Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»
(ВлГУ)**

УТВЕРЖДАЮ
Заведующий кафедрой ИСПИ
_____ И.Е. Жигалов
«20» марта 2025 г.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ МЕЖДИСЦИПЛИНАРНОГО КУРСА

«РАЗРАБОТКА ИНФОРМАЦИОННЫХ РЕСУРСОВ С ИСПОЛЬЗОВАНИЕМ ПРОГРАММНЫХ ПЛАТФОРМ»

В РАМКАХ ПРОФЕССИОНАЛЬНОГО МОДУЛЯ

«РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЙ НА СТОРОНЕ СЕРВЕРА»

09.02.09 Веб-разработка
Разработчик веб приложений

Владимир, 2025

Методические указания к лабораторным работам междисциплинарного курса «Разработка информационных ресурсов с использованием программных платформ» разработал преподаватель КИТП Гуськов Н.О.

Методические указания к лабораторным работам рассмотрены и одобрены на заседании УМК специальности 09.02.09 Веб-разработка
протокол № 1 от «10» марта 2025 г.

Председатель УМК специальности _____ И.Е. Жигалов

Методические указания к лабораторным работам рассмотрены и одобрены на заседании кафедры ИСПИ
протокол № 7а от «12» марта 2025 г.

Рецензент от работодателя:
руководитель группы обеспечения
качества программного обеспечения
ООО «БСЦ МСК» _____ С.С. Смирнова

Лабораторная работа № 1

Тема: Запуск первого Spring Boot приложения

Цель работы: Ознакомиться с основами создания Java веб-приложения с использованием платформы Spring Boot. Получить опыт запуска первого приложения, настройки зависимостей и начальной конфигурации проекта.

Порядок выполнения работы

1. Установите необходимые инструменты:

- Убедитесь, что установлены JDK (рекомендуется версия 11 или выше) и IDE (например, IntelliJ IDEA или Eclipse).

- Установите Maven для управления зависимостями.

2. Создайте новый проект Spring Boot:

- Используйте Spring Initializr для создания проекта. Выберите зависимости Spring Web и DevTools.

- Настройте параметры проекта: укажите название группы, артефакта и базовую структуру проекта.

3. Изучите структуру проекта:

- Ознакомьтесь с файлами и папками, созданными Spring Initializr. Обратите внимание на структуру папок src/main/java, src/main/resources и на файл application.properties.

- Проверьте файл pom.xml (если используется Maven) на наличие всех зависимостей.

4. Создайте первый контроллер:

- Создайте Java-класс с аннотацией @RestController.

- Добавьте метод с аннотацией @GetMapping, который будет возвращать строку, например "Hello, Spring Boot!".

5. Запустите приложение:

- Используйте команду mvn spring-boot:run или запустите проект через IDE.

- Убедитесь, что приложение успешно запустилось и доступно на порту 8080 по умолчанию.

- Перейдите по адресу http://localhost:8080/ или http://localhost:8080/{ваш-метод} и проверьте, что контроллер возвращает ожидаемый результат.

6. Изучите логи и завершите приложение:

- Обратите внимание на логи в консоли во время запуска. Изучите основные этапы инициализации Spring Boot.

- Остановите приложение через IDE или терминал.

Варианты заданий

1. Добавьте еще один метод в контроллере, который возвращает JSON-ответ (например, объект с полями message и `status`).

2. Измените порт, на котором работает приложение, через настройки в application.properties.

3. Добавьте зависимость Spring Boot Actuator и изучите информацию о состоянии приложения.

4. Настройте отображение строки приветствия (например, “Welcome to Spring Boot!”) из файла application.properties.

5. Создайте отдельный контроллер, который возвращает текущую дату и время.

Контрольные вопросы

1. Какие компоненты автоматически создаются при создании проекта через Spring Initializr?

2. В чем различие между аннотациями @RestController и @Controller?

3. Как запустить Spring Boot приложение через командную строку?

4. Как изменить конфигурацию порта для Spring Boot приложения?

5. Какие возможности предоставляет Spring Boot DevTools?

Лабораторная работа № 2

Тема: Основные концепции Spring Core

Цель работы: Изучить основные концепции Spring Core, такие как внедрение зависимостей (Dependency Injection) и управление компонентами в контейнере Spring. Понять, как Spring упрощает создание и тестирование Java приложений за счет инверсии управления (IoC).

Порядок выполнения работы

1. Создайте новый Spring Boot проект:

- Используйте Spring Initializr для создания нового проекта с зависимостью Spring Core (укажите только минимальные зависимости, без Spring Web).

2. Изучите основы контейнера IoC и внедрения зависимостей:

- Создайте несколько классов, которые представляют сущности вашего приложения (например, Person и `Address`).

- Определите классы-сервисы и аннотируйте их с помощью @Service, @Repository, @Component, чтобы Spring смог управлять их экземплярами.

3. Настройте внедрение зависимостей с помощью аннотаций:

- Внедрите зависимости между классами, используя аннотацию @Autowired в полях, методах или конструкторах.

- Создайте конфигурационный класс и отметьте его аннотацией @Configuration. Определите несколько компонентов с помощью метода @Bean для явного создания экземпляров объектов.

4. Работа с конфигурацией:

- Ознакомьтесь с файлом application.properties. Настройте несколько свойств и добавьте доступ к ним через аннотацию @Value в классе-сервисе.

5. Создайте тест для проверки внедрения зависимостей:

- Напишите тестовый класс с использованием аннотации @SpringBootTest.

- Проверьте, что компоненты создаются и внедряются корректно.

6. Запустите и протестируйте проект:

- Запустите приложение, убедитесь, что зависимости внедряются и компоненты взаимодействуют как задумано.

Варианты заданий

1. Создайте компонент, который представляет собой службу уведомлений, и внедрите его в класс сервиса.

2. Настройте внедрение зависимости по имени и протестируйте его работу.

3. Создайте несколько классов, реализующих один интерфейс, и настройте выбор конкретной реализации через @Qualifier.

4. Используйте аннотацию @Scope для создания экземпляра компонента с другим жизненным циклом (например, `prototype`).

5. Реализуйте простой конфигуратор приложения с использованием профилей Spring (`@Profile`), который будет активировать разные компоненты в зависимости от среды.

Контрольные вопросы

1. Что такое контейнер IoC в Spring, и как он работает?

2. В чем различие между аннотациями @Service, @Repository и @Component?

3. Какие способы внедрения зависимостей поддерживает Spring?

4. Для чего используется аннотация @Qualifier, и когда она необходима?

5. Что такое @Scope, и какие виды областей видимости поддерживаются Spring?

Лабораторная работа № 3

Тема: Spring Boot и Docker

Цель работы: Ознакомиться с основами контейнеризации Spring Boot приложений с использованием Docker. Научиться создавать Docker-образы и запускать контейнеры для упрощения развертывания и управления приложениями.

Порядок выполнения работы

1. Установите Docker:

- Убедитесь, что Docker установлен на вашем компьютере. При необходимости скачайте и установите Docker Desktop (для Windows или macOS).

- Проверьте установку, запустив команду docker --version в терминале.

2. Создайте простое Spring Boot приложение:

- Создайте новый проект Spring Boot с минимальными зависимостями (например, Spring Web).

- Напишите простой контроллер, который возвращает сообщение, например, "Hello from Docker!".

- Убедитесь, что проект запускается и работает локально.

3. Создайте Dockerfile для приложения:

- В корне проекта создайте файл Dockerfile и пропишите команды для создания Docker-образа:

```
# Указываем базовый образ
FROM openjdk:11-jdk-slim
```

```
# Указываем рабочую директорию
WORKDIR /app

# Копируем JAR-файл в контейнер
COPY target/your-application.jar app.jar

# Указываем команду запуска
ENTRYPOINT ["java", "-jar", "app.jar"]
```

- Замените your-application.jar на имя JAR-файла вашего проекта.

4. Соберите и запустите приложение в Docker:

- Сначала соберите проект в JAR-файл с помощью команды mvn clean package.

- Затем создайте Docker-образ, выполнив команду:

```
docker build -t spring-boot-docker-app .
```

- Запустите контейнер с вашим приложением:

```
docker run -p 8080:8080 spring-boot-docker-app
```

- Перейдите по адресу http://localhost:8080, чтобы убедиться, что приложение работает в контейнере Docker.

5. Оптимизация Dockerfile (опционально):

- Изучите возможность создания меньшего по размеру образа, используя многослойный подход (например, создайте временный контейнер для сборки и другой для выполнения).

6. Остановите и удалите контейнеры и образы:

- Практикуйте остановку контейнера командой docker stop, удаление контейнера docker rm, а также удаление образа docker rmi.

Варианты заданий

1. Настройте приложение так, чтобы порт, на котором оно работает, можно было задавать через переменные окружения.

2. Изучите и реализуйте многослойный Dockerfile для оптимизации размера образа.

3. Создайте docker-compose.yml файл для запуска нескольких контейнеров (например, добавьте контейнер базы данных).

4. Настройте Dockerfile для автоматического перезапуска приложения при изменении кода (например, с использованием Spring Boot DevTools).

5. Добавьте метаданные в Dockerfile (например, информацию о версии и авторе образа).

Контрольные вопросы

1. Для чего используется Docker, и какие преимущества он предоставляет при развертывании приложений?

2. Что такое Dockerfile, и какие основные команды используются для его написания?

3. Как создать и запустить Docker-образ для Spring Boot приложения?

4. В чем различие между командами COPY и ADD в Dockerfile?

5. Какие команды используются для управления контейнерами и образами в Docker?

Лабораторная работа № 4

Тема: Spring Boot Initializer

Цель работы: Ознакомиться с возможностями Spring Boot Initializr для быстрого создания проектов на платформе Spring Boot. Изучить параметры настройки проекта, включая выбор зависимостей и создание структуры проекта для различных типов приложений.

Порядок выполнения работы

1. Изучите интерфейс Spring Boot Initializr:

- Перейдите на сайт https://start.spring.io.

- Ознакомьтесь с возможностями: выбор языка (Java), версии Spring Boot, типа сборки (Maven или Gradle), параметров проекта (Group, Artifact, Name), а также доступных зависимостей.

2. Создайте проект с помощью Spring Boot Initializr:

- Задайте параметры для вашего проекта: выберите язык Java, Spring Boot версии 2.x или выше, тип сборки (Maven или Gradle).

- Укажите группу и артефакт для проекта, например, com.example и demo-app.

- Добавьте минимальные зависимости, такие как Spring Web, чтобы создать базовое веб-приложение.

3. Изучите структуру созданного проекта:

- Откройте проект в вашей IDE (например, IntelliJ IDEA).

- Ознакомьтесь с файлами и директориями, созданными Initializr:

- src/main/java: содержит основной код приложения.

- src/main/resources: конфигурационные файлы, такие как application.properties.

- pom.xml (для Maven) или build.gradle (для Gradle): файл конфигурации зависимостей.

4. Изучите и настройте параметры профилей:

- Ознакомьтесь с возможностью использования профилей в Spring. Создайте файл application-dev.properties для конфигурации профиля разработки.

- Примените настройки для конкретного профиля через параметр spring.profiles.active в application.properties.

5. Настройка логирования:

- Откройте файл application.properties и добавьте настройки логирования, например:

logging.level.org.springframework.web=DEBUG

logging.level.com.example=INFO

- Запустите приложение и проверьте вывод логов, чтобы убедиться, что конфигурация логирования работает корректно.

6. Запуск проекта:

- Запустите приложение с помощью команды mvn spring-boot:run или через вашу IDE.

- Убедитесь, что приложение запускается корректно и доступно по адресу http://localhost:8080.

Варианты заданий

1. Создайте проект с использованием Spring Boot Actuator для мониторинга состояния приложения и получения метрик.

2. Настройте параметры для различных сред (например, профили разработки и продакшн) с помощью файла application.properties и профилей Spring.

3. Добавьте поддержку логирования с помощью настройки в файле application.properties и проверкой логов в приложении.

4. Создайте проект с использованием Spring DevTools для горячей перезагрузки при изменении кода и улучшения процесса разработки.

Контрольные вопросы

1. Что такое Spring Boot Initializr и как его использовать для создания проектов?

2. Как выбрать зависимости для проекта с помощью Spring Boot Initializr?

3. Какие параметры можно настроить в файле application.properties для работы с профилями Spring?

4. Как настроить логирование в Spring Boot через файл application.properties?

5. Каковы основные различия между использованием Maven и Gradle для сборки Spring Boot приложений?

Лабораторная работа № 5

Тема: Spring Boot H2 Database

Цель работы: Ознакомиться с использованием встроенной базы данных H2 в проекте Spring Boot. Научиться конфигурировать и интегрировать H2 для хранения данных в приложении, а также работать с JPA для выполнения операций с базой данных.

Порядок выполнения работы

1. Создание проекта с использованием Spring Boot Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект.

- Выберите зависимости: Spring Web, Spring Data JPA, H2 Database.

- Скачайте и откройте проект в вашей IDE.

2. Настройка базы данных H2:

- Откройте файл application.properties и добавьте настройки для базы данных H2:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```

- Эти настройки подключат H2 как базу данных в памяти и включат консоль для просмотра данных через веб-браузер по адресу http://localhost:8080/h2-console.

3. Создание сущностей и репозиториев:

- Создайте сущность, которая будет использоваться для хранения данных. Например, сущность Product:

```
@Entity
public class Product {
   @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private Double price;
    // Геттеры и сеттеры
}
```

- Создайте интерфейс репозитория для работы с сущностью:

```
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

4. Работа с данными через REST API:

- Создайте REST-контроллер для управления данными продуктов:

```
@RestController
@RequestMapping("/products")
public class ProductController {
    @Autowired
    private ProductRepository productRepository;
    @GetMapping
    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }
    @PostMapping
    public Product createProduct(@RequestBody Product product) {
        return productRepository.save(product);
    }
}
```

- Теперь можно добавлять новые продукты через HTTP-запросы и получать список всех продуктов.

5. Запуск и тестирование приложения:

- Запустите приложение, используя команду mvn spring-boot:run или

через вашу IDE.

- Перейдите по адресу http://localhost:8080/products для получения списка продуктов.

- Используйте инструменты, такие как Postman или cURL, для отправки POST-запросов на добавление новых продуктов.

6. Просмотр данных в консоли H2:

- Перейдите по адресу http://localhost:8080/h2-console в браузере.

- Введите настройки подключения (URL: jdbc:h2:mem:testdb, Username: sa, Password: `password`).

- В консоли можно выполнять SQL-запросы для работы с данными в базе.

Варианты заданий

1. Создайте сущность и репозиторий для работы с пользователями (например, User с полями id, name, `email`).

2. Добавьте в приложение CRUD-операции (создание, чтение, обновление, удаление) для сущности Product.

3. Настройте автоматическое создание и удаление таблиц при запуске приложения с помощью JPA.

4. Добавьте поддержку валидации данных (например, с помощью аннотаций @NotNull, @Size, и т.д.) для полей сущности Product.

Контрольные вопросы

1. Что такое база данных H2 и как её настроить для использования с Spring Boot?

2. Как работает Spring Data JPA и как создать репозиторий для сущности?

3. Как выполнить операции CRUD с использованием Spring Data JPA?

4. Как включить и использовать консоль H2 для работы с данными в базе?

5. Что такое автоматическое создание таблиц в JPA и как его настроить?

Лабораторная работа № 6

Тема: Spring Boot JDBC

Цель работы: Ознакомиться с использованием JDBC в Spring Boot для работы с реляционными базами данных. Научиться конфигурировать подключение к базе данных и выполнять базовые SQL-запросы с использованием JdbcTemplate.

Порядок выполнения работы

1. Создание проекта с использованием Spring Boot Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект.

- Выберите зависимости: Spring Web, Spring JDBC, H2 Database (или другую СУБД, с которой вы будете работать).

- Скачайте и откройте проект в вашей IDE.

2. Настройка базы данных и JDBC:

- Откройте файл application.properties и добавьте настройки для базы данных. Пример для H2:

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```

- Если используется другая база данных (например, MySQL), замените параметры подключения на соответствующие.

3. Настройка JdbcTemplate:

- Spring Boot автоматически настраивает JdbcTemplate, если добавлена зависимость Spring JDBC. Однако, для явной настройки можно использовать следующее:

```
@Configuration
@EnableTransactionManagement
```

```
public class DatabaseConfig {
    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

- В этом примере DataSource автоматически настраивается из настроек в application.properties.

4. Создание модели и репозитория:

- Создайте модель для работы с данными. Например, сущность Product:

```
public class Product {
    private Long id;
    private String name;
    private Double price;
    // Геттеры и сеттеры
}
```

- Создайте сервис или репозиторий для работы с базой данных через JdbcTemplate. Пример простого репозитория:

```
@Service
public class ProductService {
    @Autowired
    private JdbcTemplate jdbcTemplate;
    public List<Product> getAllProducts() {
        String sql = "SELECT * FROM products";
        return jdbcTemplate.query(sql, new BeanPropertyRowMapper<>(Product.class));
    }
    public int saveProduct(Product product) {
```

```
        String sql = "INSERT INTO products (name, price) VALUES (?, ?)";
        return jdbcTemplate.update(sql, product.getName(), product.getPrice());
    }
}
```

5. Создание таблицы в базе данных:

- Если вы используете H2, создайте таблицу products в базе данных с помощью следующего SQL-запроса:

```
CREATE TABLE products (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255),
    price DECIMAL(10, 2)
);
```

- Это можно выполнить через консоль H2 или через JdbcTemplate в приложении.

6. Реализация REST API для работы с продуктами:

- Создайте контроллер для работы с сущностью Product:

```
@RestController
@RequestMapping("/products")
public class ProductController {
    @Autowired
    private ProductService productService;
    @GetMapping
    public List<Product> getAllProducts() {
        return productService.getAllProducts();
    }
    @PostMapping
    public void createProduct(@RequestBody Product product) {
        productService.saveProduct(product);
```

```
    }
}
```

7. Запуск и тестирование приложения:

- Запустите приложение, используя команду mvn spring-boot:run или через вашу IDE.

- Перейдите по адресу http://localhost:8080/products для получения списка продуктов.

- Используйте инструменты, такие как Postman или cURL, для отправки POST-запросов на добавление новых продуктов.

Варианты заданий

1. Использование различных типов SQL-запросов: Создайте приложение для работы с более сложными SQL-запросами, например, обновление и удаление данных.

2. Добавьте пагинацию: Настройте пагинацию для получения списка продуктов с помощью JdbcTemplate.

3. Использование PreparedStatement: Вместо использования BeanPropertyRowMapper, используйте PreparedStatement для выполнения запросов с параметрами.

4. Реализация кастомных SQL-запросов: Напишите собственные SQL-запросы для поиска продуктов по имени или цене.

Контрольные вопросы

1. Что такое JdbcTemplate и как его использовать в Spring Boot?

2. Как выполнить базовые SQL-запросы с использованием JdbcTemplate?

3. Как настроить подключение к базе данных в Spring Boot с использованием JDBC?

4. В чем отличие между JdbcTemplate и JPA?

5. Как создать и настроить таблицы в базе данных с использованием SQL-запросов?

Лабораторная работа № 7

Тема: Операции с базами данных

Цель работы: Изучить основные операции с базами данных: создание, чтение, обновление и удаление (CRUD). Научиться использовать Spring Data JPA и JdbcTemplate для взаимодействия с реляционными базами данных и выполнения этих операций.

Порядок выполнения работы

1. Создание проекта с использованием Spring Boot Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект.

- Выберите зависимости: Spring Web, Spring Data JPA, H2 Database (или другую СУБД, с которой будете работать).

- Скачайте и откройте проект в вашей IDE.

2. Настройка базы данных:

- Откройте файл application.properties и добавьте настройки для базы данных H2 (или используйте настройки для другой базы данных, например, MySQL или PostgreSQL):

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```

- Если используете другую базу данных, настройте соответствующие параметры для подключения.

3. Создание сущностей и репозиториев:

- Создайте сущность Product, которая будет использоваться для хранения данных в базе:

```
@Entity
public class Product {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private Double price;
    // Геттеры и сеттеры
}
```

- Создайте интерфейс репозитория, который будет использовать Spring Data JPA для работы с сущностью Product:

```
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

4. Операции CRUD через Spring Data JPA:

- Создание: Внедрите метод для создания нового продукта:

```
public Product createProduct(Product product) {
    return productRepository.save(product);
}
```

- Чтение: Реализуйте методы для получения всех продуктов и одного продукта по ID:

```
public List<Product> getAllProducts() {
    return productRepository.findAll();
}
public Product getProductById(Long id) {
    return productRepository.findById(id).orElse(null);
}
```

- Обновление: Метод для обновления продукта по ID:

```
public Product updateProduct(Long id, Product updatedProduct) {
    if (productRepository.existsById(id)) {
        updatedProduct.setId(id);
        return productRepository.save(updatedProduct);
```

```
        }
        return null;
    }
```

- Удаление: Метод для удаления продукта по ID:

```
    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }
```

5. Реализация REST API:

- Создайте контроллер, который будет обеспечивать доступ к CRUD-операциям через HTTP:

```
    @RestController
    @RequestMapping("/products")
    public class ProductController {
        @Autowired
        private ProductService productService;
        @PostMapping
        public Product createProduct(@RequestBody Product product) {
            return productService.createProduct(product);
        }
        @GetMapping
        public List<Product> getAllProducts() {
            return productService.getAllProducts();
        }
        @GetMapping("/{id}")
        public Product getProductById(@PathVariable Long id) {
            return productService.getProductById(id);
        }
        @PutMapping("/{id}")
        public Product updateProduct(@PathVariable Long id, @RequestBody
Product updatedProduct) {
```

```
        return productService.updateProduct(id, updatedProduct);
    }
    @DeleteMapping("/{id}")
    public void deleteProduct(@PathVariable Long id) {
        productService.deleteProduct(id);
    }
}
```

6. Тестирование приложения:

- Запустите приложение через команду mvn spring-boot:run или через IDE.

- Проверьте работу API, используя Postman или cURL:

  - Создание продукта: POST-запрос на /products

  - Получение всех продуктов: GET-запрос на /products

  - Получение продукта по ID: GET-запрос на /products/{id}

  - Обновление продукта: PUT-запрос на /products/{id}

  - Удаление продукта: DELETE-запрос на /products/{id}

7. Просмотр данных в базе:

- Перейдите по адресу http://localhost:8080/h2-console для работы с консолью H2.

- Введите параметры подключения (URL: jdbc:h2:mem:testdb, Username: sa, Password: `password`) и выполните SQL-запросы для проверки данных в базе.

Варианты заданий

1. Работа с несколькими сущностями: Создайте несколько сущностей (например, Category и `Product`), настроив связь между ними (один ко многим) и реализуйте операции CRUD для обеих сущностей.

2. Поиск с фильтрацией: Реализуйте поиск продуктов по имени или цене через Spring Data JPA.

3. Добавление валидации данных: Используйте аннотации для валидации данных сущности Product, например, @NotNull, @Min, @Max, и

добавьте обработку ошибок при получении некорректных данных от пользователя.

4. Использование кастомных SQL-запросов: Напишите кастомные SQL-запросы в репозитории для выполнения более сложных операций с базой данных.

Контрольные вопросы

1. Что такое операции CRUD и как они реализуются с использованием Spring Data JPA?

2. Как создать и настроить репозиторий для работы с базой данных в Spring Boot?

3. Как организовать обработку HTTP-запросов для операций CRUD с помощью Spring Web?

4. В чем разница между методом findAll() и findById() в Spring Data JPA?

5. Как работает аннотация @GeneratedValue в Spring JPA и для чего она используется?

Лабораторная работа № 8

Тема: Настройка Data JPA

Цель работы: Ознакомиться с настройкой и использованием Spring Data JPA для работы с реляционными базами данных. Научиться правильно конфигурировать Spring Boot для работы с JPA и выполнять основные операции с данными, используя репозитории и аннотации.

Порядок выполнения работы

1. Создание проекта с использованием Spring Boot Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект.

- Выберите зависимости: Spring Web, Spring Data JPA, H2 Database (или другую СУБД, с которой будете работать).

- Скачайте и откройте проект в вашей IDE.

2. Настройка базы данных:

- Откройте файл application.properties и добавьте настройки для базы данных H2 (или используйте настройки для другой базы данных, например, MySQL или PostgreSQL):

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
spring.jpa.hibernate.ddl-auto=update
```

- Если используется другая база данных, настройте соответствующие параметры подключения (например, для MySQL).

3. Создание сущности и репозитория:

- Создание сущности:

- Создайте модель для работы с данными. Пример сущности Product:

```
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "price")
    private Double price;
    // Геттеры и сеттеры
}
```

- Создание репозитория:

- Создайте интерфейс репозитория для работы с сущностью Product. Для этого используйте интерфейс JpaRepository, который предоставляет базовые CRUD-операции

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

4. Конфигурация Spring Data JPA:

- В Spring Boot, как правило, не требуется создавать дополнительные конфигурационные классы для настройки JPA, так как настройки уже обеспечены через зависимости в application.properties.

- В случае необходимости настроить дополнительные параметры (например, включить логирование SQL-запросов), можно добавить следующие настройки:

```
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

5. Использование Spring Data JPA в сервисах:

- Создайте сервис, который будет использовать репозиторий для выполнения операций с базой данных. Пример сервиса для работы с продуктами:

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository productRepository;
    // Получение всех продуктов
    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }
    // Получение продукта по ID
    public Product getProductById(Long id) {
        return productRepository.findById(id).orElse(null);
    }
    // Добавление нового продукта
    public Product addProduct(Product product) {
        return productRepository.save(product);
    }
    // Удаление продукта
    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }
}
```

6. Создание REST API:

- Создайте контроллер для взаимодействия с пользователем через HTTP. Контроллер будет обеспечивать доступ к CRUD-операциям для сущности Product:

```
@RestController
@RequestMapping("/products")
public class ProductController {
    @Autowired
    private ProductService productService;
    @GetMapping
    public List<Product> getAllProducts() {
        return productService.getAllProducts();
    }
    @GetMapping("/{id}")
public Product getProductById(@PathVariable Long id) {
        return productService.getProductById(id);
    }
    @PostMapping
    public Product createProduct(@RequestBody Product product) {
        return productService.addProduct(product);
    }
    @DeleteMapping("/{id}")
    public void deleteProduct(@PathVariable Long id) {
        productService.deleteProduct(id);
    }
}
```

7. Тестирование приложения:

- Запустите приложение через команду mvn spring-boot:run или через вашу IDE.

- Используйте Postman или браузер для тестирования работы API:

- GET-запрос на /products для получения списка всех продуктов.

- GET-запрос на /products/{id} для получения продукта по ID.

- POST-запрос на /products для создания нового продукта.

- DELETE-запрос на /products/{id} для удаления продукта по ID.

8. Просмотр данных в базе через H2 Console:

- Перейдите по адресу http://localhost:8080/h2-console и войдите в консоль базы данных.

- Введите параметры подключения (URL: jdbc:h2:mem:testdb, Username: sa, Password: `password`) и выполните SQL-запросы для проверки данных в базе.

Варианты заданий

1. Реализация отношения один ко многим: Создайте две сущности, например, Category и Product, и настройте связь между ними. Реализуйте CRUD-операции для обеих сущностей.

2. Добавление валидации данных: Используйте аннотации, такие как @NotNull, @Min, @Max, для валидации данных сущности Product, и обработку ошибок при некорректных данных.

3. Поиск и фильтрация данных: Реализуйте методы для поиска продуктов по имени или цене с использованием Spring Data JPA.

4. Добавление сортировки: Реализуйте возможность сортировки данных по имени или цене с использованием параметров запросов в REST API.

Контрольные вопросы

1. Что такое Spring Data JPA и как его использовать в Spring Boot?

2. Как настроить связь между сущностями в Spring Data JPA?

3. Как использовать аннотацию @Entity и что она означает?

4. В чем разница между методами findAll() и findById() в Spring Data JPA?

5. Как настроить отображение SQL-запросов в консоли при использовании Spring Data JPA?

Лабораторная работа № 9

Тема: Интеграция Spring Data JPA с PostgreSQL

Цель работы: Изучить процесс настройки и использования Spring Data JPA с PostgreSQL. Научиться интегрировать Spring Boot приложение с PostgreSQL, настроить подключение к базе данных и использовать репозитории для выполнения операций CRUD.

Порядок выполнения работы

1. Создание проекта с использованием Spring Boot Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект.

- Выберите зависимости: Spring Web, Spring Data JPA, PostgreSQL Driver.

- Скачайте и откройте проект в вашей IDE.

2. Настройка PostgreSQL:

- Убедитесь, что PostgreSQL установлен на вашем компьютере, или создайте базу данных на сервере PostgreSQL.

- Создайте базу данных для вашего проекта, например:

```
CREATE DATABASE springbootdb;
```

- Создайте пользователя с правами на эту базу данных:

```
CREATE USER springuser WITH PASSWORD 'password';
ALTER ROLE springuser SET client_encoding TO 'utf8';
ALTER ROLE springuser SET default_transaction_isolation TO 'read committed';
ALTER ROLE springuser SET timezone TO 'UTC';
GRANT ALL PRIVILEGES ON DATABASE springbootdb TO springuser;
```

3. Настройка подключения к PostgreSQL:

- В файле application.properties добавьте параметры для подключения к базе данных PostgreSQL:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/springbootdb
spring.datasource.username=springuser
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.datasource.driverClassName=org.postgresql.Driver
```

- Эти настройки обеспечат подключение к базе данных springbootdb, используя пользователя springuser с паролем password.

4. Создание сущностей и репозиториев:

- Создайте сущность Product, которая будет храниться в PostgreSQL:

```
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "name", nullable = false)
    private String name;
    @Column(name = "price")
    private Double price;
    // Геттеры и сеттеры
}
```

- Создайте интерфейс репозитория для работы с сущностью Product:

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

5. Конфигурация Spring Data JPA для PostgreSQL:

- Spring Boot автоматически конфигурирует JPA для работы с

PostgreSQL, если в проекте присутствует зависимость spring-boot-starter-data-jpa и postgresql. Но при необходимости, можно добавить настройки в классе конфигурации:

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages = "com.example.repository")
@EntityScan(basePackages = "com.example.entity")
public class JpaConfig {
   @Bean
   public LocalContainerEntityManagerFactoryBean entityManagerFactory(
      EntityManagerFactoryBuilder builder, DataSource dataSource) {
      return builder
            .dataSource(dataSource)
            .packages("com.example.entity")
            .persistenceUnit("products")
            .build();
   }
}
```

6. Использование Spring Data JPA в сервисах:

- Создайте сервис, который будет использовать репозиторий для выполнения операций CRUD с продуктами:

```
@Service
public class ProductService {
   @Autowired
   private ProductRepository productRepository;
   // Получение всех продуктов
   public List<Product> getAllProducts() {
      return productRepository.findAll();
   }
```

```
    // Получение продукта по ID
    public Product getProductById(Long id) {
        return productRepository.findById(id).orElse(null);
    }
    // Создание нового продукта
public Product addProduct(Product product) {
        return productRepository.save(product);
    }
    // Удаление продукта по ID
    public void deleteProduct(Long id) {
        productRepository.deleteById(id);
    }
}
```

7. Создание REST API:

- Создайте контроллер для взаимодействия с пользователем через HTTP. Контроллер будет обеспечивать доступ к CRUD-операциям для сущности Product:

```
@RestController
@RequestMapping("/products")
public class ProductController {
    @Autowired
    private ProductService productService;
    @GetMapping
    public List<Product> getAllProducts() {
        return productService.getAllProducts();
    }
    @GetMapping("/{id}")
    public Product getProductById(@PathVariable Long id) {
        return productService.getProductById(id);
    }
```

```
    @PostMapping
    public Product createProduct(@RequestBody Product product) {
        return productService.addProduct(product);
    }
    @DeleteMapping("/{id}")
    public void deleteProduct(@PathVariable Long id) {
        productService.deleteProduct(id);
    }
}
```

8. Тестирование приложения:

- Запустите приложение через команду mvn spring-boot:run или через вашу IDE.

- Используйте Postman или браузер для тестирования работы API:

  - GET-запрос на /products для получения списка всех продуктов.

  - GET-запрос на /products/{id} для получения продукта по ID.

  - POST-запрос на /products для создания нового продукта.

  - DELETE-запрос на /products/{id} для удаления продукта по ID.

9. Просмотр данных в базе через pgAdmin или psql:

- Подключитесь к базе данных PostgreSQL с помощью pgAdmin или psql.

- Выполните SQL-запросы, чтобы проверить данные в базе:

SELECT * FROM products;

Варианты заданий

1. Работа с несколькими сущностями и связями: Создайте две сущности (например, Category и `Product`) и настройте связь между ними (один ко многим). Реализуйте CRUD-операции для обеих сущностей.

2. Миграции базы данных с Flyway или Liquibase: Настройте миграции для базы данных с использованием Flyway или Liquibase и примените их для создания схемы базы данных.

3. Поиск с фильтрацией: Реализуйте фильтрацию продуктов по имени

или цене с использованием Spring Data JPA.

4. Использование кастомных SQL-запросов: Напишите кастомные SQL-запросы в репозитории для выполнения более сложных операций с базой данных (например, поиск по диапазону цен).

Контрольные вопросы

1. Как настроить Spring Data JPA для работы с PostgreSQL в Spring Boot?

2. Какие основные отличия в настройке подключения базы данных PostgreSQL и других СУБД?

3. Как реализовать работу с несколькими сущностями и связями между ними в Spring Data JPA?

4. Как настроить миграции базы данных с использованием Flyway или Liquibase?

5. В чем заключается использование аннотации @GeneratedValue в JPA, как она работает с PostgreSQL?

Лабораторная работа № 10

Тема: Maven методы запроса

Цель работы: Освоение работы с инструментом управления зависимостями и сборки проектов Maven, а также использование методов запроса с помощью Spring Data JPA. Научиться создавать запросы с использованием стандартных методов Spring Data JPA и настраивать их для использования в приложении Spring Boot.

Порядок выполнения работы

1. Создание проекта с использованием Spring Boot Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект.

- Выберите зависимости: Spring Web, Spring Data JPA, PostgreSQL Driver (или другую базу данных по выбору).

- Скачайте и откройте проект в вашей IDE.

2. Настройка базы данных:

- Настройте подключение к базе данных в файле application.properties:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/yourdb
spring.datasource.username=youruser
spring.datasource.password=yourpassword
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

3. Создание сущностей:

- Создайте сущность Product, которая будет храниться в базе данных. Включите несколько полей для демонстрации методов запроса.

```
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
    @Column(name = "name", nullable = false)
    private String name;
    @Column(name = "price")
    private Double price;
    @Column(name = "category")
    private String category;
    // Геттеры и сеттеры
}
```

4. Создание репозитория:

- Создайте интерфейс репозитория для работы с сущностью Product. Используйте стандартные методы Spring Data JPA.

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
    List<Product> findByCategory(String category);
    List<Product> findByPriceBetween(Double minPrice, Double maxPrice);
    List<Product> findByNameContaining(String name);
}
```

5. Создание сервисов:

- Создайте сервис, который будет использовать репозиторий для выполнения методов запроса.

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository productRepository;
    // Получение всех продуктов по категории
    public List<Product> getProductsByCategory(String category) {
        return productRepository.findByCategory(category);
```

```
    }

    // Получение продуктов по диапазону цены
    public List<Product> getProductsByPriceRange(Double minPrice, Double maxPrice) {
        return productRepository.findByPriceBetween(minPrice, maxPrice);
    }
    // Получение продуктов по частичному совпадению в имени
    public List<Product> getProductsByName(String name) {
        return productRepository.findByNameContaining(name);
    }
}
```

6. Создание REST API для работы с продуктами:

- Создайте контроллер, который будет предоставлять доступ к сервису через HTTP:

```
@RestController
@RequestMapping("/products")
public class ProductController {
    @Autowired
    private ProductService productService;
    @GetMapping("/category/{category}")
    public List<Product> getProductsByCategory(@PathVariable String category) {
        return productService.getProductsByCategory(category);
    }
    @GetMapping("/price")
    public List<Product> getProductsByPriceRange(@RequestParam Double minPrice, @RequestParam Double maxPrice) {
        return productService.getProductsByPriceRange(minPrice, maxPrice);
```

```
    }

    @GetMapping("/name")
    public List<Product> getProductsByName(@RequestParam String name) {
        return productService.getProductsByName(name);
    }
}
```

7. Тестирование приложения:

- Запустите приложение через команду mvn spring-boot:run или через вашу IDE.

- Используйте Postman или браузер для тестирования работы API:

- GET-запрос на /products/category/{category} для получения продуктов по категории.

- GET-запрос на /products/price?minPrice={minPrice}&maxPrice={maxPrice} для получения продуктов по диапазону цен.

- GET-запрос на /products/name?name={name} для получения продуктов по частичному совпадению имени.

8. Просмотр данных в базе через pgAdmin или psql:

- Подключитесь к базе данных PostgreSQL с помощью pgAdmin или psql.

- Выполните SQL-запросы для проверки результатов работы методов запроса:

```
SELECT * FROM products WHERE category = 'Electronics';
SELECT * FROM products WHERE price BETWEEN 100 AND 500;
SELECT * FROM products WHERE name LIKE '%Phone%';
```

Варианты заданий

1. Использование аннотации @Query: Напишите кастомные запросы с использованием аннотации @Query. Например, запросы для поиска продуктов по нескольким параметрам.

2. Обработка ошибок в контроллере: Реализуйте обработку ошибок (например, 404 — продукт не найден) с использованием аннотации @ExceptionHandler.

3. Пагинация и сортировка: Добавьте поддержку пагинации и сортировки в запросы, используя параметры Spring Data JPA.

Контрольные вопросы

1. Какой тип возвращаемых значений используют методы репозитория Spring Data JPA для выполнения запроса?

2. Что такое метод запроса в Spring Data JPA, и как его можно использовать?

3. Как с помощью аннотации @Query можно создавать собственные SQL-запросы в Spring Data JPA?

4. Чем отличается метод findByCategory() от метода findByPriceBetween() в репозитории?

5. Как настроить пагинацию и сортировку в Spring Data JPA запросах?

Лабораторная работа № 11

Тема: Проект Spring Boot, Data JPA, DB

Цель работы: Настройка интеграции Spring Data JPA с базой данных в проекте Spring Boot. Овладение навыками работы с репозиториями, сущностями, выполнением CRUD-операций и использованием базы данных в приложении на платформе Spring Boot.

Порядок выполнения работы

1. Создание проекта с использованием Spring Boot Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект с зависимостями: Spring Web, Spring Data JPA, PostgreSQL Driver (или другая база данных по выбору).

- Скачайте и откройте проект в вашей IDE.

2. Настройка базы данных:

- Настройте подключение к базе данных в файле application.properties:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/yourdb
spring.datasource.username=youruser
spring.datasource.password=yourpassword
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

3. Создание сущностей:

- Создайте сущности, которые будут храниться в базе данных. Например, создайте сущности для Product и Category.

- Сущность Product:

```
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    @Column(name = "name", nullable = false)
    private String name;
    @Column(name = "price")
    private Double price;
    @ManyToOne
    @JoinColumn(name = "category_id")
    private Category category;
    // Геттеры и сеттеры
}
```

- Сущность Category:

```
@Entity
@Table(name = "categories")
public class Category {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "name", nullable = false)
    private String name;
    // Геттеры и сеттеры
}
```

4. Создание репозиториев:

- Создайте интерфейсы репозиториев для работы с сущностями Product и Category:

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
    List<Product> findByCategory(Category category);
}
@Repository
```

```
public interface CategoryRepository extends JpaRepository<Category, Long> {
}
```

5. Создание сервисов:

- Создайте сервисы для работы с репозиториями. Например, сервис ProductService может включать методы для добавления, удаления и получения продуктов, а также получения продуктов по категории:

```
@Service
public class ProductService {
    @Autowired
    private ProductRepository productRepository;
    @Autowired
    private CategoryRepository categoryRepository;
    public Product addProduct(Product product) {
        return productRepository.save(product);
    }
    public List<Product> getProductsByCategory(Long categoryId) {
        Category category = categoryRepository.findById(categoryId).orElseThrow(() -> new RuntimeException("Category not found"));
        return productRepository.findByCategory(category);
    }
    public void deleteProduct(Long productId) {
        productRepository.deleteById(productId);
    }
}
```

6. Создание REST API:

- Создайте контроллер, который будет предоставлять доступ к методам сервиса через HTTP. Например, создайте методы для добавления, получения и удаления продуктов:

```
@RestController
@RequestMapping("/products")
public class ProductController {
  @Autowired
  private ProductService productService;
  @PostMapping
  public ResponseEntity<Product> addProduct(@RequestBody Product product) {
    Product savedProduct = productService.addProduct(product);
    return ResponseEntity.status(HttpStatus.CREATED).body(savedProduct);
  }
  @GetMapping("/category/{categoryId}")
  public List<Product> getProductsByCategory(@PathVariable Long categoryId) {
    return productService.getProductsByCategory(categoryId);
  }
  @DeleteMapping("/{productId}")
  public ResponseEntity<Void> deleteProduct(@PathVariable Long productId) {
    productService.deleteProduct(productId);
    return ResponseEntity.noContent().build();
  }
}
```

7. Тестирование приложения:

- Запустите приложение через команду mvn spring-boot:run или через вашу IDE.

- Используйте Postman или браузер для тестирования работы API:

- POST-запрос на /products для добавления нового продукта.

- GET-запрос на /products/category/{categoryId} для получения продуктов по категории.

- DELETE-запрос на /products/{productId} для удаления продукта.

8. Просмотр данных в базе через pgAdmin или psql:

- Подключитесь к базе данных PostgreSQL с помощью pgAdmin или psql.

- Выполните SQL-запросы для проверки данных:

SELECT * FROM products WHERE category_id = 1;

SELECT * FROM categories;

Варианты заданий

1. Реализация поиска продуктов по цене: Добавьте функциональность для поиска продуктов, цена которых больше или меньше заданного значения.

2. Обновление данных: Реализуйте метод для обновления информации о продукте через REST API.

3. Пагинация и сортировка: Добавьте поддержку пагинации и сортировки в запросы для получения продуктов.

Контрольные вопросы

1. Как настроить связь «многие к одному» в JPA?
2. В чем разница между методами save() и deleteById() репозитория?
3. Как обрабатывать ошибки, если категория или продукт не найдены?
4. Что делает аннотация @ManyToOne в JPA?
5. Как реализовать пагинацию и сортировку в Spring Data JPA?

Лабораторная работа № 12

Тема: Spring Security

Цель работы: Освоение механизма аутентификации и авторизации с использованием Spring Security. Настройка безопасности приложения на основе ролей и прав доступа для создания защищенных REST API и веб-приложений.

Порядок выполнения работы

1. Создание проекта с использованием Spring Boot Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект с зависимостями: Spring Web, Spring Security, Spring Data JPA, H2 Database (или другая база данных по выбору).

- Скачайте и откройте проект в вашей IDE.

2. Настройка Spring Security:

- Для начала включите Spring Security в проекте. По умолчанию Spring Security добавляет базовую настройку безопасности с формой аутентификации.

- Откройте файл application.properties и добавьте конфигурацию для безопасности:

```
spring.security.user.name=user
spring.security.user.password=password
spring.security.user.roles=USER
```

3. Создание сущностей и репозиториев:

- Создайте сущности для пользователей и ролей. Например:

- User:

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    @Column(name = "username", nullable = false, unique = true)
    private String username;

    @Column(name = "password", nullable = false)
    private String password;
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "user_roles",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles;
  }
```

- Role:

```
  @Entity
  @Table(name = "roles")
  public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "role_name", nullable = false, unique = true)
    private String roleName;
  }
```

- Создайте репозитории для этих сущностей:

```
  @Repository
  public interface UserRepository extends JpaRepository<User, Long> {
     Optional<User> findByUsername(String username);
  }
  @Repository
  public interface RoleRepository extends JpaRepository<Role, Long> {
  }
```

4. Конфигурация Spring Security:

- Настройте базовую конфигурацию безопасности, создав класс SecurityConfig:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
  @Autowired
  private UserDetailsService userDetailsService;
  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .authorizeRequests()
        .antMatchers("/admin/**").hasRole("ADMIN")
        .antMatchers("/user/**").hasRole("USER")
        .anyRequest().authenticated()
      .and()
      .formLogin()
        .loginPage("/login")
        .permitAll()
      .and()
      .logout()
        .permitAll();
  }
  @Override
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
  }
  @Bean
  public PasswordEncoder passwordEncoder() {
```

```
        return new BCryptPasswordEncoder();
    }
}
```

- В конфигурации укажите, какие страницы или ресурсы должны быть доступны только пользователям с определенной ролью. В этом примере /admin/** доступно только пользователям с ролью ADMIN, а /user/** — пользователям с ролью USER.

5. Создание и настройка сервиса для аутентификации:

- Создайте класс CustomUserDetailsService для загрузки пользователя из базы данных:

```
@Service
public class CustomUserDetailsService implements UserDetailsService {
    @Autowired
    private UserRepository userRepository;
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        User user = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("User not found"));
        return new org.springframework.security.core.userdetails.User(user.getUsername(), user.getPassword(), getAuthorities(user));
    }
    private Collection<? extends GrantedAuthority> getAuthorities(User user) {
        Set<GrantedAuthority> authorities = new HashSet<>();
        for (Role role : user.getRoles()) {
            authorities.add(new SimpleGrantedAuthority(role.getRoleName()));
```

```
        }
        return authorities;
    }
}
```

6. Создание контроллеров:

- Создайте контроллер для управления доступом к защищенным страницам:

```
@RestController
@RequestMapping("/user")
public class UserController {
    @GetMapping
    public String getUser() {
        return "Hello, User!";
    }
}
@RestController
@RequestMapping("/admin")
public class AdminController {
    @GetMapping
    public String getAdmin() {
        return "Hello, Admin!";
    }
}
```

7. Запуск приложения:

- Запустите приложение с помощью команды mvn spring-boot:run или через вашу IDE.

- Перейдите на страницу /login в браузере, чтобы протестировать форму аутентификации.

- Проверьте доступ к защищенным страницам, например, /admin (доступно только пользователям с ролью `ADMIN`).

8. Тестирование приложения:

- Проверьте, как работают различные роли и доступ к ресурсам:

- Пользователь с ролью USER может получить доступ к /user/**.

- Пользователь с ролью ADMIN может получить доступ к /admin/**.

- Попытка получить доступ без аутентификации или с неправильной ролью должна вернуть ошибку доступа.

Варианты заданий

1. Добавление новых ролей: Реализуйте систему добавления новых ролей для пользователей через REST API или форму администрирования.

2. Создание пользовательского контроллера: Создайте контроллер, который будет возвращать список пользователей и их ролей только для администраторов.

3. Интеграция с JWT: Замените аутентификацию на JWT (JSON Web Token), чтобы использовать токены вместо сессий.

Контрольные вопросы

1. Как работает аутентификация в Spring Security?

2. Как настроить доступ к различным страницам в зависимости от ролей пользователей?

3. Что такое GrantedAuthority и как она используется в Spring Security?

4. Какие методы необходимо переопределить в классе WebSecurityConfigurerAdapter?

5. Как использовать BCrypt для шифрования паролей?

Лабораторная работа № 13

Тема: Введение в веб-сервисы RESTful

Цель работы: владение основами разработки и использования веб-сервисов RESTful. Знакомство с принципами работы REST и создание простого REST API на платформе Spring Boot.

Порядок выполнения работы

1. Создание проекта с использованием Spring Boot Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект с зависимостями: Spring Web.

- Скачайте и откройте проект в вашей IDE.

2. Основные принципы REST:

- REST (Representational State Transfer) — архитектурный стиль, основанный на использовании стандартных HTTP-методов (GET, POST, PUT, DELETE) для работы с ресурсами.

- В RESTful-сервисах каждый ресурс идентифицируется уникальным URI, а действия с ресурсами выполняются через соответствующие HTTP-методы.

- Важные принципы REST:

- Stateless — сервер не хранит информацию о состоянии клиента.

- Uniform Interface — использование стандартных методов для взаимодействия с ресурсами.

- Client-Server — разделение клиентской и серверной части.

3. Создание первого REST API:

- Создайте класс-сущность Product:

```
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    @Column(name = "name")
    private String name;
    @Column(name = "price")
    private Double price;
    // Геттеры и сеттеры
}
```

- Создайте репозиторий для работы с сущностью Product:

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

4. Создание контроллера для работы с REST API:

- В контроллере определите методы, соответствующие действиям с ресурсами:

- GET /products — для получения списка всех продуктов.
- GET /products/{id} — для получения информации о продукте по ID.
- POST /products — для создания нового продукта.
- PUT /products/{id} — для обновления информации о продукте.
- DELETE /products/{id} — для удаления продукта.

Пример контроллера:

```
@RestController
@RequestMapping("/products")
public class ProductController {
    @Autowired
    private ProductRepository productRepository;
    @GetMapping
    public List<Product> getAllProducts() {
        return productRepository.findAll();
```

```
    }
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable
Long id) {
        Product product = productRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Product not
found"));
        return ResponseEntity.ok(product);
    }
    @PostMapping
    public Product createProduct(@RequestBody Product product) {
        return productRepository.save(product);
    }
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long
id, @RequestBody Product productDetails) {
        Product product = productRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Product not
found"));
        product.setName(productDetails.getName());
        product.setPrice(productDetails.getPrice());
        productRepository.save(product);
        return ResponseEntity.ok(product);
    }
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Long id)
{
        Product product = productRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Product not
found"));
```

```
        productRepository.delete(product);
        return ResponseEntity.noContent().build();
    }
}
```

5. Запуск и тестирование API:

- Запустите приложение с помощью команды mvn spring-boot:run или через вашу IDE.

- Используйте Postman или браузер для тестирования методов:

- GET запрос на /products для получения всех продуктов.

- GET запрос на /products/{id} для получения продукта по ID.

- POST запрос на /products для создания нового продукта.

- PUT запрос на /products/{id} для обновления продукта.

- DELETE запрос на /products/{id} для удаления продукта.

6. Реализация обработки ошибок:

- Обработайте ошибки с помощью аннотации @ExceptionHandler или @ControllerAdvice для централизованной обработки исключений.

- Например, можно добавить обработку ошибки, если продукт не найден:

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> handleRuntimeException(RuntimeException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }
}
```

7. Документирование API (опционально):

- Для автоматического документирования API можно использовать библиотеку [Springfox Swagger](https://swagger.io/tools/swagger-ui/) или [Springdoc OpenAPI](https://springdoc.org/).

- Добавьте зависимость в pom.xml:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-ui</artifactId>
    <version>1.6.9</version>
</dependency>
```

- Теперь документация будет доступна по адресу /swagger-ui.html.

Варианты заданий

1. Поиск продуктов по имени: Добавьте возможность поиска продуктов по имени, используя метод в репозитории findByName.

2. Пагинация: Реализуйте пагинацию для метода получения всех продуктов. Используйте интерфейс PagingAndSortingRepository.

3. Добавление авторизации в API: Используя Spring Security, добавьте базовую аутентификацию для доступа к API.

Контрольные вопросы

1. Что такое REST и какие основные принципы лежат в основе этого подхода?

2. В чем разница между методами GET, POST, PUT и DELETE в контексте REST API?

3. Как обработать ошибки в REST API с использованием @ExceptionHandler?

4. Как в Spring Boot настроить пагинацию для запросов к базе данных?

5. Как документировать REST API с использованием Swagger или OpenAPI?

Лабораторная работа № 14

Тема: Подходы и реализация REST

Цель работы: Изучение различных подходов к разработке RESTful веб-сервисов, а также освоение принципов реализации REST API с различными уровнями безопасности и гибкости.

Порядок выполнения работы

1. Обзор различных подходов к реализации REST API:

- Стандартный REST:

- Веб-сервис, который использует стандартные HTTP-методы (GET, POST, PUT, DELETE) для взаимодействия с ресурсами. Ресурсы идентифицируются через URI, а действия выполняются через соответствующие HTTP-методы.

- HATEOAS (Hypermedia as the engine of application state):

- Принцип REST, согласно которому клиент взаимодействует с API не только через статичные URL, но и через гипермедиа, что позволяет серверу предоставлять клиенту ссылки на дальнейшие действия.

- Гибкость REST API:

- В REST можно использовать разные форматы представления данных, например, JSON, XML или даже CSV. Важно, чтобы API мог легко масштабироваться, поддерживать различные клиентские приложения и обеспечивать понятность взаимодействий.

2. Создание проекта с использованием Spring Boot Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект с зависимостями: Spring Web, Spring Data JPA, Spring Boot DevTools.

- Скачайте и откройте проект в вашей IDE.

3. Создание сущности и репозитория:

- Создайте сущность Book для представления ресурса книги:

```
@Entity
@Table(name = "books")
```

```
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "title")
    private String title;
    @Column(name = "author")
    private String author;
    @Column(name = "isbn")
    private String isbn;
    @Column(name = "published_date")
    private LocalDate publishedDate;
    // Геттеры и сеттеры
}
```

- Создайте репозиторий для работы с книгами:

```
@Repository
public interface BookRepository extends JpaRepository<Book, Long> {
}
```

4. Создание контроллера для работы с REST API:

- В контроллере определите методы для CRUD операций с ресурсом:
  - GET /books — для получения списка всех книг.
  - GET /books/{id} — для получения книги по ID.
  - POST /books — для добавления новой книги.
  - PUT /books/{id} — для обновления книги.
  - DELETE /books/{id} — для удаления книги.

Пример контроллера:

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
```

```
    private BookRepository bookRepository;
    @GetMapping
    public List<Book> getAllBooks() {
        return bookRepository.findAll();
    }
    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable Long id)
{
        Book book = bookRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Book not
found"));
        return ResponseEntity.ok(book);
    }
    @PostMapping
    public Book createBook(@RequestBody Book book) {
        return bookRepository.save(book);
    }
    @PutMapping("/{id}")
    public ResponseEntity<Book> updateBook(@PathVariable Long id,
@RequestBody Book bookDetails) {
        Book book = bookRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Book not
found"));
        book.setTitle(bookDetails.getTitle());
        book.setAuthor(bookDetails.getAuthor());
        book.setIsbn(bookDetails.getIsbn());
        book.setPublishedDate(bookDetails.getPublishedDate());
        bookRepository.save(book);
        return ResponseEntity.ok(book);
    }
```

```
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteBook(@PathVariable Long id) {
        Book book = bookRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Book not found"));
        bookRepository.delete(book);
        return ResponseEntity.noContent().build();
    }
}
```

5. Реализация HATEOAS (опционально):

- Для добавления гипермедиа в ответы API, используйте библиотеку Spring HATEOAS. Она позволяет возвращать ссылки на другие ресурсы вместе с основным ответом.

- Добавьте зависимость в pom.xml:

```
<dependency>
    <groupId>org.springframework.hateoas</groupId>
    <artifactId>spring-hateoas</artifactId>
    <version>1.0.0.M4</version>
</dependency>
```

- Модифицируйте контроллер, чтобы он возвращал гипермедиа-ресурсы:

```
import org.springframework.hateoas.EntityModel;
import org.springframework.hateoas.Link;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*;
@GetMapping("/{id}")
public EntityModel<Book> getBookById(@PathVariable Long id) {
    Book book = bookRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Book not found"));
    EntityModel<Book> resource = EntityModel.of(book);
```

```
        Link selfLink =
linkTo(methodOn(BookController.class).getBookById(id)).withSelfRel();
        resource.add(selfLink);
        return resource;
    }
```

6. Запуск и тестирование API:

- Запустите приложение с помощью команды mvn spring-boot:run или через вашу IDE.

- Используйте Postman или браузер для тестирования методов:

  - GET запрос на /books для получения всех книг.

  - GET запрос на /books/{id} для получения книги по ID.

  - POST запрос на /books для добавления новой книги.

  - PUT запрос на /books/{id} для обновления книги.

  - DELETE запрос на /books/{id} для удаления книги.

7. Реализация пагинации:

- Для поддержки пагинации в REST API можно использовать Pageable в методах контроллера.

- Например, для метода GET /books:

```
@GetMapping
public Page<Book> getAllBooks(Pageable pageable) {
    return bookRepository.findAll(pageable);
}
```

- Пагинация будет работать через запросы с параметрами page и size, например, /books?page=0&size=10.

8. Документирование API с использованием Swagger (опционально):

- Для автоматической документации API можно использовать библиотеку [Springfox Swagger](https://swagger.io/tools/swagger-ui/).

- Добавьте зависимость в pom.xml:

```
<dependency>
    <groupId>io.springfox</groupId>
```

```
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
  </dependency>
```

- Документация будет доступна по адресу /swagger-ui/.

Варианты заданий

1. Использование фильтров и сортировки: Добавьте поддержку фильтрации и сортировки для API, например, по названию книги или автору.

2. Добавление логирования: Реализуйте логирование запросов и ответов, чтобы отслеживать все действия в вашем API.

3. Авторизация и аутентификация: Настройте базовую аутентификацию с использованием Spring Security для защиты API от несанкционированного доступа.

Контрольные вопросы

1. В чем заключается отличие между стандартной реализацией REST API и реализацией с использованием HATEOAS?

2. Как работает пагинация в Spring Data JPA и как она интегрируется с REST API?

3. Какие HTTP-методы используются для создания, чтения, обновления и удаления ресурсов в REST?

4. Как можно реализовать сортировку и фильтрацию данных в REST API?

5. Как использовать Spring HATEOAS для добавления гипермедиа-ссылок в ответы REST API?

Лабораторная работа № 15

Тема: Создание REST API используя Spring Boot

Цель работы: Освоение процесса разработки REST API с использованием Spring Boot, включающее настройку приложения, создание контроллеров, взаимодействие с базами данных через Spring Data JPA, а также обработку запросов и ответов в формате JSON.

Порядок выполнения работы

1. Создание проекта с использованием Spring Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект с зависимостями:

- Spring Web

- Spring Data JPA

- H2 Database (или PostgreSQL, если требуется)

- Spring Boot DevTools для упрощения разработки

- Скачайте и откройте проект в вашей IDE.

2. Создание сущности:

- Создайте сущность Book, которая будет представлять ресурс для работы в API. Сущность должна быть связана с базой данных через аннотации JPA:

```
@Entity
@Table(name = "books")
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "title")
    private String title;
    @Column(name = "author")
    private String author;
    @Column(name = "isbn")
```

```
    private String isbn;

    @Column(name = "published_date")
    private LocalDate publishedDate;
    // Геттеры и сеттеры
}
```

3. Создание репозитория для работы с сущностью:

- Создайте репозиторий с использованием Spring Data JPA для работы с базой данных:

```
@Repository
public interface BookRepository extends JpaRepository<Book, Long> {
}
```

4. Создание контроллера REST:

- Контроллер будет обрабатывать запросы на создание, чтение, обновление и удаление ресурсов. Для этого создайте класс BookController с методами для обработки HTTP-запросов:

```
@RestController
@RequestMapping("/books")
public class BookController {
    @Autowired
    private BookRepository bookRepository;
    // Получение списка всех книг
    @GetMapping
    public List<Book> getAllBooks() {
        return bookRepository.findAll();
    }
    // Получение книги по ID
    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable Long id)
{
```

```
        Book book = bookRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Book not found"));
        return ResponseEntity.ok(book);
    }
    // Создание новой книги
    @PostMapping
    public Book createBook(@RequestBody Book book) {
        return bookRepository.save(book);
    }
    // Обновление информации о книге
    @PutMapping("/{id}")
    public ResponseEntity<Book> updateBook(@PathVariable Long id, @RequestBody Book bookDetails) {
        Book book = bookRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Book not found"));
        book.setTitle(bookDetails.getTitle());
        book.setAuthor(bookDetails.getAuthor());
        book.setIsbn(bookDetails.getIsbn());
        book.setPublishedDate(bookDetails.getPublishedDate());
        bookRepository.save(book);
        return ResponseEntity.ok(book);
    }
    // Удаление книги по ID
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteBook(@PathVariable Long id) {
        Book book = bookRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Book not found"));
```

```
        bookRepository.delete(book);
        return ResponseEntity.noContent().build();
    }
}
```

5. Тестирование API с использованием Postman:

- Запустите приложение с помощью команды mvn spring-boot:run или через вашу IDE.

- Откройте Postman или браузер для тестирования следующих запросов:

- GET запрос на /books — для получения списка всех книг.

- GET запрос на /books/{id} — для получения информации о книге по ID.

- POST запрос на /books — для создания новой книги.

- PUT запрос на /books/{id} — для обновления информации о книге.

- DELETE запрос на /books/{id} — для удаления книги.

6. Добавление обработки ошибок:

- Добавьте обработку ошибок для случаев, когда ресурс не найден или при нарушении данных. Создайте кастомные исключения и обработчики ошибок:

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<Object> handleResourceNotFound(ResourceNotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
    }
    @ExceptionHandler(Exception.class)
    public ResponseEntity<Object> handleException(Exception ex) {
```

```
            return new ResponseEntity<>(ex.getMessage(),
HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
```

7. Настройка документации с использованием Swagger (опционально):

- Для автоматической документации REST API добавьте зависимость Swagger в pom.xml:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

- Swagger автоматически создает документацию по доступным эндпоинтам. Документация будет доступна по адресу: http://localhost:8080/swagger-ui/.

Варианты заданий

1. Поддержка фильтрации и сортировки:

- Реализуйте фильтрацию по параметрам, например, фильтрацию по автору или названию книги. Также добавьте возможность сортировки по различным полям.

2. Авторизация через JWT:

- Реализуйте авторизацию с использованием JWT (JSON Web Token) для защиты API и доступа к его методам только после аутентификации.

3. Работа с различными форматами данных:

- Добавьте поддержку различных форматов данных для API, таких как XML или CSV, наряду с JSON. Реализуйте выбор формата через заголовки запроса.

Контрольные вопросы

1. Что такое REST API и чем он отличается от других типов API?

2. Какие HTTP-методы используются для выполнения CRUD операций в REST?

3. Как в Spring Boot обрабатываются запросы и ответы в формате JSON?

4. Как можно защитить REST API с помощью Spring Security?

5. Что такое Swagger, и как он помогает в документировании REST API?

Лабораторная работа № 16

Тема: RestController

Цель работы: Овладение созданием и использованием RestController в Spring Boot для обработки HTTP-запросов в приложениях, основанных на архитектуре REST. Изучение аннотаций и механизмов Spring, используемых для создания API, а также принципов обработки запросов и отправки ответов.

Порядок выполнения работы

1. Создание проекта с использованием Spring Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект с зависимостями:

- Spring Web

- Spring Boot DevTools для ускоренной разработки

- Скачайте и откройте проект в вашей IDE.

2. Создание сущности:

- Для этого задания создайте сущность, которая будет представлять ресурс API. Например, создайте сущность Employee для хранения информации о сотрудниках:

```
@Entity
@Table(name = "employees")
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "first_name")
    private String firstName;
    @Column(name = "last_name")
    private String lastName;
    @Column(name = "email")
    private String email;
```

```
    // Геттеры и сеттеры
}
```

3. Создание репозитория для работы с сущностью:

- Создайте репозиторий с использованием Spring Data JPA для работы с базой данных:

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
}
```

4. Создание RestController:

- Создайте контроллер с использованием аннотации @RestController. Этот контроллер будет обрабатывать HTTP-запросы на получение списка сотрудников и на получение информации о сотруднике по ID.

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {
    @Autowired
    private EmployeeRepository employeeRepository;
    // Получение списка всех сотрудников
    @GetMapping
    public List<Employee> getAllEmployees() {
        return employeeRepository.findAll();
    }
    // Получение сотрудника по ID
    @GetMapping("/{id}")
    public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id) {
        Employee employee = employeeRepository.findById(id)
```

```
                .orElseThrow(() -> new RuntimeException("Employee not found"));
        return ResponseEntity.ok(employee);
    }
    // Создание нового сотрудника
    @PostMapping
    public Employee createEmployee(@RequestBody Employee employee) {
        return employeeRepository.save(employee);
    }
    // Обновление сотрудника по ID
    @PutMapping("/{id}")
    public ResponseEntity<Employee> updateEmployee(@PathVariable Long id, @RequestBody Employee employeeDetails) {
        Employee employee = employeeRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Employee not found"));
        employee.setFirstName(employeeDetails.getFirstName());
        employee.setLastName(employeeDetails.getLastName());
        employee.setEmail(employeeDetails.getEmail());
        employeeRepository.save(employee);
        return ResponseEntity.ok(employee);
    }
    // Удаление сотрудника по ID
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteEmployee(@PathVariable Long id) {
        Employee employee = employeeRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Employee not found"));
```

```
        employeeRepository.delete(employee);
        return ResponseEntity.noContent().build();
    }
}
```

5. Тестирование API с использованием Postman:

- Запустите приложение с помощью команды mvn spring-boot:run или через вашу IDE.

- Откройте Postman для тестирования следующих запросов:

- GET запрос на /employees — для получения списка всех сотрудников.

- GET запрос на /employees/{id} — для получения информации о сотруднике по ID.

- POST запрос на /employees — для создания нового сотрудника.

- PUT запрос на /employees/{id} — для обновления информации о сотруднике.

- DELETE запрос на /employees/{id} — для удаления сотрудника.

6. Обработка ошибок в RestController:

- Добавьте обработку ошибок для случаев, когда ресурс не найден. Например, создайте исключение EmployeeNotFoundException:

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class EmployeeNotFoundException extends RuntimeException {
    public EmployeeNotFoundException(String message) {
        super(message);
    }
}
```

- Используйте это исключение в контроллере:

```
@GetMapping("/{id}")
public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id) {
```

```
        Employee employee = employeeRepository.findById(id)
                .orElseThrow(() -> new EmployeeNotFoundException("Employee not found"));
        return ResponseEntity.ok(employee);
    }
```

7. Настройка Swagger для документации API (опционально):

- Добавьте зависимость Swagger в pom.xml для автоматического документирования API:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

- Запустите приложение, и Swagger сгенерирует документацию по доступным эндпоинтам. Она будет доступна по адресу: http://localhost:8080/swagger-ui/.

Варианты заданий

1. Добавление фильтрации и сортировки:

- Реализуйте возможность фильтрации сотрудников по имени или фамилии, а также добавьте возможность сортировки по различным полям.

2. Авторизация с использованием Spring Security:

- Добавьте базовую аутентификацию или авторизацию с использованием JWT для защиты API.

3. Пагинация:

- Реализуйте пагинацию для запроса списка сотрудников. Используйте объект PageRequest для ограничения и сортировки данных.

Контрольные вопросы

1. Что такое аннотация @RestController и как она отличается от обычного @Controller в Spring?

2. Какие HTTP-методы используются для создания, чтения, обновления и удаления ресурсов в RESTful веб-сервисах?

3. Как в Spring Boot реализована обработка запросов с помощью аннотации @RequestMapping?

4. Как можно использовать аннотацию @PathVariable и для чего она предназначена?

5. Что такое обработка ошибок в Spring, и как ее можно реализовать через аннотации и исключения?

Лабораторная работа № 17

Тема: JSON

Цель работы: Освоение работы с форматами данных JSON в приложениях Spring Boot. Изучение методов сериализации и десериализации объектов, а также интеграция с REST API для передачи и получения данных в формате JSON.

Порядок выполнения работы

1. Создание проекта с использованием Spring Initializr:

- Перейдите на [Spring Initializr](https://start.spring.io) и создайте новый проект с зависимостями:

- Spring Web

- Spring Boot DevTools для удобства разработки

- Скачайте и откройте проект в вашей IDE.

2. Создание сущности для работы с JSON:

- Создайте сущность, которую будете использовать для передачи данных в формате JSON. Например, сущность Product:

```
@Entity
@Table(name = "products")
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

```
    @Column(name = "name")
    private String name;
    @Column(name = "description")
    private String description;
    @Column(name = "price")
    private BigDecimal price;
    // Геттеры и сеттеры
}
```

3. Создание репозитория для работы с сущностью:

- Создайте репозиторий для работы с сущностью Product с использованием Spring Data JPA:

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

4. Создание REST-контроллера для работы с JSON:

- Создайте контроллер, который будет обрабатывать HTTP-запросы и работать с JSON данными. Используйте аннотацию @RestController для автоматической сериализации и десериализации объектов:

```
@RestController
@RequestMapping("/products")
public class ProductController {
    @Autowired
    private ProductRepository productRepository;
    // Получение всех продуктов
    @GetMapping
    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }
    // Получение продукта по ID
```

```
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        Product product = productRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Product not found"));
        return ResponseEntity.ok(product);
    }
    // Создание нового продукта
    @PostMapping
    public Product createProduct(@RequestBody Product product) {
        return productRepository.save(product);
    }
    // Обновление информации о продукте
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id, @RequestBody Product productDetails) {
        Product product = productRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Product not found"));
        product.setName(productDetails.getName());
        product.setDescription(productDetails.getDescription());
        product.setPrice(productDetails.getPrice());
        productRepository.save(product);
        return ResponseEntity.ok(product);
    }
    // Удаление продукта по ID
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
```

```
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Product not found"));
        productRepository.delete(product);
        return ResponseEntity.noContent().build();
    }
}
```

5. Тестирование API с использованием Postman:

- Запустите приложение с помощью команды mvn spring-boot:run или через вашу IDE.

- Откройте Postman для тестирования следующих запросов:

- GET запрос на /products — для получения всех продуктов.

- GET запрос на /products/{id} — для получения информации о продукте по ID.

- POST запрос на /products — для создания нового продукта.

- PUT запрос на /products/{id} — для обновления информации о продукте.

- DELETE запрос на /products/{id} — для удаления продукта.

6. Сериализация и десериализация JSON:

- Spring Boot автоматически использует Jackson для сериализации объектов в формат JSON и обратно. Например, объект Product будет преобразован в JSON при отправке через HTTP в ответе, а данные, отправленные через POST или PUT, будут автоматически преобразованы в объект Product благодаря аннотации @RequestBody.

- Пример запроса POST с передачей JSON:

```
{
  "name": "Laptop",
  "description": "High-performance laptop",
  "price": 1200.99
}
```

- Пример ответа GET с JSON:

```
{
  "id": 1,
  "name": "Laptop",
  "description": "High-performance laptop",
  "price": 1200.99
}
```

7. Добавление обработки ошибок:

- Реализуйте обработку ошибок, например, если продукт не найден. Создайте исключение ProductNotFoundException и используйте его в контроллере:

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class ProductNotFoundException extends RuntimeException {
    public ProductNotFoundException(String message) {
        super(message);
    }
}
```

- Используйте это исключение в методах контроллера:

```
@GetMapping("/{id}")
public ResponseEntity<Product> getProductById(@PathVariable Long id) {
    Product product = productRepository.findById(id)
            .orElseThrow(() -> new ProductNotFoundException("Product not found"));
    return ResponseEntity.ok(product);
}
```

8. Добавление поддержки JSON для списков:

- Также можно работать с массивами или списками объектов. Например, возвращать список продуктов в формате JSON, который будет автоматически сериализован Spring Boot:

```
@GetMapping
public List<Product> getAllProducts() {
    return productRepository.findAll();
}
```

Варианты заданий

1. Работа с вложенными объектами:

- Реализуйте поддержку вложенных объектов в JSON. Например, создайте сущность Order, которая будет содержать список продуктов.

2. Работа с коллекциями:

- Реализуйте возможность добавления нескольких продуктов в одном запросе через массив JSON объектов.

3. Использование других форматов данных:

- Добавьте поддержку других форматов для API, например, XML, и настройте возможность выбора формата через заголовки запроса.

Контрольные вопросы

1. Что такое сериализация и десериализация в контексте работы с JSON в Spring?
2. Как работает аннотация @RequestBody в Spring?
3. Как Spring Boot обрабатывает объекты в формате JSON при ответах на запросы?
4. Какие инструменты используются для работы с JSON в Spring Boot?
5. Как можно настроить обработку ошибок в Spring Boot?

Лабораторная работа № 18

Тема: Тестирование в проекте Spring Boot

Цель работы: Изучение основ тестирования в Spring Boot, включая написание юнит-тестов и интеграционных тестов для различных компонентов приложения, таких как контроллеры, сервисы и репозитории. Ознакомление с использованием фреймворков для тестирования, таких как JUnit и Mockito, а также с подходами к тестированию REST API.

Порядок выполнения работы

1. Создание тестовой конфигурации:

- Откройте существующий проект Spring Boot.

- Убедитесь, что в проекте присутствует зависимость для тестирования:

- spring-boot-starter-test (для интеграции с JUnit, Mockito и другими тестовыми библиотеками).

- Зависимость в pom.xml должна выглядеть следующим образом:

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

2. Создание юнит-тестов для сервисов:

- Для начала создайте тесты для сервисного слоя приложения. Юнит-тесты должны проверять отдельные методы бизнес-логики.

- Используйте аннотацию @MockBean для мокирования зависимостей сервисов.

- Пример теста для сервиса:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ProductServiceTest {
    @Autowired
```

```
    private ProductService productService;

    @MockBean
    private ProductRepository productRepository;
    @Test
    public void testCreateProduct() {
      Product product = new Product("Laptop", "High-performance laptop", BigDecimal.valueOf(1200.99));
Mockito.when(productRepository.save(Mockito.any(Product.class))).thenReturn(product);
      Product createdProduct = productService.createProduct(product);
      Assert.assertNotNull(createdProduct);
      Assert.assertEquals("Laptop", createdProduct.getName());
    }
  }
```

3. Интеграционные тесты для контроллеров:

- Создайте тесты для контроллеров, чтобы проверить работу API. Используйте аннотацию @WebMvcTest для тестирования только слоя контроллеров.

- Пример теста для контроллера:

```
  @RunWith(SpringRunner.class)
  @WebMvcTest(ProductController.class)
  public class ProductControllerTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    private ProductService productService;
    @Test
    public void testGetAllProducts() throws Exception {
      List<Product> products = Arrays.asList(
```

```
                new Product("Laptop", "High-performance laptop", BigDecimal.valueOf(1200.99)),
                new Product("Smartphone", "Latest model smartphone", BigDecimal.valueOf(799.99))
            );
        Mockito.when(productService.getAllProducts()).thenReturn(products);
            mockMvc.perform(MockMvcRequestBuilders.get("/products"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$[0].name").value("Laptop"))
                .andExpect(jsonPath("$[1].name").value("Smartphone"));
        }
    }
```

4. Тестирование REST API с использованием MockMvc:

- Используйте MockMvc для симуляции HTTP-запросов к API.
- Пример тестирования POST-запроса:

```
    @Test
    public void testCreateProduct() throws Exception {
        Product product = new Product("Tablet", "High-definition tablet", BigDecimal.valueOf(500.75));
        mockMvc.perform(MockMvcRequestBuilders.post("/products")
                .contentType(MediaType.APPLICATION_JSON)
                .content("{\"name\":\"Tablet\",\"description\":\"High-definition tablet\",\"price\":500.75}"))
                .andExpect(status().isCreated())
                .andExpect(jsonPath("$.name").value("Tablet"))
                .andExpect(jsonPath("$.description").value("High-definition tablet"));
    }
```

5. Тестирование обработки ошибок:

- Реализуйте тесты для обработки ошибок, например, для случая,

когда продукт не найден.

Используйте аннотацию @ResponseStatus для указания HTTP-статусов.

- Пример теста для обработки ошибки 404:

```
@Test
public void testGetProductNotFound() throws Exception {
    Mockito.when(productService.getProductById(1L)).thenThrow(new
ProductNotFoundException("Product not found"));
    mockMvc.perform(MockMvcRequestBuilders.get("/products/1"))
        .andExpect(status().isNotFound())
        .andExpect(jsonPath("$.message").value("Product not found"));
}
```

6. Запуск тестов и анализ результатов:

- Запустите все тесты с помощью вашей IDE или командой Maven:

```
mvn test
```

- Проверьте, что все тесты проходят успешно. В случае ошибок исследуйте трассировку и исправьте код.

7. Покрытие тестами:

- Убедитесь, что тестируются все основные пути выполнения кода: успешные запросы, обработка ошибок, а также возможные исключения.

- Используйте инструменты для анализа покрытия тестами, такие как JaCoCo, для того чтобы убедиться, что код покрыт тестами должным образом.

Варианты заданий

1. Тестирование с использованием данных из базы данных:

- Настройте тестирование с использованием встроенной базы данных H2 для интеграционных тестов. Протестируйте методы с реальными данными из базы.

2. Использование Mockito для имитации сервисов и репозиториев:

- Напишите тесты, используя Mockito для мокирования зависимостей в сервисах и репозиториях. Создайте несколько мока для тестирования

различных сценариев работы API.

3. Покрытие тестами сложных сценариев:

- Реализуйте тесты для сложных сценариев, например, создание нескольких сущностей в одном запросе или выполнение транзакций.

Контрольные вопросы

1. Что такое юнит-тестирование и как оно отличается от интеграционного тестирования?

2. Какие аннотации используются для тестирования контроллеров и сервисов в Spring Boot?

3. Как настроить MockMvc для тестирования REST API?

4. Какие преимущества и недостатки у использования Mockito для создания моков в тестах?

5. Какие типы тестов необходимо писать для Spring Boot приложения?

СПИСОК ЛИТЕРАТУРЫ

1. Наир, В. Предметно-ориентированное проектирование в Enterprise Java с помощью Jakarta EE, Eclipse MicroProfile, Spring Boot и программной среды Axon Framework / Наир В. , пер. с англ. А. В. Снастина. - М : ДМК Пресс. URL : https://www.studentlibrary.ru/book/ISBN9785970608722.html (дата обращения: 10.03.2025).
2. Баланов, А. Н. Комплексное руководство по разработке: от мобильных приложений до веб-технологий : учебное пособие для вузов / А. Н. Баланов. — СПб: Лань. URL: https://e.lanbook.com/book/394577 (дата обращения: 10.03.2025)
3. Васюткина, И. А. Разработка серверной части web-приложений на Java : учебное пособие/ И. А. Васюткина. - Новосибирск : НГТУ. URL : https://www.studentlibrary.ru/book/ISBN9785778243941.html (дата обращения: 10.03.2025).