выполняться параллельно. Каждая задача может обрабатывать подмножества данных, передавая их следующей задаче в очереди, пока она продолжает работать с другим подмножеством данных.

В этой главе мы временно отойдем от приложения Taco Cloud, чтобы изучить Project Reactor. Reactor - это библиотека для реактивного программирования, которая является частью семейства проектов Spring. И поскольку он служит основой поддержки реактивного программирования в Spring 5, важно, чтобы вы поняли Reactor, прежде чем мы рассмотрим создание реактивных контроллеров и репозиториев с помощью Spring. Прежде чем мы начнем работать с Reactor, давайте быстро рассмотрим основы реактивного программирования.

**10.1 Понимание реактивного программирования**

Реактивное программирование - это парадигма, альтернативная императивному программированию. Эта альтернатива существует, потому что реактивное программирование устраняет ограничение в императивном программировании. Понимая эти ограничения, вы можете лучше понять преимущества реактивной модели.

Обратите внимание, что реактивное программирование - это не серебряная пуля. Ни в коем случае не следует выводить из этой главы или любого другого обсуждения реактивного программирования, что императивное программирование-это зло, а реактивное программирование-ваш спаситель. Как и все, что вы изучаете как разработчик, реактивное программирование идеально подходит в некоторых случаях использования, а в других плохо подходит. Рекомендуется унция прагматизма.

Если вы похожи на меня и многих разработчиков, вы режете свои программные зубы императивным программированием. Есть хороший шанс, что большинство (или весь) код, который вы пишете сегодня, по-прежнему является императивным по своей природе. Императивное программирование достаточно интуитивно, что молодые студенты с легкостью изучают его в своих школьных программах STEM, и оно достаточно мощное, чтобы составлять основную часть кода, который используется во многих крупнейших предприятия.

Идея проста: вы пишете код в виде списка инструкций, которым нужно следовать, по одному в том порядке, в котором они встречаются. Задача выполнена, и программа ожидает ее завершения, прежде чем перейти к следующей задаче. На каждом этапе обработки данные, которые должны быть обработаны, должны быть полностью доступны, чтобы их можно было обрабатывать целиком.

Во время выполнения задачи, особенно если это задача I/O, такая как запись данных в базу данных или выборка данных с удаленного сервера, поток, вызвавший эту задачу, блокируется и не может ничего сделать, пока задача не завершится. Проще говоря, заблокированные потоки расточительны.

Большинство языков программирования, включая Java, поддерживают параллельное программирование. Довольно просто запустить другой поток в Java и отправить его на выполнение некоторой работы, в то время как вызывающий поток продолжает что-то еще. Но хотя создавать потоки легко, эти потоки, скорее всего, сами блокируются. Управление параллелизмом в нескольких потоках является сложной задачей. Больше потоков означает больше сложности.

Напротив, реактивное программирование носит функциональный и декларативный характер. Вместо описания набора шагов, которые должны выполняться последовательно, реактивное программирование включает описание конвейера или потока, через который проходят данные. Вместо того чтобы требовать, чтобы данные были доступны для обработки в целом, реактивный поток обрабатывает данные по мере их поступления. Фактически, поступающие данные могут быть бесконечными (например, постоянный поток данных о температуре в реальном времени).

Чтобы применить аналогию с реальным миром, рассмотрим императивное программирование как водяной шар и реактивное программирование как садовый шланг. Оба являются подходящими способами удивить и замочить ничего не подозревающего друга в жаркий летний день. Но они отличаются по стилю исполнения:

-Воздушный шар воды несет свою полезную нагрузку всю сразу, замачивая свою намеченную цель в момент удара. Однако водяной шар имеет конечную емкость, и если вы хотите замочить больше людей (или одного и того же человека в большей степени), ваш единственный выбор-увеличить количество водных шаров.

-Садовый шланг несет свою полезную нагрузку в виде потока воды, который течет от крана к соплу. Емкость садового шланга может быть конечной в любой момент времени, но она неограничена в течение водного боя. Пока вода поступает в шланг из крана, она будет продолжать течь через шланг и распыляться из сопла. Тот же садовый шланг легко масштабируется, чтобы замочить столько друзей, сколько вы хотите.

Нет ничего изначально плохого в водяных шарах (или императивном программировании), но человек, держащий садовый шланг (или применяющий реактивное программирование), имеет преимущество в отношении масштабируемости и производительности.

**10.1.1 Определение реактивных потоков**

Reactive Streams - инициатива, начатая в конце 2013 года инженерами из Netflix, Lightbend и Pivotal (компания, стоящая за Spring). Reactive Streams стремится обеспечить стандарт для асинхронной обработки потока с неблокирующим обратным давлением (backpressure).

Мы уже затронули асинхронную черту реактивного программирования; это то, что позволяет нам выполнять задачи параллельно для достижения большей масштабируемости. Противодавление - это средство, с помощью которого потребители данных могут избежать перегруженности слишком быстрым источником данных, устанавливая ограничения на то, сколько они готовы обработать.

**Java Streams vs. Reactive Streams**

Существует много общего между Java Streams и Reactive Streams. Для начала, они оба имеют слово потоки в своих именах. Они также предоставляют функциональный API для работы с данными. На самом деле, как вы увидите позже, когда мы посмотрим на Reactor, они даже выполняют одни и те же операции.

Однако Java Streams обычно являются синхронными и работают с конечным набором данных. По сути, они являются средством перебора коллекции с помощью функций.

Reactive Streams поддерживают асинхронную обработку наборов данных любого размера, включая бесконечные наборы данных. Они обрабатывают данные в режиме реального времени, когда они становятся доступными, с противодавлением (backpressure), чтобы не перегружать своих потребителей.

Спецификация реактивных потоков может быть описана четырьмя определениями интерфейсами: Publisher, Subscriber, Subscription, и Processor. Publisher создает данные, которые он отправляет Subscriber на Subscription. Интерфейс Publisher объявляет единый метод subscribe() с помощью которого Subscriber может подписаться на Publisher:

public interface Publisher<T> {

    void subscribe(Subscriber<? super T> subscriber);

}

После того как Subscriber подписался, он может получать события от Publisher. Эти события отправляются через методы интерфейса Subscriber:

public interface Subscriber<T> {

    void onSubscribe(Subscription sub);

    void onNext(T item);

    void onError(Throwable ex);

    void onComplete();

}

Первое событие, которое получит Subscriber, - это вызов функции onSubscribe(). Когда Publisher вызывает функцию onSubscribe(), он передает Subscription объект Subscriber. Именно через Subscription Subscriber может управлять своей подпиской:

public interface Subscription {

    void request(long n);

    void cancel();

}

Subscriber может вызвать функцию request(), чтобы запросить отправку данных, или функцию cancel(), чтобы указать, что он больше не заинтересован в получении данных и отменяет подписку. При вызове функции request() Subscriber передает long значение, чтобы указать, сколько элементов данных он готов принять. Именно здесь возникает обратное давление (backpressure), препятствующее Publisher отправлять больше данных, чем может обработать Subscriber. После того, как Publisher отправил столько элементов, сколько было запрошено, Subscriber может снова вызвать функцию request(), чтобы запросить больше.

После того, как Subscriber запросил данные, данные начинают поступать через поток. Для каждого элемента, опубликованного Publisher, будет вызван метод onNext() для доставки данных Subscriber. Если есть какие-либо ошибки, вызывается onError(). Если у Publisher нет больше данных для отправки и он не будет генерировать больше данных, он вызовет onComplete(), чтобы сообщить подписчику, что он завершил процесс.

Что касается интерфейса Processor, это комбинация Subscriber и Publisher, как показано здесь:

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {}

Как Subscriber, Processor будет получать данные и обрабатывать их каким-либо образом. Затем он будет “переоденется” и выступит в качестве Publisher, чтобы публиковать результаты для своих Subscribers.

Как вы можете видеть, спецификация Reactive Streams довольно проста. Довольно легко понять, как можно построить конвейер обработки данных, который начинается с Publisher, прогоняет данные через ноль или более Processors, а затем передает конечные результаты в Subscriber.

Однако интерфейсы Reactive Streams не могут использоваться для создания такого потока функциональным способом. Project Reactor - это реализация спецификации Reactive Streams, которая предоставляет функциональный API для создания Reactive Streams. Как вы увидите в следующих главах, Reactor является основой модели реактивного программирования в Spring 5. В оставшейся части этой главы мы собираемся исследовать (и, осмелюсь сказать, очень весело провести время) Project Reactor.

**10.2 Начало работы с Reactor**

Реактивное программирование требует от нас думать совсем иначе, чем императивное программирование. Вместо того, чтобы описывать набор шагов, которые необходимо предпринять, реактивное программирование означает построение конвейера, по которому будут проходить данные. Когда данные проходят через конвейер, они могут быть изменены или использованы каким-либо образом.

Например, предположим, что вы хотите взять имя человека, изменить все его буквы на заглавные, использовать его для создания приветственного сообщения, а затем, наконец, напечатать его. В модели императивного программирования код будет выглядеть примерно так:

String name = "Craig";

String capitalName = name.toUpperCase();

String greeting = "Hello, " + capitalName + "!";

System.out.println(greeting);

В императивной модели каждая строка кода выполняет шаг, один за другим, и определенно в одном и том же потоке. Каждый шаг блокирует выполнение потока до следующего шага до его завершения.

Функциональный, реактивный код может достичь того же этого же:

Mono.just("Craig")

   .map(n -> n.toUpperCase())

   .map(cn -> "Hello, " + cn + "!")

   .subscribe(System.out::println);

Не волнуйтесь про, возможное, недопонимание этого примера; мы скоро поговорим об операциях just(), map() и subscribe(). На данный момент важно понимать, что хотя реактивный пример все еще следует пошаговой модели, на самом деле это конвейер, через который проходят данные. На каждом этапе конвейера данные каким-то образом изменяются, но нельзя точно сказать, какой поток выполняет какие операции. Все операции могут выполняться в одном потоке ... или не в одном.

Mono в этом примере является одним из двух основных типов Reactor. Второй это - Flux. Оба являются реализациями Reactive Streams Publisher. Поток представляет собой конвейер из нуля, одного или многих (потенциально бесконечных) элементов данных. Mono - это специализированный реактивный тип, оптимизированный для случаев, когда известно, что в наборе данных содержится не более одного элемента данных.

Reactor vs. RxJava (ReactiveX)

Если вы уже знакомы с RxJava или ReactiveX, возможно, вы думаете, что Mono и Flux звучат во многом как Observable и Single. На самом деле, они примерно эквивалентны семантически. Они даже предлагают много одинаковых операций.

Хотя в этой книге мы сосредоточимся на Reactor, вы, возможно, будете рады узнать, что можно скрывать типы Reactor и RxJava. Более того, как вы увидите в следующих главах, Spring также может работать с типами RxJava.

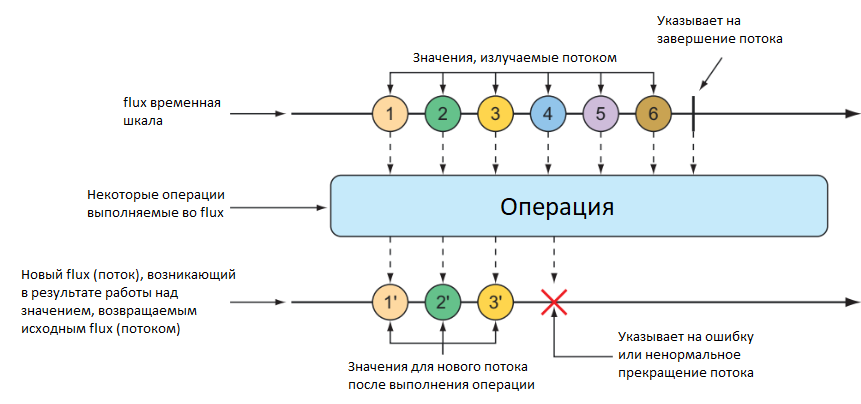
На самом деле в предыдущем примере три Mono. Операция just() создает первую. Когда Mono выдает значение, это значение присваивается операции map(), которая описывает что слово должно быть написано заглавными буквами и использовано для создания другого Mono. Когда второй Mono публикует свои данные, он передается второй операции map() для выполнения конкатенации строк, результаты которой используются для создания третьего Mono. Наконец, вызов метода subscribe() подписывается на Mono, получает данные и печатает их.

**10.2.1 Схема реактивных потоков**

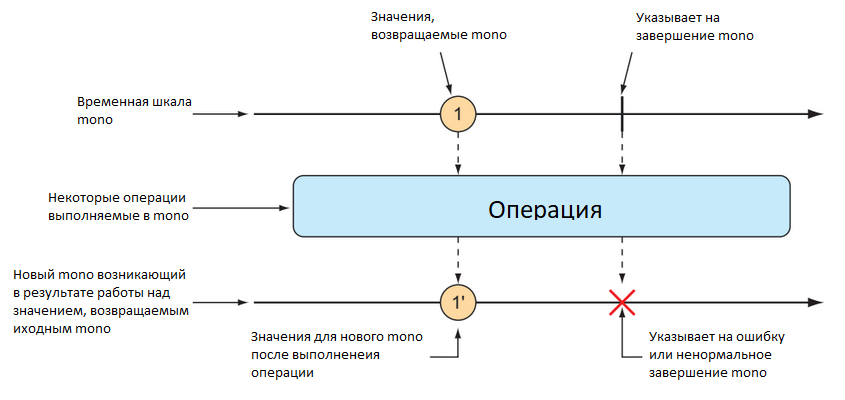
Реактивные потоки часто иллюстрируются мраморными (marble) диаграммами. Мраморные (marble) диаграммы в своей простейшей форме изображают временную шкалу данных, потоков проходящих через Flux или Mono вверху, операцию в середине и временную шкалу результирующего Flux или Mono внизу. На рис. 10.1 показан шаблон диаграммы состояния потока. Как вы можете видеть, когда данные проходят через исходный Flux, он обрабатывается через некоторую операцию, в результате чего возникает новый Flux, через который проходят обработанные данные.

На рисунке 10.2 показана аналогичная мраморная (marble) диаграмма, но для Mono. Как видите, ключевое отличие состоит в том, что у Mono будет либо ноль, либо один элемент данных, либо ошибка.

В разделе 10.3 мы рассмотрим многие операции, поддерживаемые Flux и Mono, и будем использовать мраморные (marble) диаграммы для визуализации их работы.



**Рисунок 10.1 Мраморная (marble) диаграмма, иллюстрирующая основной поток Flux**



**Рисунок 10.2 Мраморная (marble) диаграмма, иллюстрирующая основной поток Mono**

**10.2.2 Добавление Reactor зависимостей**

Чтобы начать работу с Reactor, добавьте следующую зависимость в сборку проекта:

<dependency>

   <groupId>io.projectreactor</groupId>

   <artifactId>reactor-core</artifactId>

</dependency>

Reactor также предоставляет отличную поддержку тестирования. Мы собираемся написать множество тестов для своего кода Reactor, поэтому вам обязательно нужно добавить эту зависимость в вашу сборку:

<dependency>

   <groupId>io.projectreactor</groupId>

   <artifactId>reactor-test</artifactId>

   <scope>test</scope>

</dependency>

Я предполагаю, что вы добавляете эти зависимости в проект Spring Boot, который обрабатывает управление зависимостями (dependency management) для вас, поэтому нет необходимости указывать элемент <version> для зависимостей. Но если вы хотите использовать Reactor в проекте, отличном от Spring Boot, вам потребуется настроить спецификацию Reactor BOM (bill of materials) в сборке. Следующая запись управления зависимостями добавляет Reactor Bismuth в сборку:

<dependencyManagement>

   <dependencies>

      <dependency>

         <groupId>io.projectreactor</groupId>

         <artifactId>reactor-bom</artifactId>

         <version>Bismuth-RELEASE</version>

         <type>pom</type>

         <scope>import</scope>

      </dependency>

   </dependencies>

</dependencyManagement>

Теперь, когда Reactor находится в разработке вашего проекта, вы можете начать создавать реактивные конвейеры с Mono и Flux. В оставшейся части этой главы мы рассмотрим несколько операций, предлагаемых Mono и Flux.

**10.3 Применение общих реактивных операций**

Flux и Mono являются наиболее важными строительными блоками, предоставляемыми Reactor, и операции, предлагаемые этими двумя реактивными типами, представляют собой раствор, который связывает их вместе для создания конвейеров, по которым могут передаваться данные. Между Flux и Mono существует более 500 операций, каждую из которых можно условно классифицировать как

-Операции создания

-Комбинированные операции

-Операции трансформации

-Логические операции

Как бы ни было интересно смотреть на каждую из 500 операций, чтобы увидеть, как они работают, в этой главе просто не хватит места. В этом разделе я выбрал несколько наиболее полезных операций для экспериментов. Начнем с операций создания.

***ПРИМЕЧАНИЕ***

Где примеры Mono? Mono и Flux используют много одинаковых операций, поэтому в большинстве случаев нет необходимости показывать одну и ту же операцию дважды, один раз для Mono и еще раз для Flux. Более того, хотя операции Mono полезны, на них немного менее интересно смотреть, чем на те же самые операции, когда предоставляется Flux. Большинство примеров, с которыми мы будем работать, будут связаны с Flux. Просто знайте, что Mono обычно имеет эквивалентные операции.

**10.3.1 Реактивные типы создания**

Часто при работе с реактивными типами в Spring вы получаете Flux или Mono из репозитория или службы, поэтому вам не нужно создавать их самостоятельно. Но иногда вам нужно будет создать нового реактивного издателя.

Reactor предоставляет несколько операций для создания Flux и Mono. В этом разделе мы рассмотрим некоторые из наиболее полезных операций создания.

***СОЗДАНИЕ ИЗ ОБЪЕКТОВ***

Если у вас есть один или несколько объектов, из которых вы хотите создать Flux или Mono, вы можете использовать статический метод just() в Flux или Mono для создания реактивного типа, данные которого управляются этими объектами. Например, следующий метод тестирования создает Flux из пяти объектов String:

@Test

public void createAFlux\_just() {

   Flux<String> fruitFlux = Flux

      .just("Apple", "Orange", "Grape", "Banana", "Strawberry");

}

На данный момент Flux создан, но у него нет подписчиков. Без подписчиков данные не будут передаваться. Думая об аналогии с садовым шлангом, вы прикрепили садовый шланг к патрубку, и с другой стороны есть вода из коммунальной компании - но пока вы не включите патрубок, вода не будет течь. Подписка на реактивный тип - это то, как вы включаете поток данных.

Чтобы добавить подписчика, вы можете вызвать метод subscribe() в Flux:

fruitFlux.subscribe(

   f -> System.out.println("Here's some fruit: " + f)

);

Здесь лямбда, заданная для subscribe(), на самом деле является java.util.Consumer, который используется для создания Reactive Streams Subscriber. При вызове subscribe() данные начинают передаваться. В этом примере промежуточных операций нет, поэтому данные передаются напрямую от Flux к Subscriber.

Печать записей из Flux или Mono на консоль - это хороший способ увидеть реактивный тип в действии. Но лучший способ на самом деле протестировать Flux или Mono - использовать StepVerifier от Reactor. С учетом Flux или Mono, StepVerifier подписывается на реактивный тип и затем применяет утверждения к данным, когда они проходят через поток, в конце концов проверяя, что поток завершается должным образом.

Например, чтобы проверить, что предписанные данные проходят через fruitFlux, вы можете написать тест, который выглядит следующим образом:

StepVerifier.create(fruitFlux)

   .expectNext("Apple")

   .expectNext("Orange")

   .expectNext("Grape")

   .expectNext("Banana")

   .expectNext("Strawberry")

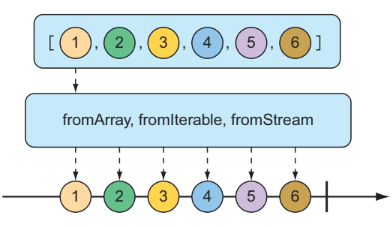
   .verifyComplete();

В этом случае StepVerifier подписывается на Flux и затем утверждает, что каждый элемент соответствует ожидаемому названию плода. Наконец, он проверяет, что после того, как клубника произведена Flux, Flux завершается.

В оставшейся части примеров в этой главе вы будете использовать StepVerifier для написания обучающих тестов - тестов, которые проверяют поведение и помогают понять, как что-то работает, - чтобы узнать некоторые из наиболее полезных операций Reactor.

***СОЗДАНИЕ ИЗ КОЛЛЕКЦИЙ***

Flux также может быть создан из массива, Iterable или Java Stream. Рисунок 10.3 иллюстрирует, как это работает с мраморной диаграммой.



**Рисунок 10.3. Поток может быть создан из массива, Iterable или Stream.**

Чтобы создать Flux из массива, вызовите статический метод fromArray(), передав в исходный массив:

@Test

public void createAFlux\_fromArray() {

   String[] fruits = new String[] {

      "Apple", "Orange", "Grape", "Banana", "Strawberry" };

   Flux<String> fruitFlux = Flux.fromArray(fruits);

   StepVerifier.create(fruitFlux)

      .expectNext("Apple")

      .expectNext("Orange")

      .expectNext("Grape")

      .expectNext("Banana")

      .expectNext("Strawberry")

      .verifyComplete();

}

Поскольку исходный массив содержит те же имена фруктов, которые вы использовали при создании Flux из списка объектов, данные, передаваемые Flux, будут иметь те же значения. Таким образом, вы можете использовать тот же StepVerifier, что и раньше, чтобы проверить этот Flux.

Если вам нужно создать Flux из java.util.List, java.util.Set или любой другой реализации java.lang.Iterable, вы можете передать его в статический метод fromIterable():

@Test

public void createAFlux\_fromIterable() {

   List<String> fruitList = new ArrayList<>();

   fruitList.add("Apple");

   fruitList.add("Orange");

   fruitList.add("Grape");

   fruitList.add("Banana");

   fruitList.add("Strawberry");

   Flux<String> fruitFlux = Flux.fromIterable(fruitList);

   // ... проверить шаги

}

Или, если у вас есть Java Stream, который вы хотели бы использовать в качестве источника для Flux, fromStream() - это метод, который вы будете использовать:

@Test

public void createAFlux\_fromStream() {

   Stream<String> fruitStream =

      Stream.of("Apple", "Orange", "Grape", "Banana", "Strawberry");

   Flux<String> fruitFlux = Flux.fromStream(fruitStream);

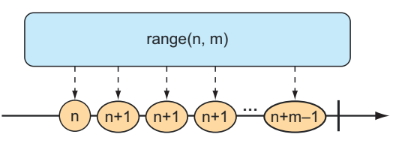
   // ... проверить шаги

}

Опять же, тот же StepVerifier, что и раньше, можно использовать для проверки данных, опубликованных Flux.

***ГЕНЕРИРОВАНИЕ FLUX ДАННЫХ***

Иногда у вас нет данных для работы, и вам просто нужно, чтобы Flux действовал как счетчик, отдавая число, которое увеличивается с каждым новым значением. Для создания счетчика Flux вы можете использовать статический метод range(). Диаграмма на рисунке 10.4 иллюстрирует, как работает range().



**Рисунок 10.4 Создание Flux из диапазона приводит к публикации сообщений в противоположном стиле.**

Следующий метод тестирования демонстрирует, как создать Flux диапазон:

@Test

public void createAFlux\_range() {

   Flux<Integer> intervalFlux =

   Flux.range(1, 5);

   StepVerifier.create(intervalFlux)

      .expectNext(1)

      .expectNext(2)

      .expectNext(3)

      .expectNext(4)

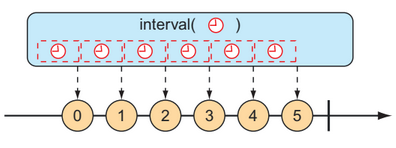
      .expectNext(5)

      .verifyComplete();

}

В этом примере диапазон Flux создается с начальным значением 1 и конечным значением 5. StepVerifier доказывает, что он опубликует пять элементов, которые являются целыми числами от 1 до 5.

Еще один метод создания Flux, похожий на range(), представляет собой interval(). Как и метод range(), interval() создает поток, который возвращает увеличивающееся значение. Но то, что делает interval() особенным, заключается в том, что вместо того, чтобы указывать начальное и конечное значение, вы указываете длительность или как часто значение должно возвращаться. На рисунке 10.5 показана мраморная диаграмма для метода создания interval().



**Рисунок 10.5 Flux, созданный из интервала, имеет периодическую запись, опубликованную в нем. (A Flux created from an interval has a periodic entry published to it.)**

Например, чтобы создать поток интервалов, который выдает значение каждую секунду, можно использовать статический метод interval() следующим образом:

@Test

public void createAFlux\_interval() {

   Flux<Long> intervalFlux =

   Flux.interval(Duration.ofSeconds(1))

      .take(5);

   StepVerifier.create(intervalFlux)

      .expectNext(0L)

      .expectNext(1L)

      .expectNext(2L)

      .expectNext(3L)

      .expectNext(4L)

      .verifyComplete();

}

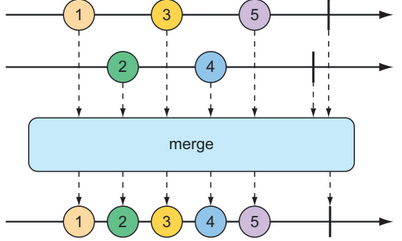
Обратите внимание,что значение, возвращаемое потоком интервалов, начинается с 0 и увеличивается на каждом последующем элементе. Кроме того, поскольку interval() не имеет максимального значения, он потенциально будет работать вечно. Поэтому вы также используете операцию take(), чтобы ограничить результаты первыми пятью записями. Подробнее об операции take() мы поговорим в следующем разделе.

**10.3.2 Комбинирование реактивных типов**

Однажды может возникнуть задача когда Вам придется работать с двумя реактивными типами, которые вам нужно как-то объединить. Или, в других случаях, вам может потребоваться разделить Flux на несколько реактивных типов. В этом разделе мы рассмотрим операции, которые объединяют и разделяют Flux и Mono в Reactor.

***СЛИЯНИЕ РЕАКТИВНЫХ ТИПОВ***

Предположим, у вас есть два потока потока и нужно создать один результирующий поток, который будет производить данные, как только он станет доступен из любого из вышерасположенных Flux streams. Чтобы объединить один поток с другим, можно использовать операцию mergeWith(), как показано на marble диаграмме на рис.10.6.



**Рисунок 10.6 слияние двух Flux потоков чередя их сообщения в новом Flux.**

Например, предположим, что у вас есть Flux, значения которого являются именами телевизионных и киношных персонажей, и у вас есть второй Flux, значениями которого являются названия продуктов, которые эти персонажи любят есть. Следующий тестовый метод показывает, как можно объединить два объекта Flux с помощью метода mergeWith():

@Test

public void mergeFluxes() {

   Flux<String> characterFlux = Flux

      .just("Garfield", "Kojak", "Barbossa")

      .delayElements(Duration.ofMillis(500));

   Flux<String> foodFlux = Flux

      .just("Lasagna", "Lollipops", "Apples")

      .delaySubscription(Duration.ofMillis(250))

      .delayElements(Duration.ofMillis(500));

   Flux<String> mergedFlux = characterFlux.mergeWith(foodFlux);

   StepVerifier.create(mergedFlux)

      .expectNext("Garfield")

      .expectNext("Lasagna")

      .expectNext("Kojak")

      .expectNext("Lollipops")

      .expectNext("Barbossa")

      .expectNext("Apples")

      .verifyComplete();

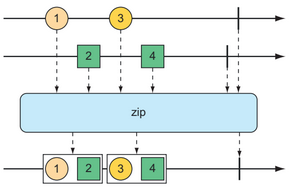
}

Обычно Flux публикует данные настолько быстро, насколько это возможно. Таким образом, вы используете операцию delayElements() в обоих созданных потоках Flux, чтобы немного их замедлить - отправляя запись каждые 500 мс. Кроме того, чтобы поток продуктов начинал передаваться, после Flux имен, вы применяете операцию delaySubscription() к потоку продуктов, чтобы он не отправлял никаких данных, пока не пройдет 250 мс после подписки.

После объединения двух объектов Flux создается новый объединенный Flux. Когда StepVerifier подписывается на объединенный поток, он, в свою очередь, подписывается на два исходных Flux потока, начиная поток данных.

Порядок предметов, отдаваемый из объединенного потока, совпадает со временем их передачи из источников. Поскольку оба объекта Flux настроены на отдачу с регулярной скоростью, значения будут чередоваться через объединенный поток, в результате чего будет получено имя, затем пища, затем имя и т. Д. Если время либо Flux должно быть изменены, возможно, вы увидите два персонажа или два продукта, опубликованные один за другим.

Поскольку mergeWith() не может гарантировать идеальное взаимодействие между его источниками, вы можете рассмотреть операцию zip() вместо этого. Когда два объекта Flux сжимаются вместе, это приводит к новому Flux, который создает кортеж элементов, где кортеж содержит один элемент из каждого исходного потока. На рис. 10.7 показано, как два объекта Flux можно сжать вместе.



**Рис. 10.7. Сжатие двух потоков Flux приводит к созданию Flux, содержащего наборы по одному элементу от каждого потока.**

Чтобы увидеть действие zip() в действии, рассмотрите следующий метод тестирования, который объединяет Flux персонажей и Flux продукты вместе:

@Test

public void zipFluxes() {

   Flux<String> characterFlux = Flux

      .just("Garfield", "Kojak", "Barbossa");

   Flux<String> foodFlux = Flux

      .just("Lasagna", "Lollipops", "Apples");

   Flux<Tuple2<String, String>> zippedFlux =

   Flux.zip(characterFlux, foodFlux);

   StepVerifier.create(zippedFlux)

      .expectNextMatches(p ->

         p.getT1().equals("Garfield") &&

         p.getT2().equals("Lasagna"))

      .expectNextMatches(p ->

         p.getT1().equals("Kojak") &&

         p.getT2().equals("Lollipops"))

      .expectNextMatches(p ->

         p.getT1().equals("Barbossa") &&

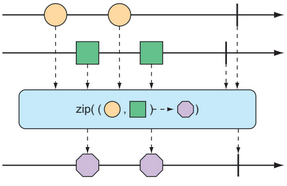
         p.getT2().equals("Apples"))

      .verifyComplete();

}

Обратите внимание, что в отличие от mergeWith(), операция zip() является статической операцией создания. Созданный Flux имеет идеальное выравнивание между персонажами и их любимыми блюдами. Каждый элемент, испускаемый из сжатого потока, представляет собой Tuple2 (контейнерный объект, который содержит два других объекта), содержащий элементы из каждого исходного потока в порядке их публикации.

Если вы предпочитаете не работать с Tuple2, а работать с каким-то другим типом, вы можете предоставить функцию zip(), которая создает любой объект, который вы хотите, учитывая два элемента (как показано на диаграмме marble на рисунке 10.8).



**Рисунок 10.8 альтернативная форма операции zip приводит к Flux сообщений, созданных из одного элемента каждого входящего Flux.**

Например, в следующем методе тестирования показано, как связать Flux имен с Flux продуктов питания, чтобы получить в результате Flux из String объектов:

@Test

public void zipFluxesToObject() {

   Flux<String> characterFlux = Flux

      .just("Garfield", "Kojak", "Barbossa");

   Flux<String> foodFlux = Flux

      .just("Lasagna", "Lollipops", "Apples");

   Flux<String> zippedFlux =

      Flux.zip(characterFlux, foodFlux, (c, f) -> c + " eats " + f);

   StepVerifier.create(zippedFlux)

      .expectNext("Garfield eats Lasagna")

      .expectNext("Kojak eats Lollipops")

      .expectNext("Barbossa eats Apples")

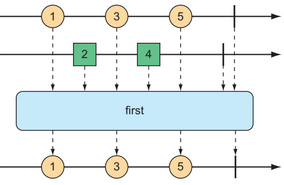
      .verifyComplete();

}

Функция, заданная для zip() (заданная здесь как лямбда), просто объединяет два элемента в предложение, которое отдается зипованным Flux.

***ВЫБОР ПЕРВОГО РЕАКТИВНОГО ТИПА ДЛЯ ПУБЛИКАЦИИ***

Предположим, у вас есть два Flux объекта, и вместо того, чтобы объединить их, вы просто хотите создать новый Flux, который будет генерировать значения из первого Flux, который создает значение. Как показано на рисунке 10.9, операция first() выбирает первый из двух объектов Flux и отображает значения, которые она публикует.



**Рис. 10.9. first операция выбирает первый Flux, который отправляет сообщение, и после этого создает только сообщения из этого потока.**

Следующий метод тестирования создает быстрый Flux и медленный Flux (где “медленный " означает, что он не будет публиковать элемент до 100 мс после подписки). Используя first(), он создает новый Flux, который будет публиковать значения только из первого исходного Flux для публикации значения:

@Test

public void firstFlux() {

   Flux<String> slowFlux = Flux.just("tortoise", "snail", "sloth")

      .delaySubscription(Duration.ofMillis(100));

   Flux<String> fastFlux = Flux.just("hare", "cheetah", "squirrel");

   Flux<String> firstFlux = Flux.first(slowFlux, fastFlux);

   StepVerifier.create(firstFlux)

      .expectNext("hare")

      .expectNext("cheetah")

      .expectNext("squirrel")

      .verifyComplete();

}

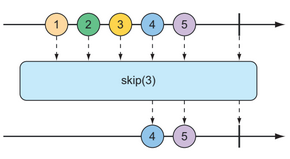
В этом случае, поскольку медленный Flux не будет публиковать никаких значений до 100 мс после начала публикации быстрого Flux, вновь созданный Flux будет просто игнорировать медленный Flux и публиковать только значения из быстрого Flux.

**10.3.3 Преобразование и фильтрация реактивных потоков**

Когда данные проходят через поток, вам, вероятно, потребуется отфильтровать некоторые значения и изменить другие значения. В этом разделе мы рассмотрим операции, которые преобразуют и фильтруют данные, проходящие через реактивный поток.

ФИЛЬТРАЦИЯ ДАННЫХ ИЗ РЕАКТИВНЫХ ТИПОВ

Один из самых основных способов фильтрации данных при их поступлении из Flux - просто игнорировать первые записи. Операция skip(), показанная на рисунке 10.10, делает именно это.



**Рисунок 10.10. Операция skip пропускает указанное количество сообщений перед передачей оставшихся сообщений в результирующий Flux.**

Учитывая Flux с несколькими записями, операция skip() создаст новый Flux, который пропускает заданное количество элементов, прежде чем отдавать остальные элементы из исходного Flux. Следующий метод тестирования показывает, как использовать skip():

@Test

public void skipAFew() {

   Flux<String> skipFlux = Flux.just(

      "one", "two", "skip a few", "ninety nine", "one hundred")

   .skip(3);

   StepVerifier.create(skipFlux)

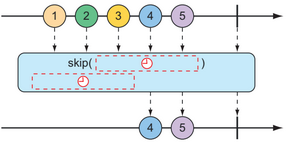
      .expectNext("ninety nine", "one hundred")

      .verifyComplete();

}

В этом случае у вас есть Flux из пяти String элементов. Вызов метода skip(3) для этого потока создает новый Flux, который пропускает первые три элемента и публикует только последние два элемента.

Но, возможно, вы не хотите пропускать определенное количество элементов, а вместо этого нужно пропустить некоторое количество элементов, определяемое не количеством, а временем. Альтернативная форма операции skip(), показанная на рисунке 10.11, создает Flux, который ожидает, пока не пройдет некоторое заданное время, прежде чем отдавать элементы из исходного Flux.



**Рисунок 10.11. Альтернативная форма операции skip ждет, пока не пройдет некоторое время, прежде чем передавать сообщения в результирующий поток.**

Следующий метод тестирования использует функцию skip() для создания Flux, который ожидает четыре секунды, прежде чем начинает отдавать какие-либо значения. Поскольку этот Flux был создан из Flux, который имеет односекундную задержку между элементами (используя delayElements()), будут переданы только последние два элемента:

@Test

public void skipAFewSeconds() {

   Flux<String> skipFlux = Flux.just(

         "one", "two", "skip a few", "ninety nine", "one hundred")

      .delayElements(Duration.ofSeconds(1))

      .skip(Duration.ofSeconds(4));

   StepVerifier.create(skipFlux)

      .expectNext("ninety nine", "one hundred")

      .verifyComplete();

}

Вы уже видели пример метода take(), но в познакомившись с методом skip(), take() можно рассматривать как противоположность skip (). В то время как функция skip() пропускает первые несколько элементов, функция take() выдает только первые несколько элементов (как показано на диаграмме marble на рис. 10.12):

@Test

public void take() {

   Flux<String> nationalParkFlux = Flux.just(

         "Yellowstone", "Yosemite", "Grand Canyon",

         "Zion", "Grand Teton")

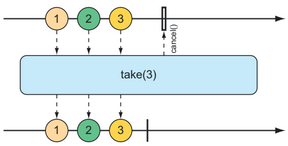
      .take(3);

   StepVerifier.create(nationalParkFlux)

      .expectNext("Yellowstone", "Yosemite", "Grand Canyon")

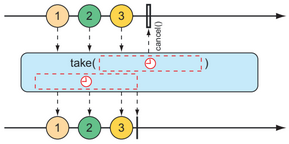
      .verifyComplete();

}



**Рис. 10.12 операция take передает только первые несколько сообщений из входящего Flux, а затем отменяет подписку.**

Как и skip(), take() также имеет альтернативную форму, основанную на длительности, а не на количестве элементов. Он будет принимать и отдавать столько элементов, сколько проходит через исходный поток, пока не пройдет некоторый период времени, после чего поток завершится. Это показано на рисунке 10.13.



**Рис. 10.13. Альтернативная форма операции take передает сообщения в результирующий поток до тех пор, пока не пройдет некоторое время.**

В следующем методе тестирования используется альтернативная форма take() для отправки максимально возможного количества элементов в первые 3,5 секунды после подписки:

@Test

public void take() {

   Flux<String> nationalParkFlux = Flux.just(

         "Yellowstone", "Yosemite", "Grand Canyon",

         "Zion", "Grand Teton")

      .delayElements(Duration.ofSeconds(1))

      .take(Duration.ofMillis(3500));

   StepVerifier.create(nationalParkFlux)

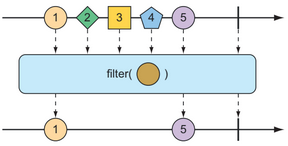
      .expectNext("Yellowstone", "Yosemite", "Grand Canyon")

      .verifyComplete();

}

Операции skip() и take() можно рассматривать как операции фильтрации, где критерии фильтра основаны на количестве или длительности. Для более общей фильтрации значений Flux вы найдете операцию filter() весьма полезной.

При наличии предиката, который решает, будет ли элемент проходить через поток или нет, операция filter() позволяет выборочно публиковать на основе любых критериев, которые вы хотите. Мраморная диаграмма на рисунке 10.14 показывает, как работает filter().



**Рисунок 10.14. Входящий Flux может быть отфильтрован так, что полученный Flux получает только сообщения, которые соответствуют заданному предикату.**

Чтобы увидеть filter() в действии, рассмотрите следующий метод тестирования:

@Test

public void filter() {

   Flux<String> nationalParkFlux = Flux.just(

         "Yellowstone", "Yosemite", "Grand Canyon",

         "Zion", "Grand Teton")

      .filter(np -> !np.contains(" "));

   StepVerifier.create(nationalParkFlux)

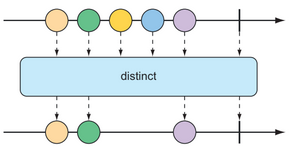
      .expectNext("Yellowstone", "Yosemite", "Zion")

      .verifyComplete();

}

Здесь filter() задается предикатом в виде лямбды, которая принимает только String значения без пробелов. Следовательно, «Grand Canyon» и «Grand Teton» отфильтровываются из итогового Flux.

Возможно, вам нужна фильтрация для всех предметов, которые вы уже получили. Операция Different(), как показано на рисунке 10.15, приводит к тому, что Flux публикует только элементы из исходного потока, которые еще не были опубликованы.



**Рисунок 10.15. Операция distinct отфильтровывает любые повторяющиеся сообщения.**

В следующем тесте только уникальные String значения будут излучаться из Flux:

@Test

public void distinct() {

   Flux<String> animalFlux = Flux.just(

         "dog", "cat", "bird", "dog", "bird", "anteater")

      .distinct();

   StepVerifier.create(animalFlux)

      .expectNext("dog", "cat", "bird", "anteater")

      .verifyComplete();

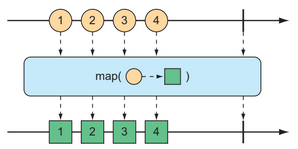
}

Хотя «dog» и «bird» публикуются дважды из исходного потока, отдельный поток публикует их только один раз.

***МАППИНГ РЕАКТИВНЫХ ДАННЫХ***

Одной из наиболее распространенных операций, которые вы будете использовать в Flux или Моно, является преобразование опубликованных элементов в какую-либо другую форму или тип. Типы Reactor предлагают операции map() и flatMap() для этой цели.

Операция map() создает Flux, который просто выполняет преобразование, как предписано данной функцией, для каждого объекта, который он получает до его повторной публикации. На рисунке 10.16 показано, как работает операция map().



**Рисунок 10.16. Операция map выполняет преобразование входящих сообщений в новые сообщения в результирующем потоке.**

В следующем методе тестирования Flux String значения, представляющие баскетболистов, сопоставляются с новы Flux объектами Player:

@Test

public void map() {

   Flux<Player> playerFlux = Flux

      .just("Michael Jordan", "Scottie Pippen", "Steve Kerr")

      .map(n -> {

         String[] split = n.split("\\s");

         return new Player(split[0], split[1]);

      });

   StepVerifier.create(playerFlux)

      .expectNext(new Player("Michael", "Jordan"))

      .expectNext(new Player("Scottie", "Pippen"))

      .expectNext(new Player("Steve", "Kerr"))

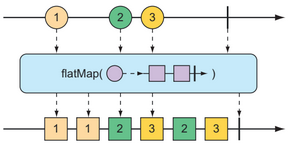
      .verifyComplete();

}

Функция, заданная для map() (как лямбда), разбивает входящую String по пробелу и использует полученный массив String-ов для создания объекта Player. Хотя поток, созданный с помощью just(), переносил объекты String, Flux, полученный из map(), переносит объекты Player.

Что важно понимать в map(), так это то, что сопоставление выполняется синхронно, так как каждый элемент публикуется исходным Flux. Если вы хотите выполнить сопоставление асинхронно, вы должны рассмотреть операцию flatMap().

Операция flatMap() требует некоторой мысли и практики, чтобы овладеть всеми навыками. Как показано на рисунке 10.17, вместо простого сопоставления одного объекта другому, как в случае map(), flatMap() сопоставляет каждый объект новому Mono или Flux. Результаты Mono или Flux сведены в новый результирующий Flux. Когда используется вместе с subscribeOn(), flatMap() может раскрыть асинхронную мощь типов Reactor.



**Рис. 10.17. Операция плоской карты (flat map) использует промежуточный Flux для выполнения преобразования, следовательно, допускает асинхронные преобразования.**

Следующий метод тестирования демонстрирует использование flatMap() и subscribeOn():

@Test

public void flatMap() {

   Flux<Player> playerFlux = Flux

      .just("Michael Jordan", "Scottie Pippen", "Steve Kerr")

      .flatMap(n -> Mono.just(n)

      .map(p -> {

         String[] split = p.split("\\s");

         return new Player(split[0], split[1]);

      })

      .subscribeOn(Schedulers.parallel())

   );

   List<Player> playerList = Arrays.asList(

      new Player("Michael", "Jordan"),

      new Player("Scottie", "Pippen"),

      new Player("Steve", "Kerr"));

   StepVerifier.create(playerFlux)

      .expectNextMatches(p -> playerList.contains(p))

      .expectNextMatches(p -> playerList.contains(p))

      .expectNextMatches(p -> playerList.contains(p))

      .verifyComplete();

}

Обратите внимание, что flatMap() получает лямбда-функцию, которая преобразует входящую String в Mono типа String. Затем к Mono применяется операция map() для преобразования String в Player.

Если вы остановитесь прямо здесь, результирующий поток будет передавать объекты Player, созданные синхронно в том же порядке, что и в примере с map(). Но операции с Mono завершаются вызовом subscribeOn(), чтобы указать, что каждая подписка должна проходить в параллельном потоке. Следовательно, операции сопоставления для нескольких входящих объектов String могут выполняться асинхронно и параллельно.

Хотя subscribeOn() называется очень похоже на subscribe(), но они совершенно разные. В то время как subscribe() - это глагол, подписывающийся на реактивный поток и эффективно запускающий его, subscribeOn() - является более описательной, определяя, как подписка должна обрабатываться параллельно. Reactor не навязывает какую-либо конкретную модель параллелизма; с помощью subscribeOn() вы можете указать модель параллелизма, используя один из статических методов из планировщиков, который вы хотите использовать. В этом примере вы использовали parallel(), которая использует рабочие потоки из фиксированного пула (размер которого соответствует числу ядер ЦП). Но планировщики поддерживают несколько моделей параллелизма, например, описанных в таблице 10.1.

**Таблица 10.1 Модели параллелизма для планировщиков (Schedulers)**

Метод планировщика: Описание

**.immediate()** - *Выполняет подписку в текущем потоке.*

**.single()** - *Выполняет подписку в одном многоразовом потоке. Повторно использует один и тот же поток для всех абонентов.*

**.newSingle()** - *Выполняет подписку в выделенном потоке для каждого вызова.*

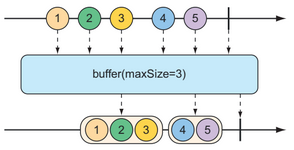
**.elastic()** - *Выполняет подписку в в рабочем пуле из неограниченного эластичного пула. Новые рабочие потоки создаются по мере необходимости, а простаивающие рабочие потоки удаляются (по умолчанию через 60 секунд).*

**.parallel()** - *Выполняет подписку в рабочем пуле из из пула фиксированного размера, размер которого соответствует числу ядер ЦП.*

Преимуществом использования flatMap() и subscribeOn() является то, что вы можете увеличить пропускную способность потока, разделив работу между несколькими параллельными потоками. Но поскольку работа выполняется параллельно, без гарантий того, что будет постоянный порядок выполнения потоков, невозможно определить порядок элементов, передаваемых в результирующий поток. Таким образом, StepVerifier может только проверить, что каждый исходящий элемент существует в ожидаемом списке объектов Player и что до завершения потока будет три таких элемента.

***БУФЕРИЗАЦИЯ ДАННЫХ В РЕАКТИВНОМ ПОТОКЕ***

В процессе обработки данных, проходящих через Flux, может оказаться полезным разбить поток данных на небольшие части. Операция buffer(), показанная на рисунке 10.18, может помочь в этом.



**Рисунок 10.18 Операция buffer приводит к листу Flux заданного максимального размера, которые формируется на основе входящего Flux.**

Учитывая, что Flux у нас String значений, каждое из которых содержит имя фрукта, вы можете создать новый Flux коллекции List, в которой каждый List содержит не более указанного числа элементов:

@Test

public void buffer() {

   Flux<String> fruitFlux = Flux.just(

      "apple", "orange", "banana", "kiwi", "strawberry");

   Flux<List<String>> bufferedFlux = fruitFlux.buffer(3);

   StepVerifier

      .create(bufferedFlux)

      .expectNext(Arrays.asList("apple", "orange", "banana"))

      .expectNext(Arrays.asList("kiwi", "strawberry"))

      .verifyComplete();

}

В этом случае String элементы Flux помещаются в новые Flux коллекции List, содержащие не более трех элементов в каждой. Следовательно, исходный Flux, который передает пять значений String, будет преобразован в Flux, который передает две коллекции List, одна из которых содержит три фрукта, а другая - два фрукта.

И что? Буферизация значений из реактивного потока в нереактивные коллекции List представляется контрпродуктивной. Но когда вы комбинируете buffer() с flatMap(), это позволяет параллельно обрабатывать каждую из коллекций List:

Flux.just(

   "apple", "orange", "banana", "kiwi", "strawberry")

   .buffer(3)

   .flatMap(x ->

      Flux.fromIterable(x)

      .map(y -> y.toUpperCase())

      .subscribeOn(Schedulers.parallel())

      .log()

   ).subscribe();

В этом новом примере вы по-прежнему буферизуете Flux из пяти String значений в Flux коллекций List. Но затем вы применяете flatMap() к этому Flux коллекций List. Это берет каждый List буфер и создает новый Flux из его элементов, а затем применяет к нему операцию map(). Следовательно, каждый буферный List дополнительно обрабатывается параллельно в отдельных потоках.

Чтобы доказать, что это работает, я также включил операцию log() для применения к каждому под-Flux-у. Операция log() просто регистрирует все события Reactive Streams, чтобы вы могли видеть, что на самом деле происходит. В результате в log записываются следующие записи (для краткости удален компонент времени):

[main] INFO reactor.Flux.SubscribeOn.1 -

   onSubscribe(FluxSubscribeOn.SubscribeOnSubscriber)

[main] INFO reactor.Flux.SubscribeOn.1 - request(32)

[main] INFO reactor.Flux.SubscribeOn.2 -

   onSubscribe(FluxSubscribeOn.SubscribeOnSubscriber)

[main] INFO reactor.Flux.SubscribeOn.2 - request(32)

[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(APPLE)

[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onNext(KIWI)

[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(ORANGE)

[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onNext(STRAWBERRY)

[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(BANANA)

[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onComplete()

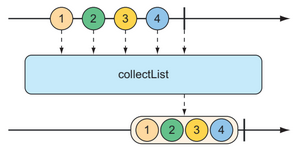
[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onComplete()

Как ясно видно из записей журнала, фрукты в первом буфере (apple, orange и banana) обрабатываются в потоке parallel-1. Между тем, фрукты во втором буфере (kiwi и strawberry) обрабатываются в parallel-2. Как видно из того факта, что записи журнала из каждого буфера сплетены вместе, два буфера обрабатываются параллельно.

Если по какой-то причине вам нужно собрать все, что Flux генерирует в List, вы можете вызвать buffer() без аргументов:

Flux<List<String>> bufferedFlux = fruitFlux.buffer();

В результате создается новый Flux, который генерирует List, содержащий все элементы, опубликованные исходным Flux. Вы можете добиться того же самого с помощью операции collectList(), показанной на диаграмме на рисунке 10.19.



**Рисунок 10.19 В результате операции сбора списка получается Mono, содержащий список всех сообщений, отправляемых входящим Flux.**

Вместо того чтобы создавать Flux, который публикует List, collectList() создает Mono, который публикует List. Следующий метод тестирования показывает, как это можно использовать:

@Test

public void collectList() {

   Flux<String> fruitFlux = Flux.just(

      "apple", "orange", "banana", "kiwi", "strawberry");

   Mono<List<String>> fruitListMono = fruitFlux.collectList();

   StepVerifier

      .create(fruitListMono)

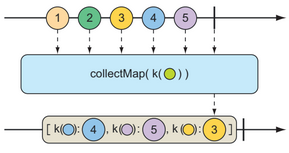
      .expectNext(Arrays.asList(

      "apple", "orange", "banana", "kiwi", "strawberry"))

      .verifyComplete();

}

Еще более интересный способ сбора элементов, возвращаемых Flux, - это собирать их в Map. Как показано на рисунке 10.20, операция collectMap() приводит к Mono, который публикует Map, заполненную записями, ключ которых рассчитывается данной функцией.



**Рис. 10.20. Операция collectMap приводит к получению Mono, содержащему Map сообщений, передаваемым входящим Flux, где ключ выводится из некоторой характеристики входящих сообщений.**

Чтобы увидеть collectMap() в действии, взгляните на следующий метод тестирования:

@Test

public void collectMap() {

   Flux<String> animalFlux = Flux.just(

      "aardvark", "elephant", "koala", "eagle", "kangaroo");

   Mono<Map<Character, String>> animalMapMono =

      animalFlux.collectMap(a -> a.charAt(0));

       StepVerifier

        .create(animalMapMono)

        .expectNextMatches(map -> {

         return

            map.size() == 3 &&

            map.get('a').equals("aardvark") &&

            map.get('e').equals("eagle") &&

            map.get('k').equals("kangaroo");

      })

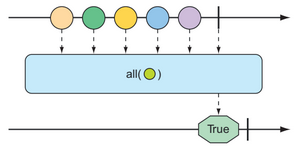
   .verifyComplete();

}

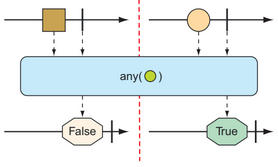
Источник Flux испускает имена нескольких животных. К этому Flux вы применяете collectMap() для создания нового Mono, который создает Map, где значение ключа определяется первой буквой имени животного, а значение-само имя животного. В случае, если два названия животных начинаются с одной и той же буквы (например, elephant и eagle или koala и kangaroo), последняя запись, проходящая через поток, переопределяет все предыдущие записи.

**10.3.4 Выполнение логических операций над реактивными типами**

Иногда вам просто нужно знать, соответствуют ли записи, опубликованные Mono или Flux, некоторым критериям. Операции all() и any() выполняют такую логику. Рисунки 10.21 и 10.22 иллюстрируют, как работают all() и any().



**Рисунок 10.21 Поток может быть проверен, чтобы убедиться, что все сообщения удовлетворяют некоторому условию в операции all.**



**Рисунок 10.22 поток может быть проверен, что по крайней мере одно сообщение удовлетворяет некоторому условию any операции.**

Предположим, вы хотите знать, что каждая строка, публикуемая Flux, содержит букву a или букву k. Следующий тест показывает, как использовать all() для проверки этого условия:

@Test

public void all() {

   Flux<String> animalFlux = Flux.just(

      "aardvark", "elephant", "koala", "eagle", "kangaroo");

   Mono<Boolean> hasAMono = animalFlux.all(a -> a.contains("a"));

   StepVerifier.create(hasAMono)

      .expectNext(true)

      .verifyComplete();

   Mono<Boolean> hasKMono = animalFlux.all(a -> a.contains("k"));

   StepVerifier.create(hasKMono)

      .expectNext(false)

      .verifyComplete();

}

В первом StepVerifier, проверяется наличие буквы a. Операция all применяется к исходному Flux, в результате чего получается Mono типа Boolean. В этом случае все названия животных содержат букву а, поэтому Mono будет содержать true. Но на втором этапе проверки результирующий Mono будет выдавать false, потому что не все имена животных содержат k.

Вместо того, чтобы выполнять проверку "все или ничего", Возможно, будет достаточно, если хотя бы одна запись соответствует условиям. В этом случае операция any() - это то, что вы хотите. Этот новый тестовый случай использует any() для проверки букв t и z:

@Test

public void any() {

   Flux<String> animalFlux = Flux.just(

      "aardvark", "elephant", "koala", "eagle", "kangaroo");

   Mono<Boolean> hasAMono = animalFlux.any(a -> a.contains("t"));

   StepVerifier.create(hasAMono)

      .expectNext(true)

      .verifyComplete();

   Mono<Boolean> hasZMono = animalFlux.any(a -> a.contains("z"));

   StepVerifier.create(hasZMono)

      .expectNext(false)

      .verifyComplete();

}

В первом StepVerifier вы видите, что полученный Mono возвращает true, потому что по крайней мере одно имя животного имеет букву t (в частности, elephant). Во втором случае полученное Mono возвращает false, потому что ни одно из имен животных не содержит z.

ИТОГО:

-Реактивное программирование включает в себя создание конвейеров, по которым передаются данные.

-Спецификация Reactive Streams определяет четыре типа: «Издатель» (Publisher), «Подписчик» (Subscriber), «Подписка» (Subscription) и «Трансформер» (Transformer) (который является комбинацией Publisher и Subscriber).

-Проект Reactor реализует Reactive Streams и абстрагирует определения потоков в два основных типа, Flux и Mono, каждый из которых предлагает несколько сотен операций.

-Spring 5 использует Reactor для создания реактивных контроллеров, репозиториев, REST клиентов и другой поддержки реактивной платформы.

[Spring in Action Covers Spring 5.0 перевод на русский. Глава 11](https://beginner-mercenary.blogspot.com/2019/11/spring-in-action-covers-spring-50-11.html)

**Глава 11. Разработка реактивных API**

**Эта глава охватывает:**

*-Использование Spring WebFlux*

*-Написание и тестирование реактивных контроллеров и клиентов*

*-Использование REST API*

*-Защита реактивных веб-приложений*

Теперь, когда вы хорошо познакомились с реактивным программированием и Project Reactor, вы готовы начать применять эти методы в своих Spring приложениях. В этой главе мы собираемся вернуться к некоторым контроллерам, которые вы написали в главе 6, чтобы воспользоваться преимуществами модели реактивного программирования Spring 5.

Более конкретно, мы собираемся взглянуть на новую реактивную веб-инфраструктуру Spring 5 - Spring WebFlux. Как вы вскоре поймете, Spring WebFlux удивительно похож на Spring MVC, что делает его простым в применении, наряду с тем, что вы уже знаете о создании REST API в Spring.

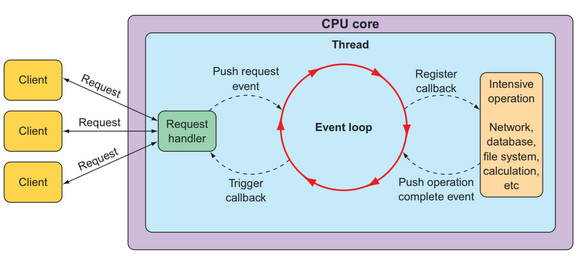
**11.1 Работа с Spring WebFlux**

Типичные веб-фреймворки на основе сервлетов, такие как Spring MVC, являются блокирующими и многопоточными по своей природе, используя один поток на соединение. Когда запросы обрабатываются, рабочий поток извлекается из пула потоков для обработки запроса. Тем временем поток запросов блокируется, пока рабочий поток не уведомит его о завершении.

Следовательно, блокирование веб-фреймворков не будет эффективно масштабироваться при большом объеме запросов. Задержка в медленных рабочих потоках усугубляет ситуацию, поскольку для возврата рабочего потока в пул, готового обработать другой запрос, потребуется больше времени. В некоторых случаях такое расположение вполне приемлемо. Фактически, это во многом такой подход, как большинство веб-приложений разрабатывалось более десяти лет. Но времена меняются.

Клиенты этих веб-приложений изменились и не смену людям, иногда просматривающих веб-сайты, пришли люди, часто потребляющих контент и использующих приложения, которые координируются с HTTP API. И в наши дни так называемый Интернет вещей (где люди даже не участвуют) предоставляет автомобили, реактивные двигатели и другие нетрадиционные клиенты, постоянно обменивающиеся данными с веб-API. С увеличением числа клиентов, использующие веб-приложения, масштабируемость становится более важной, чем когда-либо.

Асинхронные веб-инфраструктуры, напротив, обеспечивают более высокую масштабируемость при меньшем количестве потоков - обычно по одному на ядро ЦП. Применяя метод, известный как цикл обработки событий (как показано на рисунке 11.1), эти платформы способны обрабатывать много запросов на поток, что делает затраты на соединение более экономичными.



**Рисунок 11.1. Асинхронные веб-фреймворки применяют зацикливание событий для обработки большего количества запросов с меньшим количеством потоков.**

В цикле событий все обрабатывается как событие, включая запросы и обратные вызовы от интенсивных операций, таких как операции с базой данных и сетью. Когда требуется дорогостоящая операция, цикл событий регистрирует обратный вызов для выполнения этой операции параллельно, а затем переходит к обработке других событий.

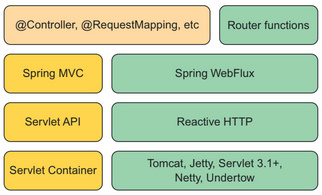
Когда операция завершена, она обрабатывается как событие в цикле событий, так же как и запросы. В результате асинхронные веб-инфраструктуры могут лучше масштабироваться при больших объемах запросов с меньшим количеством потоков, что приводит к снижению накладных расходов на управление потоками.

Spring 5 представил неблокирующую асинхронный веб-фреймворк, основанный в основном на своем Project Reactor, для удовлетворения потребностей в большей масштабируемости в веб-приложениях и API-интерфейсах. Давайте взглянем на Spring WebFlux - реактивный веб-фреймворк для Spring.

**11.1.1 Описание Spring WebFlux**

Когда команда Spring обдумывала, как добавить модель реактивного программирования в веб-слой, быстро стало очевидно, что это будет трудно сделать без большой работы в Spring MVC. Это будет включать в себя код ветвления, чтобы решить, реагировать ли на запросы реактивно или нет. По сути, результатом будут два веб-фреймворка, упакованные как один, с инструкциями if для разделения реактивных и нереактивных действий.

Вместо того чтобы пытаться встроить реактивную модель программирования в Spring MVC, было решено создать отдельный реактивный веб-фреймворк, заимствуя как можно больше от Spring MVC. Итогом стал Spring WebFlux. Рисунок 11.2 иллюстрирует полный стек веб-разработки, определенный Spring 5.



**Рис. 11.2 Spring 5 поддерживает реактивные веб-приложения с новым веб-фрэймворком Web Flux, который является родственным Spring MVC и разделяет многие из ее основных компонентов.**

В левой стороне рисунка 11.2 вы видите стек Spring MVC, который был представлен в версии 2.5 Spring Framework. Spring MVC (описанный в главах 2 и 6) расположен поверх API сервлетов Java, для которого требуется контейнер сервлета (например, Tomcat).

Spring WebFlux (с правой стороны) не имеет связей с API сервлета, поэтому он строится поверх реактивного HTTP API, который является реактивным приближением той же функциональности, предоставляемой API сервлета. И поскольку Spring WebFlux не связан с API сервлета, для его запуска не требуется контейнер сервлета. Вместо этого, он может работать на любом неблокирующем веб-контейнере, включая Netty, Undertow, Tomcat, Jetty, или любой контейнер Servlet 3.1 или выше.

На рис. 11.2 наиболее примечательным является верхний левый прямоугольник, представляющий компоненты, общие для Spring MVC и Spring WebFlux, в первую очередь аннотации, используемые для определения контроллеров. Поскольку Spring MVC и Spring WebFlux используют одни и те же аннотации, Spring WebFlux во многом неотличим от Spring MVC.

Прямоугольник в верхнем правом углу представляет альтернативную модель программирования, которая определяет контроллеры с функциональной парадигмой программирования вместо использования аннотаций. Более подробно о функциональной модели веб-программирования Spring мы расскажем в разделе 11.2.

Наиболее существенное различие между Spring MVC и Spring WebFlux сводится к тому, какую зависимость вы добавляете в свою сборку. При работе с Spring WebFlux вам необходимо добавить Spring Boot WebFlux стартер вместо стандартного веб стартера (например, spring-boot-starter-web). В файле проекта pom.xml это выглядит так:

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

ПРИМЕЧАНИЕ. Как и с большинством Spring Boot стартеров, этот стартер также можно добавить в проект, установив флажок Reactive Web в Initializr.

Интересным побочным эффектом использования WebFlux вместо Spring MVC является то, что встроенным сервером по умолчанию для WebFlux является Netty вместо Tomcat. Netty - один из нескольких асинхронных серверов, управляемых событиями, и он естественным образом подходит для реактивной веб-инфраструктуры, такой как Spring WebFlux.

Помимо использования другой стартер зависимости, методы контроллера Spring WebFlux обычно принимают и возвращают реактивные типы, такие как Mono и Flux, вместо типов доменов и коллекций. Контроллеры Spring WebFlux также могут работать с типами RxJava, такими как Observable, Single и Completable.

**РЕАКТИВНЫЙ SPRING MVC?**

Хотя Spring WebFlux контроллеры обычно возвращают Mono и Flux, это не означает, что Spring MVC не может работать с реактивными типами. Методы контроллера Spring MVC также могут возвращать Mono или Flux, если хотите.

Разница в том, как эти типы используются. В то время как Spring WebFlux представляет собой действительно реактивный веб-фрэйворк, позволяющую обрабатывать запросы в цикле обработки событий, Spring MVC основан на сервлетах, полагаясь на многопоточность для обработки нескольких запросов.

Давайте включим Spring WebFlux в работу, переписав некоторые из контроллеров API Taco Cloud, чтобы использовать преимущества Spring WebFlux.

**11.1.2 Написание реактивных контроллеров**

Возможно, вы помните, что в главе 6 вы создали несколько контроллеров для REST API Taco Cloud. Эти контроллеры имели методы обработки запросов, которые имели дело с вводом и выводом с точки зрения типов доменов (таких как Order и Taco) или наборов этих типов доменов. В качестве напоминания рассмотрим следующий фрагмент из DesignTacoController, который вы написали в главе 6:

@RestController

@RequestMapping(path="/design", produces="application/json")

@CrossOrigin(origins="\*")

public class DesignTacoController {

   ...

   @GetMapping("/recent")

   public Iterable<Taco> recentTacos() {

      PageRequest page = PageRequest.of(

         0, 12, Sort.by("createdAt").descending());

      return tacoRepo.findAll(page).getContent();

   }

...

}

Как уже было ранее написано, контроллер latestTacos() обрабатывает HTTP-запросы GET для /design/recent, чтобы вернуть список недавно созданных тако. Более конкретно, он возвращает Iterable с типом Taco. Это в первую очередь потому, что это то, что возвращается из метода findAll() репозитьтлоия, или, точнее, из метода getContent() объекта Page, возвращаемого из findAll().

Это прекрасно работает, но Iterable не является реактивным типом. Вы не сможете применить к нему какие-либо реактивные операции, и при этом вы не сможете позволить фрэймворку использовать его как реактивный тип для разделения любой работы на несколько потоков. То, что вы хотели бы, чтобы recentTacos() возвращали Flux <Taco>.

Простой, но несколько ограниченный вариант здесь - переписать recentTacos() для преобразования Iterable во Flux. А заодно избавимся от кода паджинации и заменим его вызовом метода take() у Flux:

@GetMapping("/recent")

public Flux<Taco> recentTacos() {

   return Flux.fromIterable(tacoRepo.findAll()).take(12);

}

Используя Flux.fromIterable(), вы конвертируете Iterable<Taco> в Flux<Taco>. И теперь, когда вы работаете с Flux, вы можете использовать операцию take(), чтобы ограничить возвращаемый Flux максимум 12 объектами Taco. Код не только проще, он также имеет дело с реактивным Flux, а не простым Iterable.

До сих пор написание реактивного кода было выигрышным шагом. Но было бы еще лучше, если бы репозиторий давало вам Flux для начала, чтобы вам не нужно было выполнять преобразование. Если бы это было так, то recentTacos() можно было бы написать так:

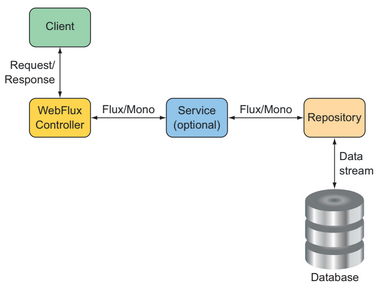
@GetMapping("/recent")

public Flux<Taco> recentTacos() {

   return tacoRepo.findAll().take(12);

}

Это даже лучше! В идеале реактивный контроллер будет вершиной стека, который является реактивным от начала до конца, включая контроллеры, репозитории, базу данных и любые службы, которые могут находиться между ними. Такой сквозной реактивный стек показан на рис. 11.3.



**Рис. 11.3. Чтобы максимизировать преимущества реактивного веб-фрэймворка, она должна быть частью полного сквозного реактивного стека.**

Такой end-to-end стек требует, чтобы репозиторий был написан так, чтобы он возвращал Flux вместо Iterable. Мы рассмотрим написание реактивных репозиториев в следующей главе, но вот краткий обзор того, как может выглядеть реактивный TacoRepository:

public interface TacoRepository

   extends ReactiveCrudRepository<Taco, Long> {

}

Однако наиболее важно отметить, что помимо работы с Flux вместо Iterable,, а также того, как вы получаете этот Flux, программная модель для определения реактивного контроллера WebFlux ничем не отличается от нереактивного контроллера Spring MVC. Оба аннотируются с помощью @RestController и высокоуровневого @RequestMapping на уровне класса. И оба имеют функции обработки запросов, которые аннотируются с помощью @GetMapping на уровне метода. Разница только в том, какой тип возвращают методы обработчика.

Еще одно важное замечание заключается в том, что, хотя вы получаете Flux<Taco> из репозитория, вы можете вернуть его без вызова subscribe(). Действительно, фреймворк вызовет для вас метод subscribe(). Это означает, что при обработке запроса /design/recent будет вызван метод recentTacos(), который вернется до того, как данные будут даже получены из базы данных!

**ВОЗВРАЩЕНИЕ ОДИНОЧНЫХ ЗНАЧЕНИЙ**

В качестве другого примера рассмотрим метод tacoById() из DesignTacoController, как он был написан в главе 6:

@GetMapping("/{id}")

public Taco tacoById(@PathVariable("id") Long id) {

   Optional<Taco> optTaco = tacoRepo.findById(id);

   if (optTaco.isPresent()) {

      return optTaco.get();

   }

   return null;

}

Здесь этот метод обрабатывает запросы GET для /design/{id} и возвращает одиночный объект Taco. Поскольку findById() репозитория возвращает Optional, вам пришлось написать какой-то неуклюжий код, чтобы справиться с этим. Но предположим на минуту, что findById() возвращает Mono<Taco> вместо Optional<Taco>. В этом случае вы можете переписать tacoById(), чтобы выглядело следующим образом:

@GetMapping("/{id}")

public Mono<Taco> tacoById(@PathVariable("id") Long id) {

   return tacoRepo.findById(id);

}

Вау! Это намного проще. Однако более важно то, что, возвращая Mono<Taco> вместо Taco, вы позволяете Spring WebFlux обрабатывать ответ реагирующим образом. Следовательно, ваш API будет лучше масштабироваться в ответ на большие нагрузки.

**РАБОТА С ТИПАМИ RXJAVA**

Стоит отметить, что хотя типы Reactor, такие как Flux и Mono, являются естественным выбором при работе с Spring WebFlux, вы также можете выбрать работу с типами RxJava, такими как Observable и Single. Например, предположим, что между DesignTacoController и внутренним репозиторием находится служба, которая работает в терминах типов RxJava. В этом случае метод recentTacos() может быть написан так:

@GetMapping("/recent")

public Observable<Taco> recentTacos() {

   return tacoService.getRecentTacos();

}

Аналогично, метод tacoById() может быть написан для работы с RxJava Single, а не Mono:

@GetMapping("/{id}")

public Single<Taco> tacoById(@PathVariable("id") Long id) {

   return tacoService.lookupTaco(id);

}

Кроме того, методы контроллера Spring WebFlux также могут возвращать RxJava Completable, который эквивалентен Mono<Void> в Reactor. WebFlux также может возвращать Flowable в качестве альтернативы Observable или Reactor Flux.

**РЕАКТИВНАЯ ОБРАБОТКА ВХОДНЫХ ДАННЫХ**

До сих пор мы интересовались только тем, какие реактивные типы возвращают методы контроллера. Но с Spring WebFlux вы также можете принять Mono или Flux в качестве входных данных для метода-обработчика. Для демонстрации рассмотрим оригинальную реализацию postTaco() из DesignTacoController:

@PostMapping(consumes="application/json")

@ResponseStatus(HttpStatus.CREATED)

public Taco postTaco(@RequestBody Taco taco) {

   return tacoRepo.save(taco);

}

postTaco() не только возвращает простой объект Taco, но также принимает объект Taco, связанный с содержимым в теле запроса. Это означает, что postTaco() не может быть вызван до тех пор, пока полезная нагрузка запроса не будет полностью подготовлена для создания экземпляра объекта Taco. Это означает, что postTaco() не может быть возвращен, пока не завершится блокирующий вызов метода save() репозитория. Другими словами, запрос блокируется дважды: при поступлении в postTaco() и снова внутри postTaco(). Но, применив небольшое реактивное кодирование к postTaco(), вы можете сделать его полностью неблокирующим методом обработки запросов:

@PostMapping(consumes="application/json")

@ResponseStatus(HttpStatus.CREATED)

public Mono<Taco> postTaco(@RequestBody Mono<Taco> tacoMono) {

   return tacoRepo.saveAll(tacoMono).next();

}

Здесь postTaco() принимает Mono<Taco> и вызывает метод saveAll() хранилища, который, как вы увидите в следующей главе, принимает любую реализацию Reactive Streams Publisher, включая Mono или Flux. Метод saveAll() возвращает Flux<Taco>, но поскольку вы работаете с Mono, вы знаете, что существует не более одного Taco, который будет опубликован Flux. Поэтому вы можете вызвать next(), чтобы получить Mono <Taco>, который вернется из postTaco().

Принимая Mono<Taco> в качестве входных данных, метод вызывается немедленно, не дожидаясь готовности Taco на основе тела запроса. И поскольку репозиторий также является реактивным, он примет Mono и немедленно вернет Flux<Taco>, из которого вы вызываете next(), и вернете полученный Mono<Taco> ... и все это еще до того, как запрос будет обработан!

Spring WebFlux является фантастической альтернативой Spring MVC, предлагая возможность написания реактивных веб-приложений с использованием той же модели разработки, что и Spring MVC. Но у Spring 5 есть еще одна новая хитрость. Давайте посмотрим, как создавать реагирующие API, используя новый функциональный стиль программирования Spring 5.

**11.2 Определение функциональных обработчиков запросов**

Модель программирования Spring MVC, основанная на аннотациях, существует начиная с Spring 2.5 и пользуется большой популярностью. Это имеет несколько недостатков.

Во-первых, любое программирование на основе аннотаций включает разделение в определении того, ***что***аннотация должна делать и ***как***она должна это делать. Сами аннотации определяют, ***что***; а ***как***это определено в другом месте в рамках кода. Это усложняет модель программирования, когда речь заходит о какой-либо настройке или расширении, поскольку такие изменения требуют работы в коде, внешнем по отношению к аннотации. Более того, отладка такого кода сложна, потому что вы не можете установить точку останова для аннотации.

Кроме того, поскольку популярность Spring продолжает расти, разработчики, впервые познакомившиеся с Spring в других языках и фреймворках, могут обнаружить что Spring MVC (и WebFlux) на основе аннотаций совсем не такой, как они уже его знают. В качестве альтернативы WebFlux Spring 5 представил новую модель функционального программирования для определения реактивных API.

Эта новая модель программирования используется больше как библиотека и меньше как фреймворк, позволяя сопоставлять запросы с кодом обработчика без аннотаций. Написание API с использованием модели функционального программирования Spring включает четыре основных типа:

-RequestPredicate — объявляет виды запросов, которые будут обработаны.

-RouterFunction - бъявляет, как соответствующий запрос должен быть направлен в код обработчика.

-ServerRequest - представляет собой HTTP-запрос, включая доступ к информации в header и body.

-ServerResponse - представляет ответ HTTP, включая информацию header и body

В качестве простого примера, который объединяет все эти типы, рассмотрим следующий пример Hello World:

package demo;

import static org.springframework.web.

        reactive.function.server.RequestPredicates.GET;

import static org.springframework.web.

        reactive.function.server.RouterFunctions.route;

import static org.springframework.web.

        reactive.function.server.ServerResponse.ok;

import static reactor.core.publisher.Mono.just;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.reactive.function.server.RouterFunction;

@Configuration

public class RouterFunctionConfig {

    @Bean

    public RouterFunction<?> helloRouterFunction() {

        return route(GET("/hello"),

            request -> ok().body(just("Hello World!"), String.class));

    }

}

Первое, на что нужно обратить внимание, это то, что вы произвели статический импорт нескольких вспомогательных классов, которые вы можете использовать для создания вышеупомянутых функциональных типов. Вы также статически импортировали Mono, чтобы остальную часть кода было легче читать и понимать.

В этом классе аннотированном как @Configuration у вас есть один метод @Bean типа RouterFunction <?>. Как уже упоминалось, RouterFunction объявляет сопоставления между одним или несколькими объектами RequestPredicate и функциями, которые будут обрабатывать соответствующие запрос(ы).

Метод route() из RouterFunctions принимает два параметра: RequestPredicate и функцию для обработки совпадающих запросов. В этом случае метод GET() из RequestPredicates объявляет RequestPredicate, который совпадает с HTTP-запросами GET для пути /hello.

Что касается функции-обработчика, она написана как лямбда, хотя она также может быть ссылкой на метод. Хотя это явно не объявлено, лямбда-обработчик принимает ServerRequest в качестве параметра. Он возвращает ServerResponse, используя ok() из ServerResponse и body() из BodyBuilder, который был возвращен из ok(). Это было сделано для того, чтобы создать ответ с кодом состояния HTTP 200 (ОК) и полезной нагрузкой body с надписью Hello World!

Метод helloRouterFunction() объявляет RouterFunction, которая обрабатывает только один вид запроса. Но если вам нужно обработать запрос другого типа, вам не нужно писать другой метод @Bean, хотя вы можете это сделать. Вам нужно только вызвать andRoute(), чтобы объявить другое сопоставление RequestPredicate-to-function. Например, вот как вы можете добавить другой обработчик для запросов GET для /bye:

@Bean

public RouterFunction<?> helloRouterFunction() {

    return route(GET("/hello"),

            request -> ok().body(just("Hello World!"), String.class))

        .andRoute(GET("/bye"),

            request -> ok().body(just("See ya!"), String.class));

}

Hello World примеры отлично подходят для знакомства с чем-то новым. Но давайте немного расширим этот пример и посмотрим, как использовать функциональную модель веб-программирования Spring для обработки запросов, которые напоминают реальные сценарии.

Чтобы продемонстрировать, как функциональная модель программирования может использоваться в реальном приложении, давайте заново напишем функционал DesignTacoController в функциональном стиле. Следующий класс конфигурации является функциональным аналогом DesignTacoController:

@Configuration

public class RouterFunctionConfig {

    @Autowired

    private TacoRepository tacoRepo;

    @Bean

    public RouterFunction<?> routerFunction() {

        return route(GET("/design/taco"), this::recents)

            .andRoute(POST("/design"), this::postTaco);

    }

    public Mono<ServerResponse> recents(ServerRequest request) {

        return ServerResponse.ok()

            .body(tacoRepo.findAll().take(12), Taco.class);

    }

    public Mono<ServerResponse> postTaco(ServerRequest request) {

        Mono<Taco> taco = request.bodyToMono(Taco.class);

        Mono<Taco> savedTaco = tacoRepo.save(taco);

        return ServerResponse

            .created(URI.create(

                "http://localhost:8080/design/taco/" +

                savedTaco.getId()))

            .body(savedTaco, Taco.class);

    }

}

Как вы можете видеть, метод routerFunction() объявляет bean RouterFunction<?> как в примере Hello World. Но это зависит от того, какие типы запросов обрабатываются и как они обрабатываются. Но он отличается тем, какие типы запросов обрабатываются и как они обрабатываются. В этом случае функция маршрутизатора создается для обработки запросов GET для /design/taco и POST для /design.

Маршруты обрабатываются ссылками на методы. Лямбды хороши, когда поведение функции RouterFunction относительно простое и краткое. Во многих случаях, однако, лучше извлечь эту функциональность в отдельный метод (или даже в отдельный метод в отдельном классе), чтобы поддерживать читаемость кода.

Запросы GET на /design/taco будут обрабатываться методом recents(). Он использует внедренный TacoRepository для извлечения Mono<Taco>, из которого он берет 12 элементов. А запросы POST для /design обрабатываются методом postTaco(), который извлекает Mono<Taco> из входящего ServerRequest. Затем метод postTaco() использует TacoRepository, чтобы сохранить его, прежде чем вернуть Mono<Taco>, который возвращающийся из метода save().

**11.3 Тестирование реактивных контроллеров**

Когда дело дойдет до тестирования реактивных контроллеров, Spring 5 не оставит нас в беде. Действительно, Spring 5 представил WebTestClient, новую утилиту для тестирования, которая упрощает написание тестов для реактивных контроллеров, написанных с использованием Spring WebFlux. Чтобы увидеть, как писать тесты с помощью WebTestClient, давайте используем его для тестирования метода recentTacos() из DesignTacoController, который вы написали в разделе 11.1.2.

**11.3.1. Тестирование GET запросов**

Одна вещь, которую мы хотели бы заявить о методе recentTacos(), заключается в том, что если для пути /design/recent выдается запрос HTTP GET, то ответ будет содержать полезную нагрузку JSON с не более чем 12 тако. Тестовый класс в следующем листинге - хорошее начало.

**Листинг 11.1. Использование WebTestClient для тестирования DesignTacoController**

package tacos;

import static org.mockito.Mockito.\*;

import java.util.ArrayList;

import java.util.List;

import org.junit.Test;

import org.mockito.Mockito;

import org.springframework.http.MediaType;

import org.springframework.test.web.reactive.server.WebTestClient;

import reactor.core.publisher.Flux;

import tacos.Ingredient.Type;

import tacos.data.TacoRepository;

import tacos.web.api.DesignTacoController;

public class DesignTacoControllerTest {

   @Test

   public void shouldReturnRecentTacos() {

      Taco[] tacos = {            //Задание тестовых данных

         testTaco(1L), testTaco(2L),

         testTaco(3L), testTaco(4L),

         testTaco(5L), testTaco(6L),

         testTaco(7L), testTaco(8L),

         testTaco(9L), testTaco(10L),

         testTaco(11L), testTaco(12L),

         testTaco(13L), testTaco(14L),

         testTaco(15L), testTaco(16L)};

      Flux<Taco> tacoFlux = Flux.just(tacos);

      TacoRepository tacoRepo = Mockito.mock(TacoRepository.class);

      when(tacoRepo.findAll()).thenReturn(tacoFlux);    //Mocks TacoRepository

      WebTestClient testClient = WebTestClient.bindToController(

         new DesignTacoController(tacoRepo))

         .build();        //Создание WebTestClient

      testClient.get().uri("/design/recent")

         .exchange()        //Запрашивает последние тако

         .expectStatus().isOk() //Проверяет ожидаемый ответ

         .expectBody()

         .jsonPath("$").isArray()

         .jsonPath("$").isNotEmpty()

         .jsonPath("$[0].id").isEqualTo(tacos[0].getId().toString())

         .jsonPath("$[0].name").isEqualTo("Taco 1").jsonPath("$[1].id")

         .isEqualTo(tacos[1].getId().toString()).jsonPath("$[1].name")

         .isEqualTo("Taco 2").jsonPath("$[11].id")

         .isEqualTo(tacos[11].getId().toString())

         ...

         .jsonPath("$[11].name").isEqualTo("Taco 12").jsonPath("$[12]")

         .doesNotExist();

         .jsonPath("$[12]").doesNotExist();

   }

   ...

}

Первое, что должен сделать метод shouldReturnRecentTacos(), - это установить тестовые данные в форме Flux<Taco>. Этот Flux затем предоставляется как возвращаемое значение из метода findAll() фиктивного (mock) TacoRepository.

Что касается Taco объектов, которые будут опубликованы Flux, они создаются с помощью служебного метода testTaco(), который при присвоении номера создает объект Taco, ID и имя которого основаны на этом числе. Метод testTaco() реализован следующим образом:

private Taco testTaco(Long number) {

   Taco taco = new Taco();

   taco.setId(UUID.randomUUID());

   taco.setName("Taco " + number);

   List<IngredientUDT> ingredients = new ArrayList<>();

   ingredients.add(

      new IngredientUDT("INGA", "Ingredient A", Type.WRAP));

   ingredients.add(

      new IngredientUDT("INGB", "Ingredient B", Type.PROTEIN));

   taco.setIngredients(ingredients);

   return taco;

}

Для простоты все тестовые тако будут иметь одинаковые два ингредиента. Но их ID и имя будут определяться по указанному номеру.

Между тем, вернувшись в метод shouldReturnRecentTacos(), вы создали экземпляр DesignTacoController, внедрив в конструктор фиктивный TacoRepository. Контроллер передается в WebTestClient.bindToController() для создания экземпляра WebTestClient.

Завершив настройку, вы теперь готовы использовать WebTestClient для отправки запроса GET в /design/recent и проверки того, что ответ соответствует вашим ожиданиям. Вызов get().uri("/design/recent") описывает запрос, который вы хотите выполнить. Затем вызов метода exchange() отправляет запрос, который будет обработан контроллером, с которым связан WebTestClient - DesignTacoController.

Наконец, вы можете подтвердить, что ответ соответствует ожиданиям. Вызывая waitStatus(), вы утверждаете, что ответ имеет код состояния HTTP 200 (OK). После этого вы видите несколько вызовов jsonPath(), которые задают условия, что JSON в теле ответа имеет значения, которые он должен иметь. Последнее условие проверяет, что 12-й элемент (в массиве, начинающемся с нуля) не существует, поскольку в результате никогда не должно быть более 12 элементов.

Если JSON возвращается сложным, с большим количеством данных или сильно вложенными данными, это может быть утомительно использовать jsonPath(). Фактически, я пропустил многие вызовы jsonPath() в листинге 11.1, чтобы сэкономить место. В тех случаях, когда использование функции jsonPath() может быть неудобным, WebTestClient предлагает функцию json(), которая принимает параметр String, содержащий JSON, для сравнения ответа с ним.

Например, предположим, что вы создали полный ответ JSON в файле с именем recent-tacos.json и поместили его в путь к классам с путем /tacos. Затем вы можете переписать условия WebTestClient, чтобы они выглядели так:

ClassPathResource recentsResource =

   new ClassPathResource("/tacos/recent-tacos.json");

String recentsJson = StreamUtils.copyToString(

   recentsResource.getInputStream(), Charset.defaultCharset());

testClient.get().uri("/design/recent")

   .accept(MediaType.APPLICATION\_JSON)

   .exchange()

   .expectStatus().isOk()

   .expectBody()

   .json(recentsJson);

Поскольку json() принимает String, вы должны сначала загрузить ресурс classpath в String. К счастью, Spring StreamUtils делает это играючи с copyToString(). String, возвращаемый функцией copyToString(), будет содержать весь JSON, который вы ожидаете получить в ответе на ваш запрос. Если передать его методу json(), контроллер получит правильный вывод.

Другой вариант, предлагаемый WebTestClient, позволяет сравнивать тело ответа со списком значений. МетодwellBodyList() принимает либо Class, либо ParameterizedTypeReference, указывающий тип элементов в списке, и возвращает объект ListBodySpec, для которого можно делать утверждения. Используя expectBodyList(), вы можете переписать тест, чтобы использовать подмножество тех же самых тестовых данных, которые вы использовали для создания моковского TacoRepository:

testClient.get().uri("/design/recent")

   .accept(MediaType.APPLICATION\_JSON)

   .exchange()

   .expectStatus().isOk()

   .expectBodyList(Taco.class)

      .contains(Arrays.copyOf(tacos, 12));

Здесь вы утверждаете, что тело ответа содержит список, содержащий те же элементы, что и первые 12 элементов исходного массива Taco, созданного вами в начале тестового метода.

**11.3.2 Тестирование POST-запросов**

WebTestClient может делать больше, чем просто проверять GET запросы в контроллерах. Его также можно использовать для тестирования любого метода HTTP, включая запросы GET, POST, PUT, PATCH, DELETE и HEAD. Таблица 11.1 отображает методы HTTP на методы WebTestClient.

Таблица 11.1. WebTestClient тестирует любые запросы к контроллерам Spring WebFlux.

HTTP методы : WebTestClient метод

GET : .get()

POST : .post()

PUT : .put()

PATCH : .patch()

DELETE : .delete()

HEAD : .head()

В качестве примера тестирования еще один метод тестирования HTTP-запроса к контроллеру Spring WebFlux, давайте посмотрим на другой тест с DesignTacoController. На этот раз вы напишете тест конечной точки создания taco вашего API, отправив POST запрос в /design:

@Test

public void shouldSaveATaco() {

   TacoRepository tacoRepo = Mockito.mock(

   TacoRepository.class);                    //Устанавливает тестовые данные

   Mono<Taco> unsavedTacoMono = Mono.just(testTaco(null));

   Taco savedTaco = testTaco(null);

   savedTaco.setId(1L);

   Mono<Taco> savedTacoMono = Mono.just(savedTaco);

   when(tacoRepo.save(any())).thenReturn(savedTacoMono); //Mocks TacoRepository

   WebTestClient testClient = WebTestClient.bindToController(  //Создание WebTestClient

      new DesignTacoController(tacoRepo)).build();

   testClient.post()            //POST для тако

      .uri("/design")

      .contentType(MediaType.APPLICATION\_JSON)

      .body(unsavedTacoMono, Taco.class)

      .exchange()

      .expectStatus().isCreated()            //Проверяет ответ

      .expectBody(Taco.class)

      .isEqualTo(savedTaco);

}

Как и в предыдущем методе тестирования, shouldSaveATaco начинается с настройки некоторых тестовых данных, моккинга TacoRepository, и создания WebTestClient, который привязан к контроллеру. Затем он использует WebTestClient для отправки POST запроса в /design с телом типа application/json и полезной нагрузкой, которая является JSON-сериализованной формой Taco в несохраненном Mono. После выполнения exchange() тест утверждает, что ответ имеет статус HTTP 201 (CREATED) и полезную нагрузку в теле, равную сохраненному объекту Taco.

**11.3.3. Тестирование на живом сервере**

Тесты, которые вы написали до сих пор, основывались на моковской реализации среды Spring WebFlux, так что реальный сервер не понадобился бы. Но вам может потребоваться протестировать контроллер WebFlux в контексте сервера, такого как Netty или Tomcat, и, возможно, с хранилищем или другими зависимостями. То есть вы должны быть способны написать интеграционный тест.

Чтобы написать интеграционный тест WebTestClient, начните с аннотирования класса теста с помощью @RunWith и @SpringBootTest, как и любого другого интеграционного теста Spring Boot:

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT)

public class DesignTacoControllerWebTest {

   @Autowired

   private WebTestClient testClient;

}

Установив для атрибута webEnvironment значение WebEnvironment.RANDOM\_PORT, вы просите Spring запустить работающий сервер, прослушивающий случайным образом выбранный порт (Вы могли бы также установить для webEnvironment значение WebEnvironment.DEFINED\_PORT и указать порт с атрибутом properties, но это обычно нежелательно. Это открывает риск столкновения портов с работающим сервером).

Вы, думаю, заметите, что вы также автоматически подключили WebTestClient к тестовому классу. Это означает не только то, что вам больше не нужно создавать его в своих методах тестирования, но и то, что вам не нужно указывать полный URL-адрес при отправке запросов. Это потому, что WebTestClient будет настроен, чтобы узнать, на каком порту работает тестовый сервер. Теперь вы можете переписать shouldReturnRecentTacos() в качестве интеграционного теста, в котором используется автоматическая привязка WebTestClient:

@Test

public void shouldReturnRecentTacos() throws IOException {

   testClient.get().uri("/design/recent")

      .accept(MediaType.APPLICATION\_JSON).exchange()

      .expectStatus().isOk()

      .expectBody()

      .jsonPath("$[?(@.id == 'TACO1')].name")

      .isEqualTo("Carnivore")

      .jsonPath("$[?(@.id == 'TACO2')].name")

      .isEqualTo("Bovine Bounty")

      .jsonPath("$[?(@.id == 'TACO3')].name")

      .isEqualTo("Veg-Out");

}

Вы, несомненно, заметили, что в этой новой версии shouldReturnRecentTacos() кода гораздо меньше. Больше нет необходимости создавать WebTestClient, потому что вы будете использовать экземпляр с автосвязыванием. И нет необходимости издеваться над TacoRepository, потому что Spring создаст экземпляр DesignTacoController и внедрит его в настоящий TacoRepository. В этой новой версии метода теста вы используете выражения JSONPath для проверки значений, передаваемых из базы данных.

WebTestClient полезен, когда в ходе теста вам нужно использовать API, предоставляемый контроллером WebFlux. Но что делать, когда само ваше приложение использует какой-то другой API? Давайте обратим наше внимание на клиентскую сторону реактивной веб-истории Spring и посмотрим, как WebClient предоставляет REST-клиент, который работает с реактивными типами, такими как Mono и Flux.

**11.4 Использование реактивного API REST**

В главе 7 вы использовали RestTemplate для отправки клиентских запросов в Taco Cloud API. RestTemplate является старым таймером, появившимся в Spring версии 3.0. В свое время он использовался для бесчисленных запросов от имени приложений, которые его используют.

Но все методы, предоставляемые RestTemplate, работают с нереактивными типами доменов и коллекциями. Это означает, что если вы хотите работать с данными ответа реактивным способом, вам нужно обернуть их в Flux или Mono. И если у вас уже есть Flux или Mono, и вы хотите отправить их в запросе POST или PUT, вам нужно будет извлечь данные в нереактивный тип, прежде чем делать запрос.

Было бы хорошо, если бы был способ использовать RestTemplate изначально с реактивными типами. Не бойтесь. Spring 5 предлагает WebClient в качестве реактивной альтернативы RestTemplate. WebClient позволяет отправлять и получать реактивные типы при отправке запросов на внешние API.

Использование WebClient сильно отличается от использования RestTemplate. Вместо того, чтобы иметь несколько методов для обработки различных типов запросов, WebClient имеет свободный интерфейс в стиле конструктора, который позволяет вам описывать и отправлять запросы. Общий шаблон использования для работы с WebClient:

-Создать экземпляр WebClient (или внедрить bean-компонент WebClient)

-Укажите HTTP метод запроса на отправку

-Укажите URI и любые заголовки, которые должны быть в запросе

-Отправить request

-Получить response

Давайте рассмотрим несколько примеров работы WebClient, начиная с того, как использовать WebClient для отправки HTTP-запросов GET.

**11.4.1 GET ресурсов**

В качестве примера использования WebClient, предположим, что вам нужно извлечь объект Ingredient по его идентификатору из Taco Cloud API. Используя RestTemplate, вы можете использовать метод getForObject(). Но с WebClient вы создаете запрос, получаете ответ, а затем извлекаете Mono, который публикует объект Ingredient:

Mono<Ingredient> ingredient = WebClient.create()

   .get()

   .uri("http://localhost:8080/ingredients/{id}", ingredientId)

   .retrieve()

   .bodyToMono(Ingredient.class);

ingredient.subscribe(i -> { ... })

Здесь вы создаете новый экземпляр WebClient с помощью create(). Затем вы используете get() и uri(), чтобы определить запрос GET для http://localhost:8080/ingredients/{id}, где заполнитель {id} будет заменен значением ingredientId.. Метод retrieve() выполняет запрос. Наконец, вызов bodyToMono() извлекает полезную нагрузку ответа в Mono<Ingredient>, к которому вы можете продолжить применять дополнительные операции Mono.

Чтобы применить дополнительные операции к Mono, возвращенному из bodyToMono(), важно подписаться на него еще до того, как запрос будет отправлен. Делать запросы, которые могут вернуть коллекцию значений, так же просто. Например, следующий фрагмент кода выбирает все ингредиенты:

Flux<Ingredient> ingredients = WebClient.create()

   .get()

   .uri("http://localhost:8080/ingredients")

   .retrieve()

   .bodyToFlux(Ingredient.class);

ingredients.subscribe(i -> { ... })

В большинстве случаев выборка нескольких элементов аналогична отправке запроса на один элемент. Большая разница в том, что вместо того, чтобы использовать bodyToMono() для извлечения тела ответа в Mono, вы используете bodyToFlux() для извлечения его во Flux.

Как и в случае с bodyToMono(), Flux, возвращенный из bodyToFlux(), еще не подписан. Это позволяет применять дополнительные операции (фильтры, map-ы и т. д.) к Flux до того, как данные начнут проходить через него. Поэтому важно подписаться на получившийся Flux, иначе запрос даже не будет отправлен.

**ВЫПОЛНЕНИЕ ЗАПРОСОВ С БАЗОВЫМ URI**

Вы можете использовать общий базовый URI для множества разных запросов. В этом случае может быть полезно создать bean-компонент WebClient с базовым URI и внедрить его везде, где это необходимо. Такой bean может быть объявлен так:

@Bean

public WebClient webClient() {

   return WebClient.create("http://localhost:8080");

}

Затем в любом месте, где вам нужно сделать запросы с использованием этого базового URI, WebClient компонент может быть внедрен и использован следующим образом:

@Autowired

WebClient webClient;

public Mono<Ingredient> getIngredientById(String ingredientId) {

   Mono<Ingredient> ingredient = webClient

      .get()

      .uri("/ingredients/{id}", ingredientId)

      .retrieve()

      .bodyToMono(Ingredient.class);

   ingredient.subscribe(i -> { ... })

}

Поскольку WebClient уже был создан, вы можете сразу приступить к работе, вызвав get(). Что касается URI, то при вызове uri() необходимо указать только путь относительно базового URI.

**ТАЙМ-АУТ ДЛЯ ДЛИТЕЛЬНЫХ ЗАПРОСОВ**

Единственное, на что вы можете рассчитывать, - это то, что сети не всегда надежны или так быстры, как вы ожидаете. Или, может быть, удаленный сервер вяло обрабатывает запрос. В идеале, запрос к удаленной службе будет возвращен в разумные сроки. Но если нет, было бы здорово, если бы клиент не замер в ожидании ответа слишком долго.

Чтобы избежать задержки клиентских запросов медленной сетью или службой, можно использовать метод timeout() из Flux или Mono, чтобы ограничить время ожидания публикации данных. В качестве примера рассмотрим, как можно использовать функцию timeout() при извлечении данных ингредиента:

Flux<Ingredient> ingredients = WebClient.create()

   .get()

   .uri("http://localhost:8080/ingredients")

   .retrieve()

   .bodyToFlux(Ingredient.class);

ingredients

   .timeout(Duration.ofSeconds(1))

   .subscribe(

      i -> { ... },

      e -> {

         // ошибка тайм-аута обработки

})

Как вы можете видеть, прежде чем подписаться на Flux, вы вызвали timeout(), указав продолжительность 1 секунда. Если запрос может быть выполнен менее чем за 1 секунду, то нет никаких проблем. Но если запрос занимает больше 1 секунды, он истекает и вызывается обработчик ошибок, указанный в качестве второго параметра для subscribe().

**11.4.2 Отправка ресурсов**

Отправка данных с помощью WebClient не сильно отличается от получения данных. В качестве примера предположим, что у вас есть Mono<Ingredient> и вы хотите отправить запрос POST с Ingredient, опубликованным Mono, в URI с относительным путем /ingredients. Все, что вам нужно сделать, это использовать метод post() вместо get() и указать, что Mono будет использоваться для заполнения тела запроса путем вызова body():

Mono<Ingredient> ingredientMono = ...;

Mono<Ingredient> result = webClient

   .post()

   .uri("/ingredients")

   .body(ingredientMono, Ingredient.class)

   .retrieve()

   .bodyToMono(Ingredient.class);

result.subscribe(i -> { ... })

Если у вас нет Mono или Flux для отправки, но вместо этого есть объект домена, вы можете использовать syncBody(). Например, предположим, что вместо Mono<Ingredient> у вас есть Ingredient, который вы хотите отправить в теле запроса:

Ingedient ingredient = ...;

Mono<Ingredient> result = webClient

   .post()

   .uri("/ingredients")

   .syncBody(ingredient)

   .retrieve()

   .bodyToMono(Ingredient.class);

result.subscribe(i -> { ... })

Если вместо запроса POST вы хотите обновить Ingredient с помощью запроса PUT, вы вызываете put() вместо post() и соответственно корректируете путь URI:

Mono<Void> result = webClient

   .put()

   .uri("/ingredients/{id}", ingredient.getId())

   .syncBody(ingredient)

   .retrieve()

   .bodyToMono(Void.class)

   .subscribe();

Запросы PUT обычно имеют пустые полезные нагрузки ответа, поэтому вы должны указать bodyToMono() вернуть Mono типа Void. При подписке на этот Mono, запрос будет отправлен.

**11.4.3 Удаление ресурсов**

WebClient также позволяет удалять ресурсы с помощью метода delete(). Например, следующий код удаляет ингредиент для переданого ID:

Mono<Void> result = webClient

   .delete()

   .uri("/ingredients/{id}", ingredientId)

   .retrieve()

   .bodyToMono(Void.class)

   .subscribe();

Как и в случае PUT запросов, запросы DELETE обычно не имеют полезной нагрузки. Вы снова возвращаетесь и подписываетесь на Mono<Void>, чтобы отправить запрос.

**11.4.4 Обработка ошибок**

До сих пор все примеры WebClient предполагали удачное завершение; не было ответов с кодами состояния 400 или 500. Если будет возвращен любой из статусов ошибок, WebClient зарегистрирует ошибку; в противном случае, он будет молча игнорировать это.

Если вам нужно обработать такие ошибки, то вызов onStatus() можно использовать для указания того, как должны обрабатываться различные коды состояния HTTP. onStatus() принимает две функции: функцию предиката, которая используется для соответствия статусу HTTP, и функцию, которая, учитывая объект ClientResponse, возвращает Mono<Throwable>.

Чтобы продемонстрировать, как onStatus() может быть использован для создания пользовательского обработчика ошибок, рассмотрим следующее использование WebClient, целью которого является извлечение ингредиента с учетом его идентификатора:

Mono<Ingredient> ingredientMono = webClient

   .get()

   .uri("http://localhost:8080/ingredients/{id}", ingredientId)

   .retrieve()

   .bodyToMono(Ingredient.class);

До тех пор значение в идентификаторе ингредиента соответствует известному ресурсу ингредиента, то результирующий Mono опубликует объект ингредиента, когда он будет подписан. Но что произойдет, если не будет подходящего ингредиента?

При подписке на Mono или Flux, который может закончиться ошибкой, важно зарегистрировать получателя ошибок, а также получателя данных в вызове метода subscribe():

ingredientMono.subscribe(

ingredient -> {

// обрабатывать данные ингредиента

...

},

error-> {

// разобраться с ошибкой

...

});

Если ресурс ингредиента найден, то первая лямбда (потребитель данных), предоставленная subscribe(), вызывается с соответствующим объектом ингредиента. Но если он не найден, то запрос отвечает кодом состояния HTTP 404 (не найден), что приводит к тому, что вторая лямбда (потребитель ошибок) по умолчанию получает исключение WebClientResponseException.

Самая большая проблема с WebClientResponseException заключается в том, что он довольно неспецифичен относительно того, что могло пойти не так, чтобы вызвать сбой Mono. Его название предполагает, что была ошибка в ответе на запрос, сделанный WebClient, но вам нужно изучить исключение WebClientResponseException, чтобы узнать, что пошло не так. И в любом случае было бы неплохо, если бы исключение, предоставленное потребителю ошибок, было более специфичным для домена, а не для WebClient.

Добавляя пользовательский обработчик ошибок, вы можете предоставить код, который преобразует код состояния в Throwable по вашему выбору. Предположим, что вы хотите, чтобы неудачный запрос для ресурса ингредиента привел к тому, что Mono завершилось ошибкой с UnknownIngredientException. Вы можете добавить вызов onStatus() после вызова retrieve(), чтобы добиться этого:

Mono<Ingredient> ingredientMono = webClient

   .get()

   .uri("http://localhost:8080/ingredients/{id}", ingredientId)

   .retrieve()

   .onStatus(HttpStatus::is4xxClientError,

   response -> Mono.just(new UnknownIngredientException()))

   .bodyToMono(Ingredient.class);

Первый аргумент в вызове onStatus() - это предикат, задающий состояние Http и возвращающий значение true, если требуется обработать код состояния. И если код состояния совпадает, то ответ будет возвращен функции во втором аргументе для обработки, как он считает нужным, в конечном итоге возвращая Mono типа Throwable.

В этом примере, если код состояния представляет собой код состояния уровня 400 (например, ошибка клиента), Mono будет возвращен с UnknownIngredientException.  Это приводит к тому, что ingredientMono терпит неудачу с этим исключением.

Обратите внимание, что HttpStatus::is4xxClientError - это ссылка на метод is4xxClientError метода HttpStatus. Именно этот метод будет вызван для данного HttpStatus объекта. Если вы хотите, вы можете использовать другой метод в HttpStatus в качестве ссылки на метод; или вы можете предоставить свою собственную функцию в виде ссылки на лямбду или метод, которая возвращает boolean.

Например, вы можете получить еще более точную обработку ошибок, специально проверив состояние HTTP 404 (НЕ НАЙДЕНО), изменив вызов onStatus(), чтобы он выглядел следующим образом:

Mono<Ingredient> ingredientMono = webClient

   .get()

   .uri("http://localhost:8080/ingredients/{id}", ingredientId)

   .retrieve()

   .onStatus(status -> status == HttpStatus.NOT\_FOUND,

      response -> Mono.just(new UnknownIngredientException()))

   .bodyToMono(Ingredient.class);

Стоит также отметить, что вы можете иметь столько вызовов onStatus(), сколько вам нужно для обработки любых кодов состояния HTTP, которые могут возвращаться в ответе.

**11.4.5 Обмен запросами**

До этого момента вы использовали метод retrieve() для обозначения отправки запроса при работе с WebClient. В этих случаях метод retrieve() возвращает объект типа ResponseSpec, с помощью которого можно обрабатывать ответ с вызовами таких методов, как onStatus(), bodyToFlux() и bodyToMono(). Работа со спецификацией ответа хороша для простых случаев, но она несколько ограничена. Например, если вам нужен доступ к заголовкам ответа или значениям cookie, ResponseSpec вам не подойдет.

Когда ResponseSpec заканчивается, вы можете попробовать вызвать exchange() вместо retrieve(). Метод exchange() возвращает Mono типа ClientResponse, к которому можно применить реактивные операции для проверки и использования данных из всего ответа, включая полезную нагрузку, заголовки и файлы cookie.

Прежде чем мы рассмотрим, что отличает exchange() от retrieve(), давайте начнем с того, насколько они похожи. Следующий фрагмент кода использует WebClient и exchange() для извлечения одного ингредиента по идентификатору:

Mono<Ingredient> ingredientMono = webClient

   .get()

   .uri("http://localhost:8080/ingredients/{id}", ingredientId)

   .exchange()

   .flatMap(cr -> cr.bodyToMono(Ingredient.class));

Это примерно эквивалентно следующему примеру, который использует retrieve():

Mono<Ingredient> ingredientMono = webClient

   .get()

   .uri("http://localhost:8080/ingredients/{id}", ingredientId)

   .retrieve()

   .bodyToMono(Ingredient.class);

В примере exchange() вместо использования ResponseSpec объекта bodyToMono() для получения Mono<Ingredient> вы получаете Mono<ClientResponse>, к которому можно применить функцию плоского отображения для сопоставления ClientResponse с Mono<Ingredient>, который сглаживается в результирующий Mono.

Теперь давайте посмотрим, что отличает exchange(). Предположим, что ответ на запрос может включать заголовок с именем X\_UNAVAILABLE со значением true, указывающим на то, что (по какой-то причине) рассматриваемый ингредиент недоступен. И для обсуждения, предположим, что если этот заголовок существует, вы хотите, чтобы результирующий Mono был пустым — ничего не возвращать. Вы можете достичь этого сценария, добавив еще один вызов flatMap() таким образом, чтобы весь вызов веб-клиента выглядел следующим образом:

Mono<Ingredient> ingredientMono = webClient

   .get()

   .uri("http://localhost:8080/ingredients/{id}", ingredientId)

   .exchange()

   .flatMap(cr -> {

      if (cr.headers().header("X\_UNAVAILABLE").contains("true")) {

         return Mono.empty();

      }

      return Mono.just(cr);

   })

   .flatMap(cr -> cr.bodyToMono(Ingredient.class));

Новый вызов flatMap() проверяет заголовки данного объекта запроса клиента, ища заголовок с именем X\_UNAVAILABLE со значением true. Если он найден, он возвращает пустое Mono. В противном случае он возвращает новое Mono, содержащее ответ клиента. В любом случае, возвращенный Mono будет сглажен в Mono, с которым будет работать следующий вызов flatMap().

**11.5 Securing reactive web APIs**

Пока существует Spring Security (и даже до этого, когда он был известен как Acegi Security), его модель веб-безопасности была построена вокруг фильтров сервлетов. Ну, это имеет смысл. Если вам нужно перехватить запрос, привязанный к веб framework-у на основе сервлета, чтобы гарантировать, что у инициатора запроса есть надлежащие полномочия, фильтр сервлета является очевидным выбором. Но But Spring WebFlux вносит изменения в этот подход.

При написании веб-приложения с использованием Spring WebFlux нет никакой гарантии, что сервлеты будут задействованы. Фактически, реактивное веб-приложение, скорее всего, будет построено на Netty или каком-либо другом сервере, не являющемся сервлетом. Означает ли это, что Spring Security на основе фильтра сервлетов нельзя использовать для защиты приложений Spring WebFlux?

Это правда, что с помощью сервлетов, фильтров не подойдет при защите приложения Spring WebFlux. Но Spring Security все еще справляется с этой задачей. Начиная с версии 5.0.0, Spring Security можно использовать для защиты Spring MVC на основе сервлетов и реактивных приложений Spring WebFlux. Для этого используется Spring WebFilter, Spring-специфичный фильтр сервлетов, который не требует зависимости от API сервлета.

Что еще более примечательно, так это то, что модель конфигурации для реактивного Spring Security не сильно отличается от того, что вы видели в главе 4. Фактически, в отличие от Spring WebFlux, который имеет отдельную зависимость от Spring MVC, Spring Security является тот же стартер безопасности Spring Boot, независимо от того, собираетесь ли вы использовать его для защиты веб-приложения Spring MVC или приложения, написанного с использованием Spring WebFlux. Как напоминание, вот как выглядит security стартер:

<dependency>

   <groupId>org.springframework.boot</groupId>

   <artifactId>spring-boot-starter-security</artifactId>

</dependency>

Тем не менее, есть несколько небольших различий между моделями реактивной и нереактивной конфигурации Spring Security. Стоит взглянуть на сравнение двух моделей конфигурации.

**11.5.1 Настройка реактивного web security**

Напомним, что настройка Spring Security для защиты веб-приложения Spring MVC обычно включает в себя создание нового класса конфигурации, расширяющего WebSecurityConfigurerAdapter и снабженного аннотацией @EnableWebSecurity. Такой класс конфигурации переопределяет метод configuration() для указания специфики web security, например, какие полномочия требуются для определенных путей запроса. Следующий простой класс конфигурации Spring Security служит напоминанием о том, как настроить безопасность для нереактивного приложения Spring MVC:

@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

   @Override

   protected void configure(HttpSecurity http) throws Exception {

      http

         .authorizeRequests()

         .antMatchers("/design", "/orders").hasAuthority("USER")

         .antMatchers("/\*\*").permitAll();

   }

}

Теперь давайте посмотрим, как эта же конфигурация может выглядеть для реактивного приложения Spring WebFlux. В следующем списке показан класс конфигурации реактивной безопасности, который примерно эквивалентен простой конфигурации безопасности из предыдущих версий.

**Листинг 11.2. Настройка Spring Security для приложения Spring WebFlux**

@Configuration

@EnableWebFluxSecurity

public class SecurityConfig {

   @Bean

   public SecurityWebFilterChain securityWebFilterChain(

            ServerHttpSecurity http) {

      return http

         .authorizeExchange()

         .pathMatchers("/design", "/orders").hasAuthority("USER")

         .anyExchange().permitAll()

         .and()

         .build();

   }

}

Как вы можете видеть, есть много того, что знакомо, но в то же время отличающегося. Вместо @EnableWebSecurity этот новый класс конфигурации аннотируется с помощью @EnableWebFluxSecurity. Более того, класс конфигурации не расширяет WebSecurityConfigurerAdapter или любой другой базовый класс вообще. Поэтому он также не переопределяет методы configure().

Вместо метода configure() объявляется компонент типа SecurityWebFilterChain с помощью метода securityWebFilterChain(). Тело securityWebFilterChain() не сильно отличается от предыдущих конфигураций метода configure(), но есть некоторые тонкие изменения.

Прежде всего, конфигурация объявляется с использованием заданного объекта ServerHttpSecurity вместо объекта HttpSecurity. Используя данный ServerHttpSecurity, вы можете вызвать authorizeExchange(), который примерно эквивалентен authorizeRequests(), чтобы задать безопасность на уровне запросов.

**ПРИМЕЧАНИЕ**ServerHttpSecurity является новинкой в Spring Security 5 и является реактивным аналогом HttpSecurity.

При сопоставлении путей вы все равно можете использовать подстановочные пути в стиле Ant, но делайте это с помощью метода pathMatchers() вместо antMatchers(). И для удобства вам больше не нужно указывать универсальный путь в Ant-стиле для /\*\*, потому что anyExchange() возвращает все, что вам нужно.

Наконец, поскольку вы объявляете SecurityWebFilterChain как bean, а не переопределяете метод фрэймворка, вы должны вызвать метод build(), чтобы собрать все правила безопасности в возвращаемом SecurityWebFilterChain.

Помимо этих небольших различий, настройка веб-безопасности ничем не отличается для Spring WebFlux от Spring MVC. Но как насчет пользовательских данных?

**11.5.2 Конфигурирование службы реактивных данных пользователя**

Расширяя WebSecurityConfigurerAdapter, вы переопределяете один метод configure() для объявления правил веб-безопасности и другой метод configure() для настройки логики аутентификации, обычно путем определения объекта UserDetails. В качестве напоминания о том, как это выглядит, рассмотрим следующий переопределенный метод configure(), который использует внедренный объект UserRepository в анонимной реализации UserDetailsService для поиска пользователя по имени пользователя:

@Autowired

UserRepository userRepo;

@Override

protected void

      configure(AuthenticationManagerBuilder auth)

      throws Exception {

   auth

      .userDetailsService(new UserDetailsService() {

         @Override

         public UserDetails loadUserByUsername(String username)

               throws UsernameNotFoundException {

            User user = userRepo.findByUsername(username)

            if (user == null) {

               throw new UsernameNotFoundException(

                  username " + not found")

            }

            return user.toUserDetails();

         }

   });

}

В этой нереактивной конфигурации вы переопределяете единственный метод, требуемый UserDetailsService, loadUserByUsername(). Внутри этого метода вы используете внедренный UserRepository для поиска пользователя по заданному имени пользователя. Если имя не найдено, вы бросаете исключение UsernameNotFoundException. Но если он найден, то вы вызываете вспомогательный метод toUserDetails() для возврата результирующего объекта UserDetails.

В реактивной конфигурации безопасности вы не переопределяете метод configure(). Вместо этого вы объявляете bean-компонент ReactiveUserDetailsService. ReactiveUserDetailsService является реактивным эквивалентом UserDetailsService. Как и UserDetailsService, ReactiveUserDetailsService требует реализации только одного метода. В частности, метод findByUsername() возвращает Mono<userDetails> вместо необработанного объекта UserDetails.

В следующем примере объявляется, что bean-компонент ReactiveUserDetailsService использует внедренный UserRepository, который предположительно является реактивным Spring Data репозиторием (о котором мы поговорим подробнее в следующей главе):

@Service

public ReactiveUserDetailsService userDetailsService(

      UserRepository userRepo) {

   return new ReactiveUserDetailsService() {

      @Override

      public Mono<UserDetails> findByUsername(String username) {

         return userRepo.findByUsername(username)

            .map(user -> {

               return user.toUserDetails();

            });

      }

   };

}

Здесь Mono<UserDetails> возвращается по мере необходимости, но метод UserRepository.findByUsername() возвращает Mono<User>. Поскольку это Mono, вы можете вызвать операции, такие как map(), чтобы преобразовать Mono<User> в Mono<UserDetails>.

В этом случае операция map() применяется с лямбда-выражением, которое вызывает вспомогательный метод toUserDetails() для объекта User, опубликованного Mono. Это преобразует User в UserDetails. Как следствие, операция .map() возвращает Mono<UserDetails>, который является именно тем, что требуется для ReactiveUserDetailsService.findByUsername().

**ИТОГО:**

*- Spring WebFlux предлагает реактивный веб-фреймворк, модель программирования которого соответствует модели Spring MVC, даже разделяя многие из тех же самых аннотаций.*

*- Spring 5 также предлагает функциональную модель программирования в качестве альтернативы Spring WebFlux.*

*- Реактивные контроллеры можно протестировать с помощью WebTestClient.*

*- На стороне клиента Spring 5 предлагает WebClient, реактивный аналог Spring RestTemplate.*

*- Хотя WebFlux имеет некоторые существенные изменения для базовых механизмов защиты веб-приложений, Spring Security 5 поддерживает реактивную безопасность с помощью модели программирования, которая не сильно отличается от нереактивных приложений Spring MVC.*