







Использование Spring для разработки серверного приложения

Урок 3



Что будет на уроке сегодня

-  Введение в клиент-серверное взаимодействие
-  Spring и клиент серверное взаимодействие
-  Практика по написанию серверного приложения
-  Проверка приложения
-  Запуск приложения в докере
-  Заключение



Клиент-Сервер

Клиентом является обычно браузер или приложение, который отправляет HTTP-запросы на сервер. Сервер — это машина или приложение, которое слушает эти запросы, обрабатывает их и отправляет обратно HTTP-ответы.





Протоколы

1

HTTP и HTTPS

HTTP, или гипертекстовый протокол передачи данных, — это основной протокол, используемый в вебе для передачи данных. HTTPS — это просто защищенная версия HTTP.

2

WebSocket

WebSocket — это другой протокол, который предназначен для обеспечения двусторонней связи между клиентом и сервером.



Протоколы

1

HTTP и HTTPS

HTTP, или гипертекстовый протокол передачи данных, — это основной протокол, используемый в вебе для передачи данных. HTTPS — это просто защищенная версия HTTP.

2

WebSocket

WebSocket — это другой протокол, который предназначен для обеспечения двусторонней связи между клиентом и сервером.

3

gRPC

Еще один протокол, который стоит упомянуть, — это gRPC от Google. Он поддерживает множество языков программирования и позволяет клиентам и серверам обмениваться данными как через единственный запрос/ответ, так и через потоковые вызовы.



Протоколы

REST, или представительное состояние передачи, — это архитектурный стиль, используемый для обмена данными в вебе.

Он базируется на принципах HTTP и поощряет разработчиков создавать простые, без состояний и предсказуемые веб-сервисы.

Вот некоторые причины, по которым мы выбираем REST:

- Простота
- Без состояния
- Предсказуемость



Как HTTP связан с REST?

REST — это стиль архитектуры, который использует HTTP для обмена данными. REST определяет набор “правил”, по которым клиенты и серверы должны обмениваться данными, и HTTP - это тот язык, на котором они общаются.





HTTP-методы

1

GET

Используется для запроса данных с сервера. Это как спросить: “Привет, сервер, можно я посмотрю эту веб-страницу?”

2

POST

Используется для отправки данных на сервер, чтобы создать новый ресурс. Это как сказать: “Привет, сервер, вот новые данные, добавь их, пожалуйста, в свою базу данных.”

3

PUT

Используется для обновления существующего ресурса на сервере. Это как сказать: “Привет, сервер, вот обновленные данные, пожалуйста, замени ими старые.”

4

DELETE

Используется для удаления ресурса на сервере. Это как сказать: “Привет, сервер, пожалуйста, удали эти данные.”



HTTP-коды ответов



200 OK

Запрос был успешно обработан. Это как “Ваша операция прошла успешно!”



201 Created

Запрос был успешно обработан, и был создан новый ресурс. Это как “Я создал новые данные, которые вы просили!”



400 Bad Request

Запрос неправильно сформирован и сервер не может его обработать. Это как “Я не понимаю, что вы просите.”



404 Not Found

Запрошенный ресурс не найден на сервере. Это как “Извините, я не могу найти данные, которые вы просили.”

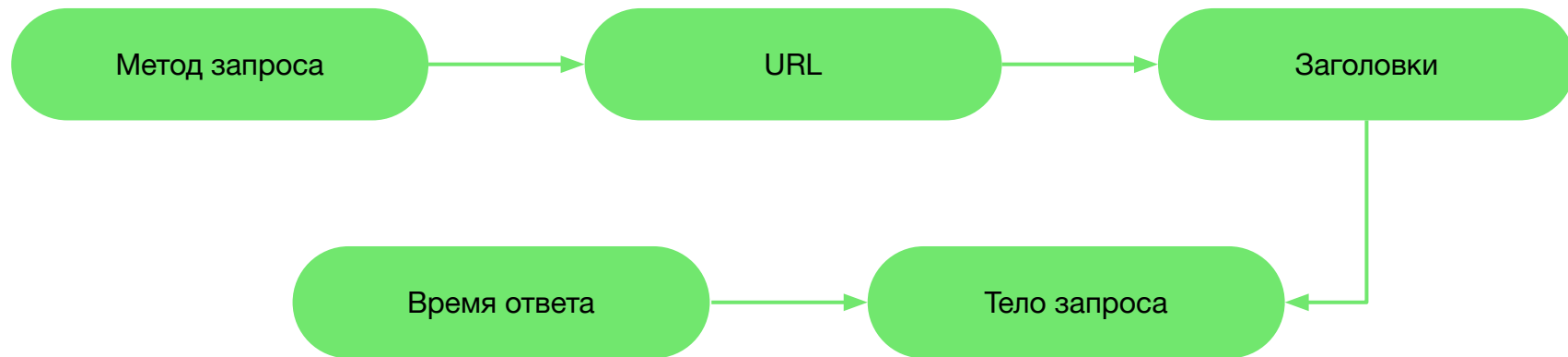


500 Internal Server Error

На сервере произошла ошибка при обработке запроса. Это как “У меня возникла проблема, и я не смог обработать ваш запрос.”



Структура запроса





Spring и клиент серверное взаимодействие

Spring - это один из самых популярных фреймворков для разработки бэкенд-приложений на Java. Он предлагает мощный и гибкий набор инструментов, которые упрощают разработку, обеспечивают безопасность и способствуют эффективной работе с данными.





Spring и клиент серверное взаимодействие

Вот почему Spring стал таким популярным:

1

Простота и гибкость

Spring значительно упрощает разработку серверных приложений.

2

Интеграция

Spring отлично интегрируется с множеством других технологий, включая базы данных, системы очередей сообщений, системы кеширования и др.

3

Сообщество и поддержка

Spring имеет большое и активное сообщество разработчиков, которые всегда готовы помочь в решении проблем и ответить на вопросы.



Контроллеры в Spring

Контроллер в Spring — это компонент, который обрабатывает входящие HTTP-запросы.

Внедрение зависимостей — это подход, при котором объект не создает или ищет свои зависимости самостоятельно, вместо этого они предоставляются ему извне.





Аннотация @Controller

Основой любого контроллера в Spring является аннотация `@Controller`. Она говорит Spring'у, что данный класс является контроллером, и должен быть использован для обработки входящих HTTP-запросов.

```
1 @Controller
2 public class MyController {
3     // код контроллера здесь
4 }
```



Аннотация @RequestMapping

Для указания, какой тип запроса и какой URL должен обрабатывать конкретный метод, используется аннотация `@RequestMapping`. Она может быть использована на уровне класса.

```
1 @Controller
2 @RequestMapping("/api")
3 public class MyController {
4     @RequestMapping(value = "/items", method = RequestMethod.GET)
5     public ResponseEntity<List<Item>> getItems() {
6         // код обработчика здесь
7     }
8 }
```



Принимаем параметры запроса

Чтобы принять параметры из URL, строки запроса или тела запроса, мы можем использовать аннотации `@PathVariable`, `@RequestParam` и `@RequestBody` соответственно.

```
1 @RequestMapping(value = "/items/{id}", method = RequestMethod.GET)
2 public ResponseEntity<Item> getItem(@PathVariable("id") Long id) {
3     // код обработчика здесь
4 }
```




Возвращаем ответ

Чтобы вернуть ответ из обработчика, мы просто возвращаем объект, который должен быть включен в тело ответа. Spring автоматически преобразует его в нужный формат (обычно JSON). Чтобы контролировать другие аспекты ответа, такие как HTTP-статус, мы можем использовать класс `ResponseEntity`.

```
1 @RequestMapping(value = "/items/{id}", method = RequestMethod.GET)
2 public ResponseEntity<Item> getItem(@PathVariable("id") Long id) {
3     Item item = getItemFromDatabase(id); // просто пример,
    реализация зависит от вашей конкретной ситуации
4     if (item == null) {
5         return new ResponseEntity<>(HttpStatus.NOT_FOUND);
6     } else {
7         return new ResponseEntity<>(item, HttpStatus.OK);
8     }
9 }
```



Работаем с параметрами запроса

Для того чтобы принять параметры из строки запроса, мы можем использовать аннотацию `@RequestParam`. Это может быть полезно, например, для реализации фильтрации или пагинации.

```
1 @RequestMapping(value = "/items", method = RequestMethod.GET)
2 public ResponseEntity<List<Item>> getItems(
3     @RequestParam(value = "page", defaultValue = "1") int page,
4     @RequestParam(value = "size", defaultValue = "10") int size) {
5     List<Item> items = getItemsFromDatabase(page, size); // это
    просто пример
6     return new ResponseEntity<>(items, HttpStatus.OK);
7 }
```



Обрабатываем исключения

Ошибки и исключения - это неизбежная часть любого приложения. В Spring мы можем использовать аннотацию `@ExceptionHandler` для определения методов, которые будут обрабатывать определенные исключения.

```
1 @Controller
2 @RequestMapping("/api")
3 public class MyController {
4
5     // другие обработчики здесь
6
7     @ExceptionHandler(ItemNotFoundException.class)
8     public ResponseEntity<String> handleItemNotFoundException(ItemNotFoundException ex) {
9         return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
10    }
11 }
```



Обрабатываем JSON

Большинство RESTful API используют JSON для обмена данными. Spring автоматически преобразует объекты в JSON и наоборот. Все, что нам нужно сделать, это использовать аннотацию `@RequestBody` для принятия JSON в качестве входных данных и возвращать объекты из наших обработчиков.

```
1 @RequestMapping(value = "/items", method = RequestMethod.POST)
2 public ResponseEntity<Item> createItem(@RequestBody Item newItem) {
3     Item createdItem = saveItemToDatabase(newItem); // это просто пример
4     return new ResponseEntity<>(createdItem, HttpStatus.CREATED);
5 }
```



Практика по написанию серверного приложения

Мы собираемся создать простое приложение для управления списком задач.
Для начала нам понадобятся следующие компоненты:

1. Модель Task для представления задач.
2. Сервис TaskService для управления задачами.
3. Контроллер TaskController для обработки HTTP-запросов.





Практика по написанию серверного приложения

1. Создание модели. Начнем с создания класса Task, который будет представлять нашу задачу.

```
1 public class Task {  
2  
3     private Long id;  
4     private String name;  
5     private String description;  
6     private boolean completed;  
7  
8     // конструкторы, геттеры и сеттеры  
9 }
```



Практика по написанию серверного приложения

2. Создание сервиса. Следующим шагом будет создание сервиса, который будет управлять нашими задачами. Мы используем простой список для хранения задач.

```
1 public class Task {  
2  
3     private Long id;  
4     private String name;  
5     private String description;  
6     private boolean completed;  
7  
8     // конструкторы, геттеры и сеттеры  
9 }
```



Практика по написанию серверного приложения

```
1 public Task createTask(Task task) {
2     tasks.add(task);
3     return task;
4 }
5
6 public Task updateTask(Long id, Task updatedTask) {
7     Task existingTask = getTask(id);
8     if (existingTask != null) {
9
10        existingTask.setName(updatedTask.getName());
11
12        existingTask.setDescription(updatedTask.getDescription());
13
14        existingTask.setCompleted(updatedTask.isCompleted());
15    }
16    return existingTask;
17 }
18
19 public void deleteTask(Long id) {
20     tasks.removeIf(task →
21         task.getId().equals(id));
22 }
23 }
```




Практика по написанию серверного приложения

3. Создание контроллера. Теперь давайте создадим контроллер, который будет обрабатывать HTTP-запросы и использовать наш сервис для выполнения операций над задачами.

```
1 @Controller
2 @RequestMapping("/api/tasks")
3 public class TaskController {
4
5     private final TaskService taskService;
6
7     // внедрение зависимостей через конструктор
8     public TaskController(TaskService taskService) {
9         this.taskService = taskService;
10    }
11
12    @RequestMapping(method = RequestMethod.GET)
13    public ResponseEntity<List<Task>> getAllTasks() {
14        return new ResponseEntity<
15        (taskService.getAllTasks(), HttpStatus.OK);
16    }
```



Практика по написанию серверного приложения

```
1 @RequestMapping(value =("/{id}", method = RequestMethod.PUT)
2     public ResponseEntity<Task> updateTask(@PathVariable("id") Long id, @RequestBody Task updatedTask) {
3         Task task = taskService.updateTask(id, updatedTask);
4         if (task != null) {
5             return new ResponseEntity<>(task, HttpStatus.OK);
6         } else {
7             return new ResponseEntity<>(HttpStatus.NOT_FOUND);
8         }
9     }
10
11 @RequestMapping(value =("/{id}", method = RequestMethod.DELETE)
12     public ResponseEntity<Void> deleteTask(@PathVariable("id") Long id) {
13         taskService.deleteTask(id);
14         return new ResponseEntity<>(HttpStatus.NO_CONTENT);
15     }
16 }
```



Зачем тестировать свое приложение?



1. Понимание функциональности: Ручное тестирование позволяет разработчикам лучше понять, как работает их приложение с точки зрения конечного пользователя.



2. Обнаружение ошибок: Несмотря на все преимущества автоматизированного тестирования, оно может не всегда обнаруживать все возможные проблемы.



3. Своевременная обратная связь: Ручное тестирование может быть выполнено в любой точке процесса разработки для получения немедленной обратной связи.



4. Валидация требований: Ручное тестирование позволяет убедиться, что приложение соответствует требованиям бизнеса и пользователя.



Swagger

Для начала, чтобы использовать Swagger, нам нужно добавить зависимость в наш проект.

Если вы используете Maven, добавьте следующую зависимость в ваш pom.xml:

```
1 <dependency>
2     <groupId>io.springfox</groupId>
3     <artifactId>springfox-swagger2</artifactId>
4     <version>2.9.2</version>
5 </dependency>
```



Конфигурация Swagger

Затем нам нужно настроить Swagger в нашем приложении. Мы можем сделать это, создав конфигурационный класс с аннотацией `@EnableSwagger2`.

```
1 @Configuration
2 @EnableSwagger2
3 public class SwaggerConfig {
4     @Bean
5     public Docket api() {
6         return new Docket(DocumentationType.SWAGGER_2)
7             .select()
8             .apis(RequestHandlerSelectors.any())
9             .paths(PathSelectors.any())
10            .build();
11    }
12 }
```



Проверка работы приложения в Swagger UI

<http://localhost:8080/swagger-ui.html>





Postman

Postman — это популярный инструмент для тестирования API, который позволяет отправлять HTTP-запросы к вашему серверу и просматривать ответы. Он имеет простой и интуитивно понятный интерфейс, что делает его удобным в использовании даже для новичков.





Скачивание и установка Postman

Перед тем как мы начнем, вам нужно скачать и установить Postman. Вы можете сделать это, перейдя на [официальный сайт Postman](#) и следуя инструкциям по установке.





Отправка запросов в Postman

Как только Postman установлен, вы можете запустить его и начать отправлять запросы к вашему API.

1. Создайте новый запрос, нажав на кнопку “+”.
2. Выберите тип запроса из выпадающего меню слева от адресной строки (например, GET, POST, PUT, DELETE).
3. Введите URL вашего API в адресной строке. Например, если вы хотите проверить метод `getAllTasks`, ваш URL будет выглядеть примерно так: `http://localhost:8080/api/tasks` (замените `localhost:8080` на адрес и порт вашего приложения, если они отличаются).
4. Если ваш метод требует параметров запроса или тела запроса, вы можете добавить их на соответствующих вкладках под адресной строкой.
5. Нажмите на кнопку “Send” справа от адресной строки, чтобы отправить запрос.
6. После отправки запроса вы увидите ответ от вашего сервера в нижней части окна.



Зачем тестировать свое приложение?



Curl: curl — это мощный инструмент командной строки, который позволяет отправлять HTTP-запросы к серверу.



GET-запросы: Для отправки GET-запроса в curl достаточно просто указать URL. Например, если вы хотите получить список всех задач, вы можете ввести следующую команду: curl http://localhost:8080/api/tasks



POST-запросы: Для отправки POST-запросов и передачи данных в теле запроса в curl вы можете использовать флаг -d (или --data).

```
1 curl -X POST -H "Content-Type:
  application/json" -d '{"name": "New
  Task", "description": "New task
  description", "completed": false}'
  http://localhost:8080/api/tasks
```



Зачем тестировать свое приложение?



PUT-запросы: PUT-запросы аналогичны POST-запросам, но они обычно используются для обновления существующих ресурсов.

```
1 curl -X PUT -H "Content-Type:
  application/json" -d '{"name":"Updated
  Task","description":"Updated task
  description","completed":true}'
  http://localhost:8080/api/tasks/1
```



DELETE-запросы: Наконец, для отправки DELETE-запроса в curl вы просто используете флаг -X DELETE (или --request DELETE). Например, для удаления задачи с идентификатором 1 вы можете ввести следующую команду: `curl -X DELETE http://localhost:8080/api/tasks/1`



DOCKER

Docker - это открытая платформа для разработки, доставки и запуска приложений. Docker позволяет “упаковать” приложение вместе со всем его окружением в контейнер, который можно легко перенести на любую машину, которая поддерживает Docker.





DOCKER

Использование Docker имеет ряд преимуществ для разработчиков:

1. **Постоянство:** Docker гарантирует, что ваше приложение будет работать одинаково в любой среде.
2. **Изоляция:** каждый Docker-контейнер работает независимо, поэтому вы можете запускать разные приложения с разными требованиями на одной машине.
3. **Быстрое развертывание:** с Docker вы можете быстро и легко развернуть и масштабировать свое приложение.
4. **Управление зависимостями:** вместо того, чтобы устанавливать все зависимости на вашей машине, вы можете упаковать их вместе с приложением в Docker-контейнер.



Запускаем наше приложение в Docker

Создание Dockerfile.

Dockerfile — это текстовый файл, который описывает, как создать Docker-образ. Вот простой Dockerfile для нашего приложения:

```
FROM openjdk:11
COPY ./target/my-app-1.0.0.jar /usr/src/my-app/my-app-1.0.0.jar
WORKDIR /usr/src/my-app
EXPOSE 8080
CMD ["java", "-jar", "my-app-1.0.0.jar"]
```



Запускаем наше приложение в Docker

Вот что делает каждая строка:

- FROM openjdk:11 говорит Docker использовать официальный образ Java 11 в качестве базового.
- COPY ./target/my-app-1.0.0.jar /usr/src/my-app/my-app-1.0.0.jar копирует JAR-файл нашего приложения в контейнер.
- WORKDIR /usr/src/my-app устанавливает рабочий каталог внутри контейнера.
- EXPOSE 8080 говорит Docker, что наше приложение будет слушать на порту 8080.
- CMD ["java", "-jar", "my-app-1.0.0.jar"] запускает наше приложение внутри контейнера.



Запускаем наше приложение в Docker

После того как у вас есть Dockerfile, вы можете собрать Docker-образ и запустить его в контейнере. Сначала убедитесь, что у вас собран JAR-файл вашего приложения (например, с помощью `mvn package`), а затем выполните следующие команды:

```
docker build -t my-app .  
docker run -p 8080:8080 my-app
```

Первая команда собирает Docker-образ с именем `my-app` из текущего каталога (где находится Dockerfile). Вторая команда запускает контейнер из образа `my-app`, пробрасывая порт 8080 из контейнера на порт 8080 хост-машины.



Спасибо за внимание

