

Spring MVC. Использование шаблонизатора Thymeleaf









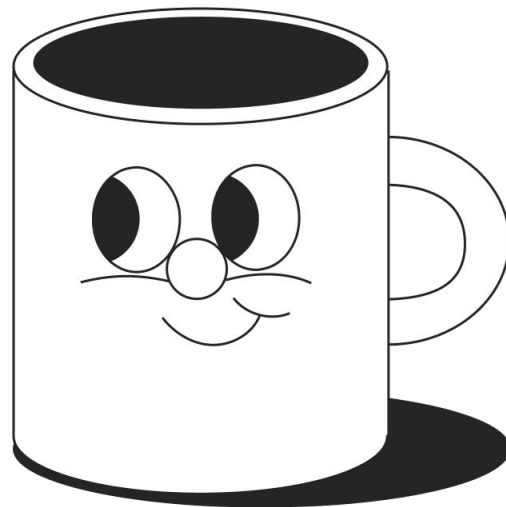
Евгений Манько

Java-разработчик, создатель данного курса

- ✨ Разрабатывал бэкенд для Яндекс, Тинькофф, МТС;
- ✨ Победитель грантового конкурса от «Росмолодежь»;
- ✨ Руководил IT-Департаментом «Студенты Москвы».

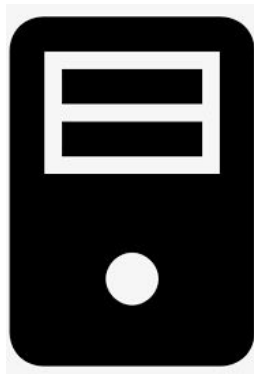
Что будет на уроке сегодня

-  Бэкенд и возврат готовых страниц
-  Разница между @Controller и @RestController
-  HTML
-  Шаблонизатор: художник веб-страниц
-  Создание Spring приложения с использованием шаблонизатора Thymleaf
-  Создание микросервиса





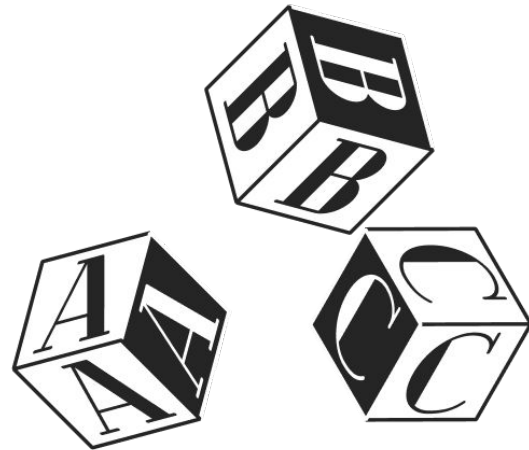
Клиент-Сервер





Разница между @Controller и @RestController

В Spring, аннотация @RestController используется для создания HTTP API, возвращая данные, которые клиент может обрабатывать, в то время как @Controller обрабатывает HTTP-запросы и возвращает веб-страницу с возможными динамическими данными. При использовании @Controller, наш код легко интегрируется с Thymeleaf и другими технологиями для создания динамических веб-страниц.





HTML

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.





HTML

Давайте начнем с тега `<title>`, который мы помещаем внутрь `<head>`. Вот как это выглядит:

```
1 <head>
2     <title>Мой первый дом</title>
3 </head>|
```

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

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



HTML

Для заголовков у нас есть теги `<h1>` до `<h6>`, где `<h1>` — это самый большой заголовок, а `<h6>` — самый маленький. Вот как это выглядит:

```
1 <body>
2   <h1>Добро пожаловать в мой дом!</h1>
3   <h2>Вот что внутри:</h2>
4   <h3>Гостиная</h3>
5   <p>Здесь у нас очень удобный диван и большой телевизор.</p>
6   <h3>Кухня</h3>
7   <p>Здесь мы готовим вкусные блюда.</p>
8 </body>|
```




HTML

Мы используем тег `` и атрибут `src` (который означает источник), чтобы указать, где находится наша картинка:

```
1 <body>
2     
3 </body>
```



HTML

После создания нового проекта мы добавим HTML-файл в директорию `src/main/resources/static`:

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <title>Мое первое Spring приложение</title>
5 </head>
6 <body>
7     <h1>Привет, мир!</h1>
8     <p>Добро пожаловать в мое первое Spring приложение!</p>
9 </body>
10 </html>|
```



HTML

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.

```
1 import org.springframework.stereotype.Controller;
2 import org.springframework.web.bind.annotation.RequestMapping;
3
4 @Controller
5 public class HomeController {
6
7     @RequestMapping("/")
8     public String home() {
9         return "index";
10    }
11 }
```



Как это работает в Spring

В Spring есть специальный компонент, который называется диспетчером (DispatcherServlet), и его работа — это как раз и есть определение, что нужно пользователю. Когда диспетчер понимает, что нужно пользователю, он ищет нужную страницу среди всех контроллеров приложения.

Давайте попробуем добавить на нашу страницу ссылку на другую страницу.





Как это работает в Spring

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

```
1 <body>
2     <h1 id="greeting">Привет, мир!</h1>
3     <p>Добро пожаловать в мое первое Spring приложение!</p>
4     <a href="/other-page">Посмотреть другую страницу</a>
5
6     <script>
7         document.getElementById('greeting').textContent = 'Привет, Spring!';
8     </script>
9 </body>
```



Шаблонизатор: художник веб-страниц

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.





Создание Spring приложения с использованием шаблонизатора Thymeleaf

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:

```
1 <!-- pom.xml -->
2 <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-thymeleaf</artifactId>
5 </dependency>
6 // build.gradle
7 implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'|
```



Создание Spring приложения с использованием шаблонизатора 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`:

```
1 <!DOCTYPE html>
2 <html xmlns:th="http://www.thymeleaf.org">
3 <head>
4     <title>Greeting</title>
5 </head>
6 <body>
7     <h1 th:text="${message}">Hello, World!</h1>
8 </body>
9 </html>|
```




Создание Spring приложения с использованием шаблонизатора Thymeleaf

Теперь давайте создадим контроллер, который будет использовать этот шаблон:

```
1 import org.springframework.stereotype.Controller;
2 import org.springframework.ui.Model;
3 import org.springframework.web.bind.annotation.GetMapping;
4
5 @Controller
6 public class GreetingController {
7
8     @GetMapping("/greeting")
9     public String greeting(Model model) {
10         model.addAttribute("message", "Привет, Thymeleaf!");
11         return "greeting";
12     }
13 }
```



Создание Spring приложения с использованием шаблонизатора Thymeleaf

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`:

```
1 body {  
2     background-color: lightblue;  
3 }  
4  
5 h1 {  
6     color: navy;  
7     font-family: 'Arial', sans-serif;  
8 }
```



Создание Spring приложения с использованием шаблонизатора Thymeleaf

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

```
1 <head>
2     <title>Greeting</title>
3     <link rel="stylesheet" href="/styles.css">
4 </head>|
```



Создание микросервиса

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.



Создание микросервиса

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.



Создание микросервиса

- **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.



Создание микросервиса

Создание моделей. Нам нужно создать класс Book с этими полями. Давайте напишем его:

```
1 package com.example.library.model;
2
3 public class Book {
4
5     private Long id;
6     private String title;
7     private String author;
8
9     // конструкторы, геттеры, сеттеры, equals и hashCode
10 }
```



Создание микросервиса

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:

```
1 package com.example.library.service;  
2  
3 import com.example.library.model.Book;  
4  
5 import java.util.ArrayList;  
6 import java.util.List;
```




Создание микросервиса

```
1 public class Library {
2
3     private List<Book> books = new ArrayList<>();
4
5     public List<Book> getAllBooks() {
6         return books;
7     }
8
9     public Book getBookById(Long id) {
10         // реализация
11     }
12
13     public void addBook(Book book) {
14         // реализация
15     }
16
17     public void deleteBook(Long id) {
18         // реализация
19     }
20 }
```



Создание микросервиса

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:

```
1 private Long idCounter = 1L;  
2  
3 public void addBook(Book book) {  
4     book.setId(idCounter++);  
5     books.add(book);  
6 }
```



Создание микросервиса

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`:

```
1 public Book getBookById(Long id) {  
2     for (Book book : books) {  
3         if (book.getId().equals(id)) {  
4             return book;  
5         }  
6     }  
7     return null;  
8 }
```



Создание микросервиса

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:

```
1 public void deleteBook(Long id) {  
2     books.removeIf(book → book.getId().equals(id));  
3 }
```



Создание микросервиса

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.





Создание микросервиса

Вот как это может выглядеть:

```
1 package com.example.library.service;  
2  
3 import com.example.library.model.Book;  
4 import org.springframework.stereotype.Service;  
5  
6 import java.util.List;  
7
```



Создание микросервиса

```
1 @Service
2 public class LibraryService {
3
4     private final Library library = new Library();
5
6     public List<Book> getAllBooks() {
7         return library.getAllBooks();
8     }
9
10    public Book getBookById(Long id) {
11        return library.getBookById(id);
12    }
13
14    public Long addBook(Book book) {
15        library.addBook(book);
16        return book.getId();
17    }
18
19    public void deleteBook(Long id) {
20        library.deleteBook(id);
21    }
22 }
```



Создание микросервиса

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:

```
1 package com.example.library.controller;
2
3 import com.example.library.model.Book;
4 import com.example.library.service.LibraryService;
5 import org.springframework.http.HttpStatus;
6 import org.springframework.http.ResponseEntity;
7 import org.springframework.web.bind.annotation.*;
8
9 import java.util.List;
```




Создание микросервиса

```
1 @RestController
2 @RequestMapping("/books")
3 public class LibraryController {
4
5     private final LibraryService libraryService;
6
7     public LibraryController(LibraryService libraryService) {
8         this.libraryService = libraryService;
9     }
10
11     @GetMapping
12     public ResponseEntity<List<Book>> getAllBooks() {
13         return new ResponseEntity<>(libraryService.getAllBooks(),
14         HttpStatus.OK);
15     }
```



Создание микросервиса

```
1 @PostMapping
2     public ResponseEntity<Long> addBook(@RequestBody Book book) {
3         return new ResponseEntity<>(libraryService.addBook(book),
4             HttpStatus.CREATED);
5     }
6
7     @GetMapping("/{id}")
8     public ResponseEntity<Book> getBookById(@PathVariable Long id) {
9         Book book = libraryService.getBookById(id);
10        if (book == null) {
11            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
12        }
13        return new ResponseEntity<>(book, HttpStatus.OK);
14    }
15
16    @DeleteMapping("/{id}")
17    public ResponseEntity<Void> deleteBook(@PathVariable Long id) {
18        libraryService.deleteBook(id);
19        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
20    }
```



Проверка работы нашего приложения

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.





Проверка работы 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).



Спасибо за внимание

