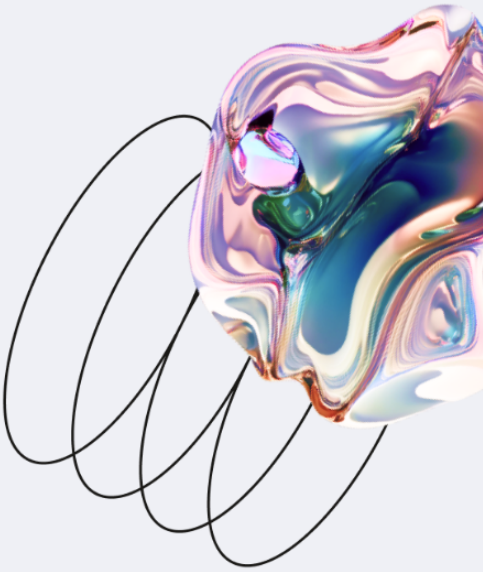
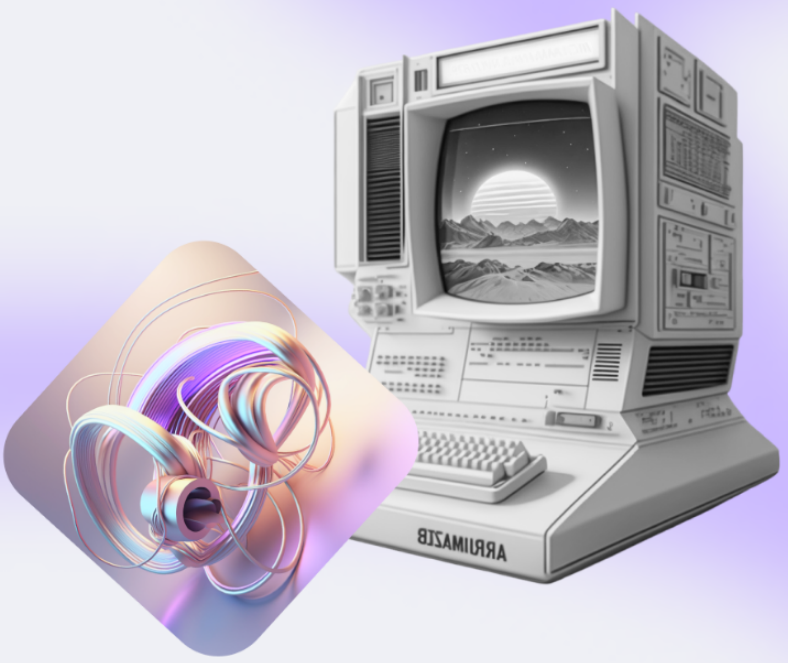
GeekBrains



Spring MVC. Использование шаблонизатора

Фреймворк Spring



Оглавление

Введение

Привет, программисты! Помните наш прошлый урок, где мы вдруг погрузились в волшебный мир Spring и решили создать свое первое серверное приложение? Ну что, готовы продолжить наше приключение? Перед тем как мы начнем, важно отметить, что наша цель - это сделать вас абсолютными гуру Spring. Ведь это ключевой навык любого Java разработчика. Итак, давайте начнем!

На предыдущем уроке мы изучили основы Spring, познакомились с контроллерами, запросами и ответами. Мы даже научились настраивать базовые вещи, такие как подключение к базе данных. Однако, было бы неправильно остановиться на этом. Ведь существует столько всего интересного в Spring!

Сегодня наша тема — Spring MVC. Это подмодуль Spring Framework, который предоставляет инфраструктуру для разработки веб-приложений. MVC, если вы не помните, означает Model-View-Controller. Это архитектурный паттерн, который позволяет разделить логику приложения на три основных компонента: данные

(Model), представление этих данных (View) и действия, которые можно выполнить с данными (Controller). Мы уже столкнулись с контроллерами, и немного затронули модели и представления, но сегодня мы углубимся в их изучение еще больше.

И да, еще одна вещь, которую мы будем изучать сегодня - это шаблонизатор Thymeleaf. Вероятно, у вас возник вопрос, что это такое и зачем оно нам нужно? Представьте, что у вас есть данные и вы хотите отобразить их в HTML. Вы бы могли написать все вручную, но что если у вас много данных, которые нужно отобразить разными способами? Именно тут на помощь приходят шаблонизаторы. Они позволяют нам создавать динамические HTML страницы, используя наши данные.

Также мы научимся работать с ресурсами в Spring. Ресурсы могут включать в себя все: от статических HTML страниц до CSS, JavaScript и изображений. Spring предоставляет удобные инструменты для работы с ними.

В общем, у нас есть много вещей, которые нужно изучить и обсудить. Уверен, что этот урок будет насыщенным и полезным, и вы обретете массу новых знаний. Но не волнуйтесь, мы пройдем через все это вместе, шаг за шагом.

Перед тем как углубиться в практическую часть, немного поговорим о том, почему изучение Spring MVC так важно. Я знаю, что многие из вас, слыша слова “архитектурный паттерн”, “шаблонизатор” и “ресурсы”, могут подумать: “Ээээ, а зачем нам все это?” Однако, поверьте мне, это крайне важные элементы для веб-разработки.

Начнем с MVC, который является ключевым аспектом разработки веб-приложений. Суть MVC в том, что он делит приложение на три основные части: модель, представление и контроллер. Это помогает нам, разработчикам, лучше организовывать код и упрощает его поддержку и расширение.

Представьте, что вы строите дом. MVC - это как разделение ответственности между бригадами строителей. Одни отвечают за фундамент и стены (это наша модель, основа данных), другие - за интерьер и декор (это наше представление, как данные отображаются), и еще одна группа - за электрику и водоснабжение (это наши контроллеры, механизмы для взаимодействия с данными). Если не соблюдать это разделение, то работа будет вестись хаотично, и в итоге будет трудно вносить изменения или исправлять ошибки. То же самое и с веб-разработкой.

Теперь о шаблонизаторе Thymeleaf. Почему он так важен? Давайте представим, что ваши данные — это ингредиенты для блюда, а HTML-страница - это уже готовое блюдо. Thymeleaf — это ваш шеф-повар, который берет эти ингредиенты и превращает их в нечто вкусное и привлекательное для глаз. Без него вам придется

самостоятельно приготовить все блюда, что займет много времени и требует специальных навыков.

Наконец, работа с ресурсами в Spring. Помните, наши веб-приложения — это не просто код. Это целый мир, в который погружается пользователь. И этот мир состоит не только из данных, но и из изображений, стилей, скриптов и многого другого. Научившись работать с ресурсами в Spring, вы сможете создать более гибкие и динамичные веб-приложения.

В общем, уверяю вас, изучение Spring MVC с его шаблонизатором Thymeleaf и работой с ресурсами — это нечто, что усилит ваши навыки разработки и сделает вас более ценным специалистом на рынке труда.

Термины, используемые в лекции

Spring MVC — Это фреймворк на базе Spring, который предлагает модель-вид-контроллер (MVC) архитектуру и готовые компоненты, которые могут быть использованы для разработки гибких и свободно масштабируемых веб-приложений.

Thymeleaf — Это современный серверный Java-шаблонизатор, способный обрабатывать как HTML, так и XML. Он хорошо интегрируется с Spring MVC и обеспечивает полноценную поддержку HTML5.

Контроллер (Controller) — В контексте Spring MVC, контроллер - это класс, который обрабатывает веб-запросы от клиента. Контроллеры обычно аннотированы @Controller.

Модель (Model) — Модель представляет собой данные, которые будут отображаться пользователю. Модель может быть любым Java объектом, который может быть сохранен в базе данных.

Вид (View) — В Spring MVC вид представляет собой то, что будет отображаться пользователю. Это может быть JSP-страница, HTML-страница, PDF-документ, Excel-документ и т.д.

DispatcherServlet — Это сердце Spring MVC, которое обрабатывает входящие запросы и маршрутизирует их к соответствующим контроллерам.

WebApplicationContext — Специфический для веб-приложений контекст Spring, который предоставляет конфигурацию для приложения в виде bean-компонентов.

Bean — В контексте Spring, bean это объект, управляемый Spring IoC контейнером.

RequestMapping — Это аннотация Spring MVC, которая используется для сопоставления (mapping) веб-запросов с определенными методами обработчика (контроллеров) в вашем приложении.

Бэкенд и возврат готовых страниц

Хотя мы живем в век цифровых технологий, когда большинство вещей переходят в интернет, не всегда очевидно, как работают наши любимые веб-приложения и сайты. Важным аспектом работы большинства веб-систем является процесс возврата готовых страниц с сервера, что часто выполняет бэкенд.

Давайте погрузимся в эту тему поближе. Что происходит, когда вы вводите адрес веб-сайта в браузере? После нажатия Enter ваш запрос отправляется на сервер, где бэкенд-приложение обрабатывает его и отправляет обратно ответ. Этот ответ, в основном, представляет собой HTML-страницу, которую браузер затем отображает для вас.

Бэкенд-приложение, в свою очередь, обычно создает эту страницу на основе шаблона. Например, вы представляете онлайн-магазин, и пользователь хочет посмотреть страницу с товарами. Ваш бэкенд получает этот запрос, обращается к базе данных, чтобы получить информацию о товарах, и затем использует шаблон страницы товаров, чтобы заполнить его актуальной информацией. Вот где Thymeleaf приходит на помощь, позволяя бэкенду легко вставлять данные в этот шаблон и создавать готовую страницу.

Вероятно, у вас есть вопрос: “А почему бы не отправлять просто данные и позволить фронтенду создавать страницу?” И это вполне разумный вопрос. На самом деле, именно так работают некоторые веб-приложения, особенно те, которые используют так называемые “одностраничные” фреймворки, такие как React или Angular. Однако, возврат готовых страниц с бэкенда имеет ряд преимуществ. Один из них - это быстродействие, потому что браузеру не нужно тратить время на создание

страницы из данных. Кроме того, бэкенд-приложения обычно более мощные и эффективные в работе с данными и шаблонами.

Разница между @Controller и @RestController

Прежде чем перейти к практическим аспектам создания веб-страниц с помощью Spring MVC и Thymeleaf, давайте поговорим о двух ключевых аннотациях, которые используются в Spring для создания контроллеров: @Controller и @RestController.

Вы уже знакомы с @RestController, так как мы использовали ее в прошлом уроке, когда создавали наше первое серверное приложение. Вспомните, @RestController используется для создания HTTP API. Каждый метод в классе, аннотированном как @RestController, обрабатывает HTTP-запрос и возвращает данные в формате, который браузер или любой другой клиент может использовать и обработать.

Однако, что происходит, когда мы хотим вернуть не просто данные, а полноценную веб-страницу, возможно, с некоторыми динамическими данными, вставленными в нее? Именно здесь мы начинаем использовать аннотацию @Controller.

Класс, помеченный аннотацией @Controller, подобен @RestController, но с одним ключевым отличием. Вместо того чтобы просто возвращать данные, @Controller обрабатывает HTTP-запросы и возвращает имя вида (view), которое затем используется для генерации HTML-страницы.

Представьте, что ваше приложение — это ресторан, а контроллеры - это официанты. @RestController — это официант, который приносит вам сырой ингредиент (данные). Вам предстоит самому приготовить из него блюдо (HTML-страница). С другой стороны, @Controller — это официант, который приносит вам уже готовое блюдо (HTML-страница с вставленными данными).

Поэтому, когда мы хотим возвратить готовую страницу с сервера, мы будем использовать @Controller. Это позволяет нам легко интегрировать наш код с Thymeleaf и другими технологиями представления, чтобы создать динамичные веб-страницы.

HTML

Что ж, теперь давайте немного поговорим об HTML. Вы когда-нибудь строили замок из детских конструкторов, например, LEGO? HTML очень похож на это. Но вместо кирпичиков LEGO мы используем специальные блоки, которые называются тегами.

HTML - это язык, который используют для создания веб-страниц. Все, что вы видите в интернете, например, на этой странице, сначала было написано на HTML. Каждая веб-страница начинается с простого HTML-документа.

Представьте, что HTML-страница — это дом. Все начинается с основания, которое в HTML называется DOCTYPE. Это как бы говорит браузеру: “Эй, мы строим дом по этому чертежу!”.

Затем у нас идет коробка, которую мы называем <html>. Это как основа нашего дома, куда мы будем ставить все наши блоки.

Внутри этой коробки есть две основные части: <head> и <body>. <head> - это как чердак нашего дома. Мы не видим его, когда смотрим на дом снаружи, но он содержит важные вещи, такие как название нашего дома (или веб-страницы), стили интерьера и так далее.

<body> — это основная часть дома, где мы живем. Это все, что мы видим, когда открываем веб-страницу: тексты, изображения, кнопки и т.д.

Теперь, что насчет этих блоков или тегов, о которых я говорил? Ну, они очень похожи на разные типы кирпичиков LEGO. У нас есть разные типы тегов для разных вещей. Например, <p> для абзацев текста, <h1> для больших заголовков, <img> для картинок и многое другое.

Так что, если вы уже умеете строить замки из LEGO, вы абсолютно готовы научиться строить веб-страницы с помощью HTML!

Давайте начнем с тега <title>, который мы помещаем внутрь <head>. Это название нашего дома, которое вы видите в верхней части браузера, когда открываете веб-страницу. Вот как это выглядит:

```
<head>
    <title>Мой первый дом</title>
</head>
```

Теперь перейдем к <body>, где происходит вся магия. Один из самых простых тегов - это <p>, который используется для создания абзацев текста. Например:

```
<body>
      <p>Привет! Добро пожаловать в мой новый дом. Здесь очень уютно.</p>
</body>
```

А что насчет заголовков? Для этого у нас есть теги <h1> до <h6>, где <h1> - это самый большой заголовок, а <h6> - самый маленький. Это как разные размеры дверей в нашем доме. Вот как это выглядит:

```
<body>
    <h1>Добро пожаловать в мой дом!</h1>
    <h2>Вот что внутри:</h2>
    <h3>Гостиная</h3>
    <p>Здесь у нас очень удобный диван и большой телевизор.</p>
    <h3>Кухня</h3>
    <p>Здесь мы готовим вкусные блюда.</p>
</body>
```

И конечно, нельзя забыть про картинки. Мы используем тег <img> и атрибут src (который означает источник), чтобы указать, где находится наша картинка. Это как вешать картины на стены:

```
<body>
    <img src="beautiful_house.jpg">
</body>
```

Вот так, используя эти и многие другие теги, мы можем строить наши веб-страницы, как дети строят замки из конструктора. Помните, что каждый тег имеет свое назначение и помогает нам организовать и представить наши данные.

Теперь, когда мы знаем основы HTML, давайте разберемся, как можно создать простое Spring приложение, которое возвращает статичную HTML страницу. Сначала мы поговорим о теории, а затем перейдем к практической части.

Когда мы говорим о статическом контенте в контексте веб-разработки, мы обычно имеем в виду файлы, которые не изменяются динамически на сервере перед тем, как они доставляются клиенту. Это могут быть HTML-файлы, CSS-стили, JavaScript-скрипты, изображения и так далее. В Spring Boot статические файлы обычно размещаются в директории src/main/resources/static. Все файлы, размещенные в этой директории, автоматически становятся доступными как статический контент вашего веб-приложения.

Важно понимать, что возвращение статической страницы означает, что она будет одинакова для всех пользователей. В отличие от динамической страницы, которая

может быть изменена на сервере для каждого запроса (например, для отображения приветствия с именем пользователя), статическая страница остается неизменной.

Теперь, когда мы знаем теорию, давайте перейдем к практической части. Первое, что нам нужно сделать, это создать новое Spring Boot приложение. Мы можем сделать это с помощью Spring Initializr на сайте start.spring.io или через вашу любимую среду разработки, которая поддерживает Spring Boot (например, IntelliJ IDEA, Eclipse и др.).

После создания нового проекта мы добавим HTML-файл в директорию src/main/resources/static. Давайте назовем его index.html, и он будет выглядеть примерно так:

```
<!DOCTYPE html>
<html>
<head>
    <title>Мое первое Spring приложение</title>
</head>
<body>
    <h1>Привет, мир!</h1>
    <p>Добро пожаловать в мое первое Spring приложение!</p>
</body>
</html>
```

Затем, в нашем Spring приложении, нам нужно создать новый контроллер. Для этого мы создадим новый класс и пометим его аннотацией @Controller. В этом контроллере мы создадим метод, который будет обрабатывать запросы к корневому URL (/) и возвращать имя нашего HTML-файла.

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {

    @RequestMapping("/")
    public String home() {
        return "index";
    }
}
```

И вот, у нас есть простое Spring приложение, которое возвращает статическую HTML-страницу! Если вы запустите приложение и откроете браузер по адресу localhost:8080, вы увидите страницу, которую мы только что создали.

Теперь, когда мы знаем, как создать простую статическую страницу в Spring приложении, давайте поговорим о некоторых дополнительных аспектах этого процесса.

Одна из особенностей статических страниц, о которой важно знать, — это то, что они не обрабатываются на сервере, а передаются браузеру “как есть”. Это означает, что если вы хотите внести какие-то изменения в страницу для разных пользователей или в зависимости от каких-то других условий, вам нужно делать это с помощью JavaScript на стороне клиента. Однако в некоторых случаях это не так удобно, как использование серверной обработки страниц, которую мы будем изучать позже с помощью Thymeleaf.

Но несмотря на эти ограничения, статические страницы все равно очень полезны. Они быстро загружаются и могут быть кэшированы браузером, что снижает нагрузку на ваш сервер. К тому же, для создания простых страниц или страниц, которые не требуют динамических изменений, статические страницы — это идеальное решение.

Давайте сейчас добавим немного динамизма на нашу статическую страницу с помощью JavaScript. Для этого нам нужно добавить тег <script> в наш HTML-файл. Этот тег позволяет включить JavaScript код прямо в HTML страницу.

```
<!DOCTYPE html>
<html>
<head>
    <title>Мое первое Spring приложение</title>
</head>
<body>
    <h1 id="greeting">Привет, мир!</h1>
    <p>Добро пожаловать в мое первое Spring приложение!</p>

    <script>
        document.getElementById('greeting').textContent = 'Привет, Spring!';
    </script>
</body>
</html>
```

В этом примере мы используем JavaScript для того, чтобы найти элемент с id “greeting” и изменить его текст на “Привет, Spring!”. Если вы обновите страницу в браузере, вы увидите, что текст заголовка изменился.

Таким образом, мы видим, что даже со статическими страницами мы можем добавить немного динамичности на клиентскую сторону. Но не забывайте, что для более сложной динамики и обработки данных на сервере нам потребуется использовать шаблонизатор, такой как Thymeleaf, о котором мы поговорим далее.

Как это работает в Spring

После того, как мы разобрались с основами HTML и создали наше первое Spring приложение, которое возвращает статическую страницу, давайте поговорим о том, как это все работает под капотом. Я постараюсь объяснить это так, чтобы это было понятно даже тем, кто не знаком с веб разработкой.

Представьте, что ваше веб-приложение — это большой магазин игрушек, а пользователь, который заходит на ваш сайт — это ребенок, который пришел выбрать игрушку.

Когда пользователь вводит адрес вашего сайта в браузер и нажимает Enter, это как будто ребенок входит в магазин и говорит: “Привет, я хочу игрушку!”. Это запрос от пользователя к вашему приложению.

Теперь ваше приложение, как хороший продавец, должно ответить на этот запрос. Если бы это был магазин, продавец мог бы спросить: “Какую игрушку вы хотите? Машинку или куклу?”. Но в веб-приложении у нас нет возможности спросить пользователя напрямую. Вместо этого наш продавец (приложение) смотрит на запрос и пытается понять, что от него хочет пользователь.

В Spring есть специальный компонент, который называется диспетчером (DispatcherServlet), и его работа — это как раз и есть определение, что нужно пользователю. Он как опытный продавец, который всегда знает, где что лежит и какую игрушку предложить ребенку.

Когда диспетчер понимает, что нужно пользователю, он ищет нужную страницу (или “игрушку”) среди всех контроллеров приложения. Это как если бы продавец пошел на склад и выбрал нужную игрушку из множества других. Каждый контроллер в Spring приложении — это как отдел в магазине, где хранятся определенные игрушки.

Когда нужная страница найдена, диспетчер возвращает ее обратно пользователю. Это как если бы продавец вернулся к ребенку и сказал: “Вот твоя игрушка!”. И теперь ребенок (пользователь) может радоваться своей новой игрушке (странице).

Так вот как работает механизм возврата веб-страниц в Spring. Но это еще не все! Давайте попробуем добавить на нашу страницу ссылку на другую страницу. Это будет как если бы продавец сказал ребенку: “Если тебе понравится эта игрушка, у нас есть еще много других в том отделе!”.

Для этого нам нужно добавить новый тег в наш HTML-файл - <a>. Этот тег создает ссылку на другую страницу. Вот как это выглядит:

```
<body>
    <h1 id="greeting">Привет, мир!</h1>
    <p>Добро пожаловать в мое первое Spring приложение!</p>
    <a href="/other-page">Посмотреть другую страницу</a>

    <script>
        document.getElementById('greeting').textContent = 'Привет, Spring!';
    </script>
</body>
```

Теперь у нас на странице есть ссылка на другую страницу. Когда пользователь кликает на нее, он отправляет новый запрос на адрес /other-page, и весь процесс повторяется снова.

Итак, мы разобрались, как работает механизм возврата веб-страниц в Spring и добавили ссылку на другую страницу. Я надеюсь, что это объяснение было понятно. На следующем уроке мы узнаем, как добавить в наше приложение еще больше интерактивности с помощью шаблонизатора Thymeleaf.

Шаблонизатор: художник веб-страниц

Теперь, когда мы знаем, как создать статическую страницу и как Spring обрабатывает запросы, давайте поговорим о том, что такое шаблонизатор. Постараюсь объяснить это так, чтобы было понятно даже ребенку.

Представьте, что вы хотите нарисовать картину. Но вместо того чтобы рисовать каждую картину с нуля, вы используете уже готовый набросок и просто заполняете его красками. Это намного быстрее и легче, чем рисовать все с начала каждый раз, не так ли?

Так вот, шаблонизатор делает то же самое, но для веб-страниц. Он берет набросок страницы (шаблон) и заполняет его данными. Это позволяет нам быстро и легко создавать динамические веб-страницы, которые могут меняться в зависимости от данных, которые мы получаем.

Например, представьте, что у вас есть страница со списком товаров в интернет-магазине. Вы не хотите создавать отдельную страницу для каждого товара, правда? Это было бы очень трудоемко и заняло бы много времени. Вместо этого вы создаете шаблон страницы и позволяете шаблонизатору заполнять его информацией о разных товарах.

Так что в основе своей шаблонизатор - это инструмент, который помогает нам создавать веб-страницы более эффективно. Он позволяет нам разделить структуру страницы (HTML) и данные, которые мы хотим отобразить на этой странице. Это делает наш код более чистым и упрощает процесс создания веб-страниц.

В следующем разделе мы узнаем больше о Thymeleaf, одном из самых популярных шаблонизаторов для Spring приложений, и увидим его в действии.

Создание Spring приложения с использованием шаблонизатора Thymeleaf

Теперь, когда мы поняли, что такое шаблонизатор, давайте поговорим о Thymeleaf и том, как его использовать в Spring приложении. Thymeleaf — это мощный шаблонизатор, который позволяет нам создавать динамические веб-страницы на стороне сервера.

Thymeleaf работает с HTML, и его особенность в том, что он может генерировать корректный и стандартный HTML5, что очень полезно для отладки и работы с дизайнерами, которые могут открывать и работать с шаблонами, не используя серверную часть.

Thymeleaf обрабатывает шаблоны и заменяет специальные Thymeleaf атрибуты и выражения на актуальные данные. Thymeleaf атрибуты обычно начинаются с префикса th:, например, th:text, th:if, th:each и так далее.

Начнем с того, что добавим Thymeleaf в наше приложение. Для этого нужно добавить следующую зависимость в файл build.gradle или pom.xml, в зависимости от того, какой системой сборки вы пользуетесь:

```
<!-- pom.xml -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

// build.gradle
implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
```

Теперь, когда Thymeleaf добавлен в наше приложение, давайте создадим шаблон. Шаблоны Thymeleaf обычно размещаются в директории src/main/resources/templates. Давайте создадим файл greeting.html:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Greeting</title>
</head>
<body>
    <h1 th:text="${message}">Hello, World!</h1>
</body>
</html>
```

Здесь мы используем атрибут th:text чтобы указать, что текст внутри тега <h1> должен быть заменен на значение переменной message.

Теперь давайте создадим контроллер, который будет использовать этот шаблон:

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class GreetingController {

    @GetMapping("/greeting")
    public String greeting(Model model) {
        model.addAttribute("message", "Привет, Thymeleaf!");
        return "greeting";
    }
}
```

В этом контроллере мы добавляем атрибут message в модель и возвращаем имя нашего шаблона greeting. Когда пользователь переходит по адресу /greeting, Thymeleaf берет шаблон greeting.html, заменяет th:text="${message}" на значение message из модели и возвращает получившуюся страницу.

Итак, мы создали Spring приложение с использованием шаблонизатора Thymeleaf. Thymeleaf дает нам большие возможности для создания динамических веб-страниц на стороне сервера, и это только начало.

Наши страницы выглядят немного скучно, не так ли? Ведь мы все любим красивые вещи, и веб-страницы не исключение. Здесь на помощь приходит CSS.

CSS (Cascading Style Sheets) - это язык стилей, который используется для описания внешнего вида документа, написанного на языке разметки, таком как HTML. Если представить HTML как скелет нашей веб-страницы, то CSS — это кожа и одежда, которые делают страницу красивой и удобной для глаз.

CSS позволяет нам контролировать такие вещи, как цвета, шрифты, отступы, выравнивание и многое другое. С помощью CSS мы можем создать совершенно уникальный дизайн для нашего веб-сайта и сделать его приятным для глаз.

Давайте добавим немного CSS в наше приложение. В Spring Boot CSS-файлы, как и другие статические файлы, обычно размещаются в директории src/main/resources/static. Создадим там новый файл styles.css:

```
body {
    background-color: lightblue;
}

h1 {
    color: navy;
    font-family: 'Arial', sans-serif;
}
```

Здесь мы делаем фон нашей страницы светло-синим и устанавливаем цвет и шрифт для заголовков первого уровня.

Чтобы применить этот CSS к нашей странице, нам нужно добавить ссылку на него в наш HTML-шаблон. Добавим в файл greeting.html следующий код внутрь тега <head>:

```
<head>
    <title>Greeting</title>
```

```
    <link rel="stylesheet"  href="/styles.css">
</head>
```

Тег <link> используется для подключения внешних ресурсов, в нашем случае - CSS файла.

Теперь, если вы откроете страницу /greeting, вы увидите, что она стала немного красивее благодаря нашему CSS.

Итак, мы узнали, как добавить CSS в наше Spring приложение и немного улучшили внешний вид нашей страницы. Попробуйте поэкспериментировать с разными стилями и увидите, как это может изменить ваши страницы!

Создание микросервиса

Теперь, когда мы изучили основы Spring MVC и Thymeleaf, представим, что мы - опытные Java-разработчики, и у нас есть новая задача - создать микросервис. Звучит профессионально, не так ли? Не беспокойтесь, это не так сложно, как кажется. Давайте сначала определим, что мы хотим сделать.

Техническое задание

Мы хотим создать простое Spring приложение, которое будет имитировать работу библиотеки. Оно должно уметь хранить книги, и пользователи смогут получить список всех книг, добавить новую книгу, получить информацию о конкретной книге по ее идентификатору и удалить книгу.

Вот что должен делать наш микросервис:

1. **GET /books**: Возвращает список всех книг. Каждая книга должна иметь уникальный идентификатор, название и автора.
2. **POST /books**: Принимает данные книги (название и автор) и добавляет ее в список книг. В ответе возвращает идентификатор новой книги.
3. **GET /books/{id}**: Возвращает информацию о книге с указанным идентификатором. Если книги с таким идентификатором нет, возвращает сообщение об ошибке.
4. **DELETE /books/{id}**: Удаляет книгу с указанным идентификатором. Если книги с таким идентификатором нет, возвращает сообщение об ошибке.

Мы не будем использовать базу данных, но мы можем имитировать ее работу с помощью коллекций в Java.

Начало работы: создание проекта

Первый шаг на нашем пути — это создание нового Spring проекта. Для этого мы будем использовать Spring Initializr, который делает этот процесс очень простым.

Перейдите на сайт start.spring.io, чтобы открыть Spring Initializr. Здесь мы увидим форму, в которой нужно заполнить некоторые детали нашего проекта.

- **Project**: выберите “Maven Project” или “Gradle Project”, в зависимости от вашей предпочитаемой системы сборки.
- **Language**: выберите “Java”. Это язык, на котором мы будем писать наше приложение.
- **Spring Boot**: выберите последнюю стабильную версию Spring Boot.
- **Project Metadata**: введите детали вашего проекта. Можете использовать “com.example” для Group и “library” для Artifact.
- **Packaging**: выберите “Jar”. Мы хотим создать исполняемый JAR файл для нашего приложения.
- **Java**: выберите версию Java, которую вы хотите использовать. Рекомендуется использовать последнюю стабильную версию.
- **Dependencies**: выберите зависимости для вашего проекта. Нам нужно добавить “Spring Web” для создания веб-приложения и “Thymeleaf” для использования шаблонизатора.

После заполнения всех данных, нажмите “Generate” для скачивания вашего нового проекта. Распакуйте скачанный ZIP-файл в удобное вам место.

Теперь у нас есть базовый каркас нашего приложения, и мы можем начать работу над его реализацией. Откройте проект в IntelliJ IDEA (или в любой другой IDE, которую вы предпочитаете).

Внутри проекта вы увидите структуру папок и файлов, которая уже знакома нам как разработчикам на Java. Главный класс нашего приложения находится в src/main/java/com/example/library/LibraryApplication.java. Это точка входа в наше приложение, и отсюда мы начнем нашу работу.

Создание моделей

Прежде чем начать писать код, давайте остановимся на минутку и подумаем: какие классы нам точно понадобятся для реализации нашего сервиса? Наш микросервис должен управлять книгами, так что нам определенно понадобится класс Book.

Каждая книга в нашей библиотеке должна иметь уникальный идентификатор, название и автора. Поэтому, нам нужно создать класс Book с этими полями. Давайте напишем его:

```
package com.example.library.model;

public class Book {

    private Long id;
    private String title;
    private String author;

    // конструкторы, геттеры, сеттеры, equals и hashCode
}
```

Теперь, когда у нас есть класс Book, давайте подумаем, что нам еще понадобится. Нам нужно хранить наши книги где-то. Для этого мы можем создать класс Library, который будет хранить список книг и предоставлять методы для добавления, получения и удаления книг. Это будет выглядеть примерно так:

```
package com.example.library.service;

import com.example.library.model.Book;

import java.util.ArrayList;
import java.util.List;

public class Library {

    private List<Book> books = new ArrayList<>();

    public List<Book> getAllBooks() {
        return books;
    }
```

```
    public Book getBookById(Long id) {
        // реализация
    }

    public void addBook(Book book) {
        // реализация
    }

    public void deleteBook(Long id) {
        // реализация
    }
}
```

Отлично, у нас есть классы Book и Library, и мы готовы перейти к следующему этапу.

Теперь, когда у нас есть основные классы, давайте добавим немного больше функциональности в нашу библиотеку. Первым делом, давайте обсудим, как мы будем идентифицировать наши книги. В нашем текущем классе Book у нас есть поле id типа Long. Но откуда мы будем брать эти идентификаторы?

Для этого мы можем использовать счетчик. Каждый раз, когда мы добавляем новую книгу, мы увеличиваем счетчик и присваиваем его значение в качестве идентификатора книги. Давайте добавим это в наш класс Library:

```
private Long idCounter = 1L;

public void addBook(Book book) {
    book.setId(idCounter++);
    books.add(book);
}
```

Теперь, каждый раз, когда мы добавляем новую книгу, она автоматически получает уникальный идентификатор.

Теперь давайте рассмотрим метод getBookById. В настоящее время он ничего не делает. Но нам нужно, чтобы он возвращал книгу с указанным идентификатором. Мы можем сделать это, проходя по списку всех книг и возвращая книгу, идентификатор которой совпадает с указанным. Если такой книги нет, мы можем возвращать null:

```
public Book getBookById(Long id) {
    for (Book book : books) {
```

```
            if (book.getId().equals(id)) {
                return book;
            }
        }
        return null;
    }
```

Наконец, давайте реализуем метод deleteBook. Он должен удалять книгу с указанным идентификатором из списка. Мы можем сделать это таким же образом, как и в методе getBookById, только вместо возвращения книги мы удаляем ее:

```
public void deleteBook(Long id) {
    books.removeIf(book -> book.getId().equals(id));
}
```

Итак, теперь у нас есть полностью функциональная библиотека, которая может хранить книги, возвращать все книги, находить книгу по идентификатору и удалять книги.

Сервсисный слой

Мы уже создали нашу модель Book и Library, которая будет действовать как репозиторий. Теперь нам нужно добавить сервисный слой, который будет служить своего рода “мостом” между нашим контроллером (который обрабатывает запросы) и нашим репозиторием (который управляет данными).

Сервисный слой - это место, где происходит большая часть бизнес-логики вашего приложения. Это позволяет нам изолировать нашу бизнес-логику от остальных слоев, что упрощает изменение и тестирование этой логики.

Для нашего приложения, нам потребуется создать класс LibraryService, который будет содержать методы для всех операций, которые мы хотим проводить с нашими книгами: получение всех книг, добавление новой книги, получение информации о книге по ее идентификатору и удаление книги.

Вот как это может выглядеть:

```
package com.example.library.service;

import com.example.library.model.Book;
import org.springframework.stereotype.Service;
```

```
import java.util.List;

@Service
public class LibraryService {

    private final Library library = new Library();

    public List<Book> getAllBooks() {
        return library.getAllBooks();
    }

    public Book getBookById(Long id) {
        return library.getBookById(id);
    }

    public Long addBook(Book book) {
        library.addBook(book);
        return book.getId();
    }

    public void deleteBook(Long id) {
        library.deleteBook(id);
    }
}
```

Мы помечаем наш класс LibraryService аннотацией @Service чтобы сообщить Spring, что этот класс является сервисом и его экземпляр должен быть создан во время запуска приложения.

В каждом из методов сервиса мы просто делегируем выполнение нашему репозиторию library, передавая входные параметры и возвращая результаты. Это обеспечивает разделение обязанностей и позволяет нам легко менять логику нашего сервиса независимо от контроллеров и репозиториев.

Создание контроллера

Теперь, когда у нас есть сервисный слой, который обрабатывает бизнес-логику нашего приложения, нам нужно создать контроллер. Контроллер в Spring MVC — это класс, который обрабатывает входящие HTTP-запросы и возвращает HTTP-ответы.

Создадим класс LibraryController, который будет использовать наш LibraryService для обработки запросов:

```
package com.example.library.controller;

import com.example.library.model.Book;
import com.example.library.service.LibraryService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/books")
public class LibraryController {

    private final LibraryService libraryService;

    public LibraryController(LibraryService libraryService) {
        this.libraryService = libraryService;
    }

    @GetMapping
    public ResponseEntity<List<Book>> getAllBooks() {
            return new ResponseEntity<>(libraryService.getAllBooks(),
HttpStatus.OK);
    }

    @PostMapping
    public ResponseEntity<Long> addBook(@RequestBody Book book) {
            return new ResponseEntity<>(libraryService.addBook(book),
HttpStatus.CREATED);
```

```
    }

    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable Long id) {
        Book book = libraryService.getBookById(id);
        if (book == null) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<>(book, HttpStatus.OK);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteBook(@PathVariable Long id) {
        libraryService.deleteBook(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }
}
```

Давайте разберем этот код по частям:

- @RestController говорит Spring, что этот класс является контроллером, и его методы должны обрабатывать HTTP-запросы.
- @RequestMapping("/books") указывает базовый путь, на который этот контроллер будет реагировать.
- Мы внедряем наш LibraryService через конструктор, чтобы контроллер мог использовать его для обработки запросов.
- @GetMapping, @PostMapping, @DeleteMapping аннотируют методы, которые должны обрабатывать GET, POST и DELETE запросы соответственно.
- @RequestBody и @PathVariable аннотации указывают, что параметр метода должен быть извлечен из тела запроса или из части пути запроса.
- Мы возвращаем ResponseEntity, чтобы мы могли указать HTTP-статус ответа и любые другие детали, которые нам могут понадобиться.

Теперь, когда у нас есть все необходимые компоненты нашего приложения, осталось сделать несколько последних шагов, чтобы все было готово к запуску.

Первое, что нужно сделать, - это убедиться, что наш сервис LibraryService подключен к нашему контроллеру. Для этого мы используем внедрение

зависимостей Spring, и конструктор LibraryController принимает LibraryService в качестве аргумента. Но мы также должны убедиться, что Spring знает, что он должен создать экземпляр LibraryService и использовать его при создании LibraryController.

Мы уже пометили класс LibraryService аннотацией @Service, чтобы Spring знал, что он должен создать экземпляр этого класса. Теперь нужно пометить LibraryController аннотацией @Controller, чтобы Spring знал, что этот класс тоже нуждается в управлении. На самом деле, мы уже использовали аннотацию @RestController, которая включает в себя @Controller, поэтому ничего дополнительно делать не нужно.

После того как все компоненты нашего приложения готовы и подключены, мы можем запустить приложение и проверить его работу.

Для запуска приложения перейдите в корневую директорию проекта и выполните команду ./mvnw spring-boot:run (для Maven) или ./gradlew bootRun (для Gradle), в зависимости от того, какую систему сборки вы выбрали при создании проекта.

И вот, наше приложение готово! Мы создали простой микросервис с использованием Spring MVC, который может управлять книгами в библиотеке.

Проверка работы нашего приложения

Теперь, когда наше приложение готово и работает, очень важно протестировать его, чтобы убедиться, что все работает, как ожидается.

Самый простой способ протестировать наше приложение - это отправить HTTP-запросы к его эндпоинтам и проверить, что они возвращают ожидаемые результаты. Мы можем сделать это с помощью любого клиента HTTP.

Один из наиболее популярных инструментов для этого - это Postman. Postman - это приложение, которое позволяет вам легко создавать и отправлять HTTP-запросы к любому эндпоинту и просматривать ответы. Давайте посмотрим, как мы можем использовать Postman для тестирования нашего приложения.

1. Сначала скачайте и установите Postman с официального сайта https://www.postman.com/downloads/.
2. Откройте Postman и создайте новый запрос, нажав на кнопку “+”.

3. Введите URL нашего эндпоинта. Если наше приложение работает локально на порту 8080, это будет что-то вроде http://localhost:8080/books.
4. Выберите тип запроса из выпадающего меню слева от URL. Мы можем начать с GET-запроса, чтобы получить список всех книг.
5. Нажмите “Send”, чтобы отправить запрос.
6. В нижней части окна вы увидите ответ от нашего приложения. Это должен быть список всех книг в нашей библиотеке (в настоящее время он должен быть пустым).

Таким образом, мы можем протестировать каждый из наших эндпоинтов, изменяя URL и тип запроса, а также добавляя тело запроса или параметры, если они необходимы.

Тестирование - это важный шаг в процессе разработки любого приложения. Оно помогает обнаружить и исправить ошибки, прежде чем они станут проблемой в продакшене.

Заключение

Дошли мы до конца нашего урока и, прежде чем попрощаться, давайте обсудим, почему изучение этих тем так важно.

Сегодня мы углубились в Spring MVC и изучили, как создавать веб-приложения с использованием этого мощного фреймворка. Мы изучили, как настроить контроллеры для обработки HTTP-запросов, использовали сервисный слой для обработки бизнес-логики и использовали Thymeleaf для рендеринга HTML-страниц. Кроме того, мы рассмотрели, как протестировать наше приложение, чтобы убедиться, что все работает как ожидалось.

Эти знания и навыки являются важной частью набора инструментов любого разработчика Java, работающего с веб-приложениями. В то время как принципы и практики, которые мы обсудили, могут применяться и за пределами Spring MVC, конкретные механизмы и технологии, которые мы использовали, делают нас более эффективными при работе с этим фреймворком.

Кроме того, понимание того, как создать и тестировать веб-приложение с нуля, может помочь нам лучше понять, как работают веб-приложения в целом, и какие

вопросы и проблемы мы можем встретить на пути. Это, в свою очередь, помогает нам стать лучшими разработчиками и более ценными членами наших команд.

Знание — это сила, и то, что мы с вами сегодня узнали, сделало нас немного сильнее.

В следующем уроке мы углубимся в Spring Data и научимся создавать репозитории для наших объектов. Это позволит нам сохранять наши объекты в базе данных и извлекать их оттуда, что сделает наше приложение еще более мощным и полезным.

Очень жду встречи с вами на следующем уроке, где мы разберемся с тем, как Spring Data может упростить нашу работу с базами данных.

Что можно почитать еще?

1. Изучение Spring Boot 2.0. Грег Тернквист
2. Освоение Spring Boot 2.0. Динеш Раджпут

Используемая литература

1. Spring Boot в действии. Крейг Уоллс
2. Spring Microservices в действии. Джон Карнелл
3. Cloud Native Java. Джош Лонг и Кенни Бастани