МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования

«Гродненский государственный университет имени Янки Купалы»

Факультет математики и информатики

Кафедра фундаментальной и прикладной математики

Лыч Максим Александрович

**Разработка веб-приложения для создания списка покупок.**

Курсовой проект

студента 3 курса специальности

1-31 03 03-01 «Прикладная математика»

дневной формы получения образования

Научный руководитель:

Шпак Дарья Сергеевна,

доцент кафедры фундаментальной и

прикладной математики, кандидат

физико-математических наук.

Гродно 2024

РЕЗЮМЕ

Тема курсового проекта:

**«**Разработка веб-приложения для создания списка покупок**»**

Работа содержит: 55 страниц, 48 рисунков, 11 использованных источников литературы.

Ключевые слова: веб-приложение, список покупок, CRUD-операции frontend разработка, backend разработка

Предмет исследования – методы реализации full stack веб-приложения для создания списка покупок.

При выполнении курсового проекта были использованы средства:

Java, Spring framework, Vue.js, PostgreSQL, Vite, Lombok, Jackson, Maven, H2, Axios, Lombok.

Целью курсового проекта является разработка веб-приложения для создания cписка покупок.

Оглавление

[ВВЕДЕНИЕ 4](#_Toc185378977)

[1. Актуальность темы 4](#_Toc185378978)

[2. Тема и задача работы 4](#_Toc185378979)

[3. Используемые технологии и инструменты. 5](#_Toc185378980)

[Глава 1: Теоретические основы 7](#_Toc185378981)

[1. Веб-приложения и их архитектура 7](#_Toc185378982)

[2. Основы работы с Java и Spring Boot 7](#_Toc185378983)

[3. PostreSQL 7](#_Toc185378984)

[4. JWT 8](#_Toc185378985)

[5. Валидация данных 8](#_Toc185378986)

[6. Организация клиентской части с Vue.js 8](#_Toc185378987)

[7. Инструменты для работы с базой данных 9](#_Toc185378988)

[8. Преимущества Lombok, ModelMaper 9](#_Toc185378989)

[ГЛАВА 2. РАЗРАБОТКА BACKEND ЧАСТИ ВЕБ - ПРИЛОЖЕНИЯ 10](#_Toc185378990)

[1. Установка и настройка Spring Boot 10](#_Toc185378991)

[2. Настройка зависимостей 11](#_Toc185378992)

[3. Настройка базы данных 16](#_Toc185378993)

[4. Разработка структуры приложения 18](#_Toc185378994)

[5. Разработка конфигурационных классов 23](#_Toc185378995)

[6. Разработка репозиторных классов 27](#_Toc185378999)

[7. Разработка классов репозитория 28](#_Toc185379000)

[8. Разработка классов контроллера 33](#_Toc185379001)

[9. Дерево проекта 36](#_Toc185379002)

[ГЛАВА 3. РАЗРАБОТКА FRONTEND ЧАСТИ ВЕБ-ПРИЛОЖЕНИЯ 37](#_Toc185379003)

[1. Исследование структуры проекта 38](#_Toc185379004)

[2. Маршрутизация c использованием Vue Router 39](#_Toc185379005)

[3. Проектная структура и архитектура 41](#_Toc185379006)

[4. Api Client 52](#_Toc185379007)

[5. Интерфейс 53](#_Toc185379008)

[ЗАКЛЮЧЕНИЕ 55](#_Toc185379009)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 56](#_Toc185379010)

# 

# ВВЕДЕНИЕ

## Актуальность темы

Развитие цифровых технологий и интернет-сервисов значительно упростило выполнение повседневных задач, включая планирование покупок и управление списками товаров. В условиях современного ритма жизни людям важно иметь быстрый и удобный инструмент для создания, редактирования и хранения списка покупок. Использование веб-приложений для автоматизации данного процесса позволяет минимизировать временные затраты, упрощает организацию покупок и повышает их эффективность.

Актуальность разработки подобного приложения обусловлена потребностью в интуитивно понятных и многофункциональных платформах, которые обеспечивают взаимодействие пользователей с сервисом в реальном времени. Разработка приложения с применением **Java** и **Spring** для бэкенда и **Vue.js** для фронтенда позволяет создать надежное и масштабируемое решение с современным пользовательским интерфейсом, отвечающее требованиям удобства и безопасности.

## Тема и задача работы

Основной целью данной курсовой работы является разработка полнофункционального веб-приложения для создания и управления списками покупок, обеспечивающего удобный пользовательский опыт и эффективное взаимодействие с данными. Для достижения поставленной цели были определены следующие задачи:

- Разработать бэкенд часть приложения с использованием Java и Spring framework;

- Спроектировать и создать модель базы данных для хранения информации о списках покупок и товарах c использованием PostgreSQL;

- Разработать клиентскую часть приложения с использованием **Vue.js** для обеспечения интуитивно понятного интерфейса;

- Создать функционал для добавления, редактирования и удаления элементов в списке покупок;

- Реализовать возможность добавления новых товаров и выбора их из предложенного списка;

## Используемые технологии и инструменты.

1. Java 17 – основная платформа для разработки серверной части приложения.
2. Spring Boot – фреймворк для разработки и запуска веб-приложений на базе Spring.
   * Spring-Boot-Starter-Data-Jpa – для работы с базой данных с использованием ORM (Hibernate).
   * Spring-Boot-Starter-Web – для создания REST API и обработки HTTP-запросов.
   * Spring-Boot-Starter-Validation – для валидации данных на уровне контроллеров.
   * Spring-Boot-Starter-Security – для реализации безопасности и аутентификации.
   * Spring-Boot-Starter-Thymeleaf – для работы с серверной частью представления.
3. **PostgreSQL** – основная база данных для хранения информации о пользователях, списках покупок и товарах.
4. **H2 Database** – встроенная база данных для тестирования приложения на ранних этапах разработки.
5. **Liquibase** – инструмент для версионирования базы данных и управления миграциями.
6. **Lombok** – библиотека для уменьшения шаблонного кода (геттеры, сеттеры, конструкторы).
7. **ModelMapper** – библиотека для преобразования DTO и сущностей приложения.
8. **Jackson Databind** – для сериализации и десериализации JSON в Java-объекты и обратно.
9. **Vue.js** – фреймворк для разработки клиентской части приложения с поддержкой реактивного пользовательского интерфейса.
10. **Axios** – библиотека для выполнения HTTP-запросов между фронтендом и бэкендом.
11. **Maven** – инструмент для управления зависимостями и сборки проекта.

# Глава 1: Теоретические основы

## Веб-приложения и их архитектура

Веб-приложение – это программное обеспечение, клиентская часть которого выполняется в веб-браузере пользователя, а серверная часть отвечает за обработку данных и логику. Архитектура веб-приложений часто основывается на паттерне **MVC** (Model-View-Controller), который разделяет приложение на три компонента:

* **Модель** (Model) – отвечает за работу с данными и бизнес-логику приложения.
* **Представление** (View) – отображает данные пользователю.
* **Контроллер** (Controller) – управляет взаимодействием между моделью и представлением.

Применение архитектуры **REST** (Representational State Transfer) в веб-приложениях позволяет строить масштабируемые и эффективные API на основе HTTP-запросов для операций CRUD (Create, Read, Update, Delete).

## Основы работы с Java и Spring Boot

**Spring Boot** – это фреймворк, упрощающий создание веб-приложений на основе языка **Java**. Он предоставляет готовые решения для работы с базами данных, обеспечения безопасности, валидации и разработки REST API. Основные модули:

* **Spring Boot Starter Web** – для создания RESTful веб-сервисов.
* **Spring Boot Starter Data JPA** – для работы с базой данных через **JPA** (Java Persistence API).
* **Spring Boot Starter Security** – для реализации безопасности приложения и аутентификации.

## PostreSQL

**PostgreSQL** – это реляционная система управления базами данных (СУБД), которая обеспечивает надежное хранение данных, поддержку транзакций и широкие возможности масштабирования. Взаимодействие с базой данных в **Spring Boot** происходит с использованием **Spring Data JPA**, который позволяет работать с данными через репозитории и сущности. Основные операции включают:

* Создание и изменение таблиц (DDL).
* Добавление, чтение, обновление и удаление данных (DML).
* Использование SQL-запросов и JPQL для манипуляции данными.

Для упрощения миграции и версионирования базы данных применяется **Liquibase**.

## JWT

Аутентификация и авторизация являются ключевыми элементами безопасности веб-приложений. **JWT** (JSON Web Token) – это открытый стандарт для передачи информации между клиентом и сервером в зашифрованном виде. JWT состоит из трех частей:

* **Header** (заголовок) – тип токена и алгоритм подписи.
* **Payload** (полезная нагрузка) – содержит данные пользователя и метаданные.
* **Signature** (подпись) – обеспечивает целостность и подлинность данных.

JWT используется для безопасного хранения токенов доступа и передачи их между клиентом и сервером.

## Валидация данных

Валидация данных в веб-приложениях позволяет обеспечивать их корректность и защищает систему от потенциальных атак. В **Spring Boot** для валидации данных используется модуль **spring-boot-starter-validation**, основанный на **Hibernate Validator**. Основные аннотации:

* @NotNull – поле не должно быть пустым.
* @Size – ограничение на длину строки.
* @Email – проверка формата электронной почты.
* @Pattern – проверка данных на соответствие регулярному выражению.

Валидация позволяет проверять данные на уровне контроллеров перед их сохранением в базу данных.

## Организация клиентской части с Vue.js

**Vue.js** – это прогрессивный фреймворк для создания пользовательских интерфейсов. Он позволяет эффективно разрабатывать динамические одностраничные приложения (SPA). Преимущества Vue.js включают:

* **Реактивность данных** – изменения данных автоматически обновляют интерфейс.
* **Компонентный подход** – возможность разбивать интерфейс на переиспользуемые компоненты.
* **Vue Router** – для организации маршрутизации внутри приложения.
* **Axios** – для выполнения HTTP-запросов к серверу.

Интеграция Vue.js с REST API позволяет реализовать современный и удобный пользовательский интерфейс.

## Инструменты для работы с базой данных

Для управления структурой и миграциями базы данных в проекте используется **Liquibase**. Он позволяет:

* Создавать версии базы данных и откатывать изменения.
* Управлять схемой данных с помощью XML, YAML или SQL-скриптов.
* Поддерживать совместимость структуры данных на разных этапах разработки.

## Преимущества Lombok, ModelMaper

* **Lombok** – библиотека для генерации стандартного кода Java (геттеры, сеттеры, конструкторы, методы toString, equals). Использование Lombok позволяет минимизировать шаблонный код и сделать проект более чистым.
* **ModelMapper** – инструмент для преобразования данных между **DTO** (Data Transfer Object) и сущностями приложения, что упрощает передачу данных между слоями.

# ГЛАВА 2. РАЗРАБОТКА BACKEND ЧАСТИ ВЕБ - ПРИЛОЖЕНИЯ

## 

## 1. Установка и настройка Spring Boot

Установка и настройка **Spring Boot** является первым этапом разработки backend-части веб-приложения. Для создания проекта можно использовать **Spring Initializr** или любой популярный IDE с поддержкой Spring Boot:

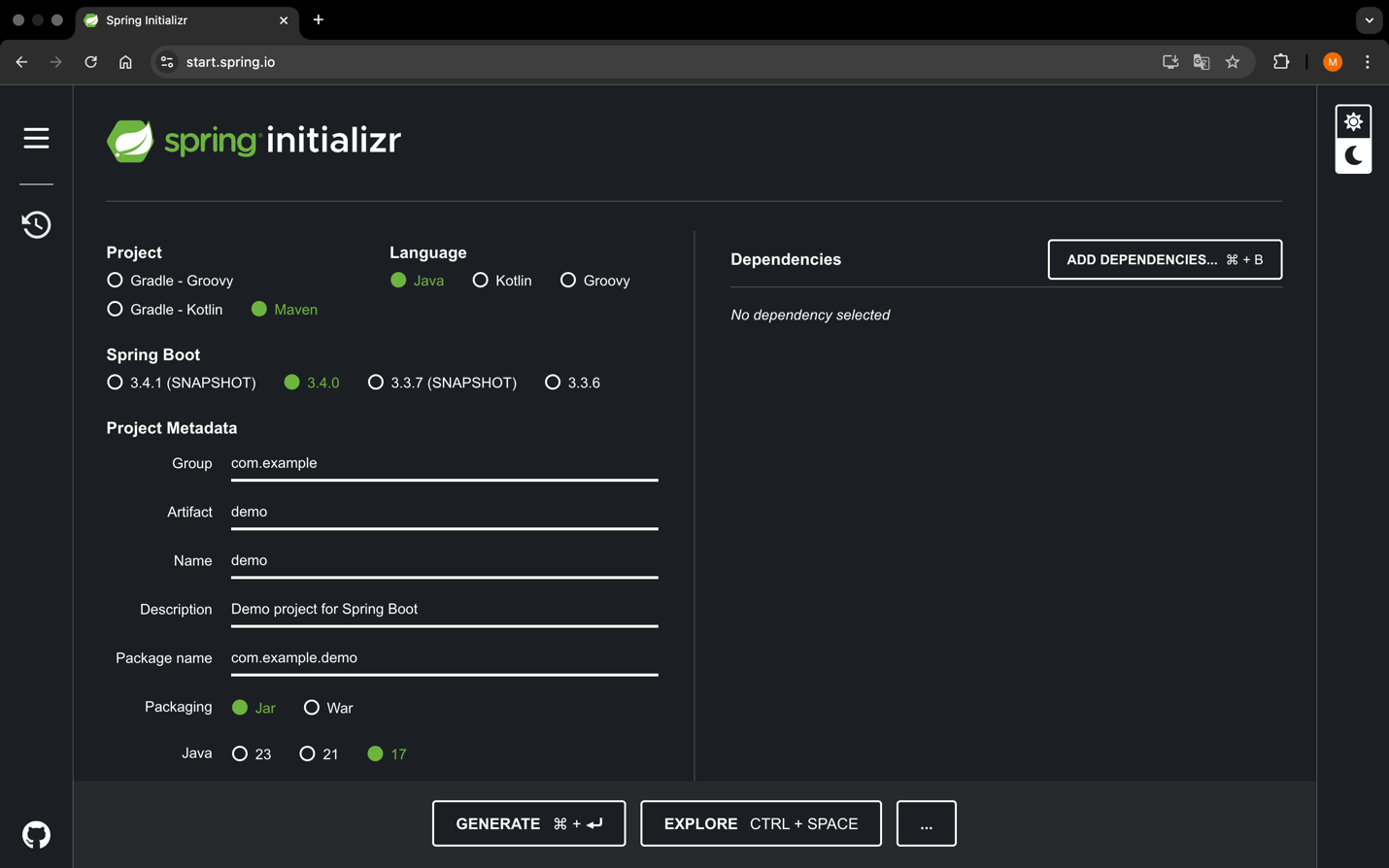


Рисунок – 2.1.1 – инструмент **Spring Initializr для настройки приложения**

Так же для начала работы необходимо настроить класс для компиляции приложения. Класс использует аннотацию @SpringBootApplication для автоматическойнастройки приложения и запускает сервер на стандартном порту **8080**:

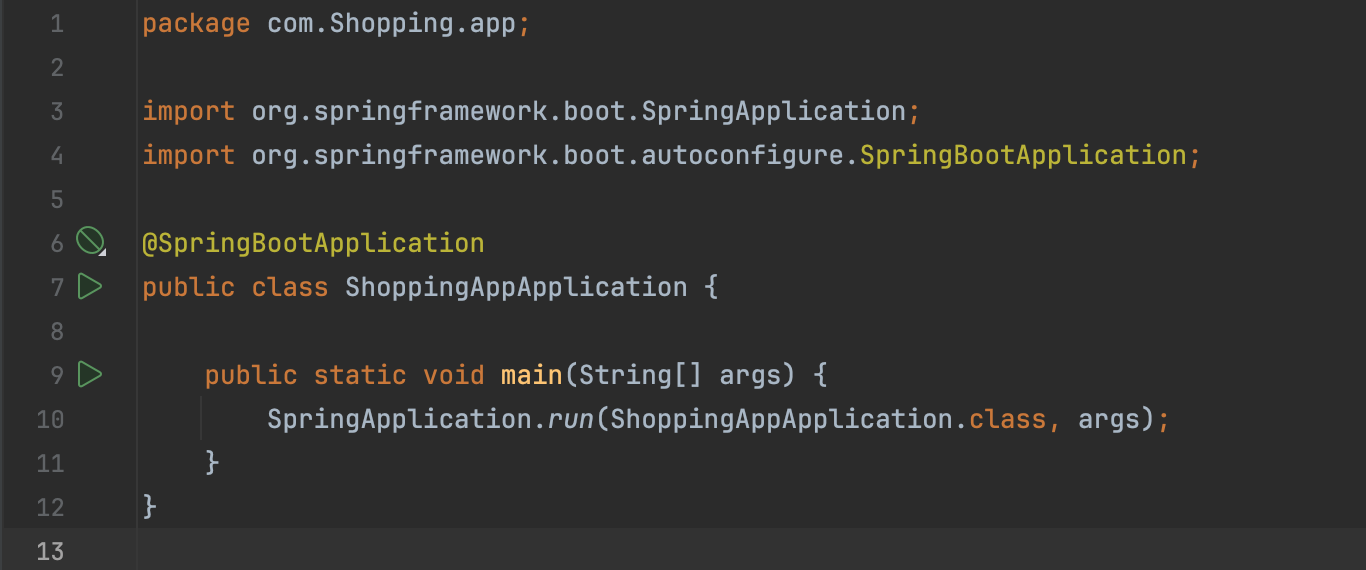


Рисунок – 2.1.2 – Фрагмент кода для запуска Spring Boot приложения.

## 2. Настройка зависимостей

Для работы с **Spring Boot** и другими необходимыми библиотеками используется **Maven** в качестве системы управления зависимостями. В данном разделе предоставлен файл pom.xml с основными зависимостями, которые понадобятся для разработки backend-части приложения.

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>3.4.0</version>  
 <relativePath/>  
 </parent>  
 <groupId>com.Shopping.app</groupId>  
 <artifactId>Shopping-app</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <packaging>jar</packaging>  
  
 <properties>  
 <java.version>17</java.version>  
 </properties>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-validation</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.liquibase</groupId>  
 <artifactId>liquibase-core</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-thymeleaf</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.postgresql</groupId>  
 <artifactId>postgresql</artifactId>  
 <version>42.5.0</version>  
 </dependency>  
 <dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>runtime</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.18.34</version>  
 <scope>provided</scope>  
 </dependency>  
 <dependency>  
 <groupId>com.fasterxml.jackson.core</groupId>  
 <artifactId>jackson-databind</artifactId>  
 <version>2.15.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.modelmapper</groupId>  
 <artifactId>modelmapper</artifactId>  
 <version>3.1.0</version>  
 </dependency>  
 </dependencies>  
  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
  
</project>

Рисунок – 2.2.1 – Содержимое pom.xml файла.

1. **Spring Boot Starter Data JPA**  
   **Назначение**: Позволяет работать с базами данных через ORM (Hibernate). Обеспечивает интеграцию Spring с JPA.

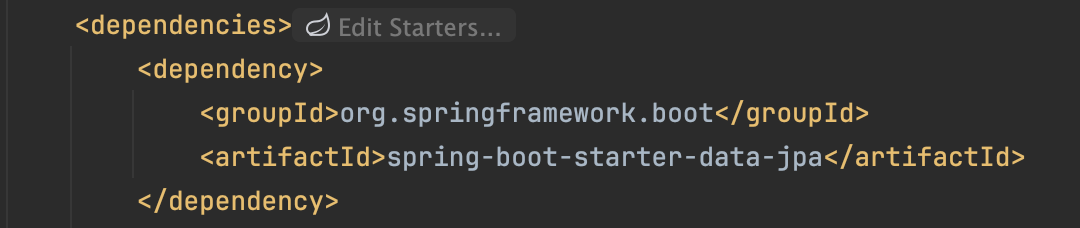


Рисунок – 2.2.2 – Внедрение зависимости **Spring Boot Starter Data JPA.**

1. **Spring Boot Starter Web**  
   **Назначение**: Используется для создания REST API и обработки HTTP-запросов.

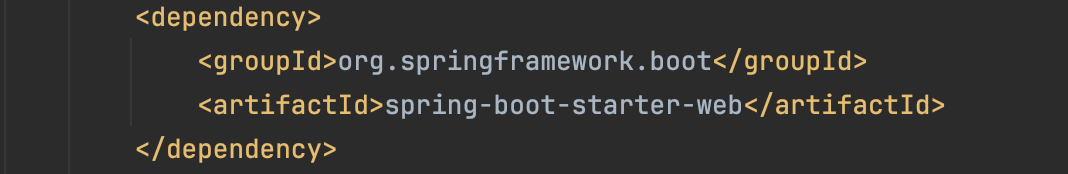


Рисунок – 2.2.3 – Внедрение зависимости **Spring Boot Starter Web.**

1. **Spring Boot Starter Validation**  
   **Назначение**: Добавляет валидацию данных с использованием аннотаций @NotNull, @Size, @Valid и других.

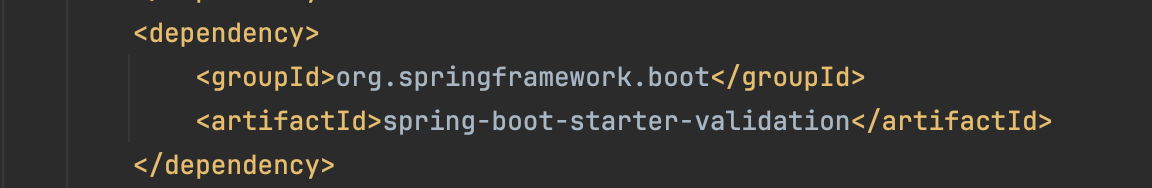
****

Рисунок – 2.2.4 – Внедрение зависимости **Spring Boot Starter Validation.**

1. **Liquibase Core**  
   **Назначение**: Управляет миграциями базы данных. Позволяет отслеживать изменения схемы БД и применять их.

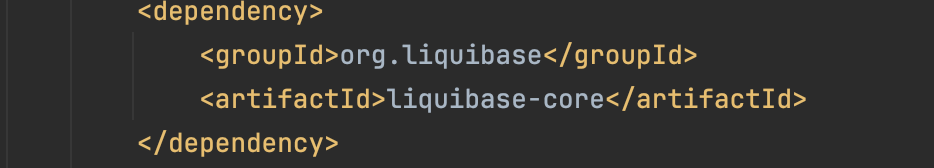


Рисунок – 2.2.5 – Внедрение зависимости **Liquibase Core.**

1. **Spring Boot Starter Thymeleaf**  
   **Назначение**: Интеграция Thymeleaf для разработки серверных HTML-шаблонов.

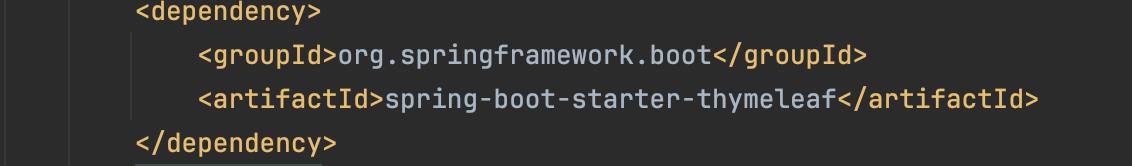


Рисунок – 2.2.6 – Внедрение зависимости **Liquibase Core.**

1. **Spring Boot Starter Security**  
   **Назначение**: Обеспечивает безопасность приложения, включая аутентификацию и авторизацию.

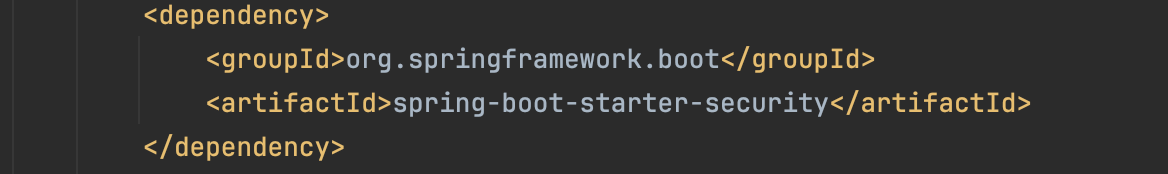


Рисунок – 2.2.7 – Внедрение зависимости **Spring Boot Starter Security.**

1. **PostgreSQL**  
   **Назначение**: Драйвер для подключения и работы с базой данных PostgreSQL.

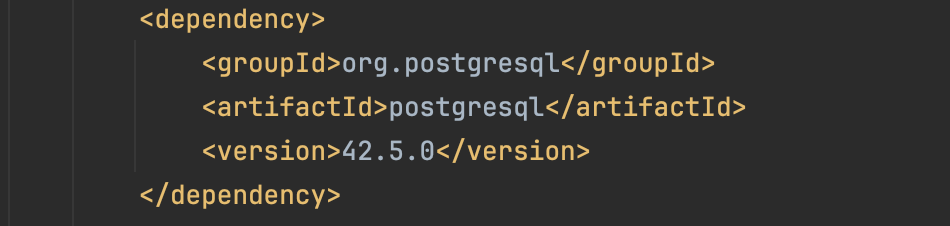


Рисунок – 2.2.8 – Внедрение зависимости **PostgreSQL.**

1. **H2 Database**  
   **Назначение**: Встроенная база данных для тестирования и разработки.

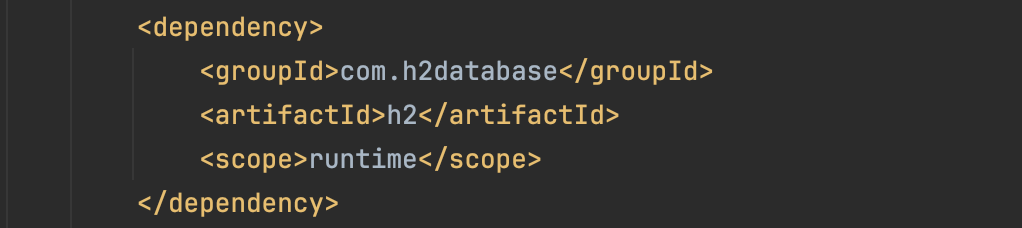


Рисунок – 2.2.9 – Внедрение зависимости **H2 Database.**

1. **Lombok**  
   **Назначение**: Генерирует геттеры, сеттеры, конструкторы и другие методы, сокращая количество кода.

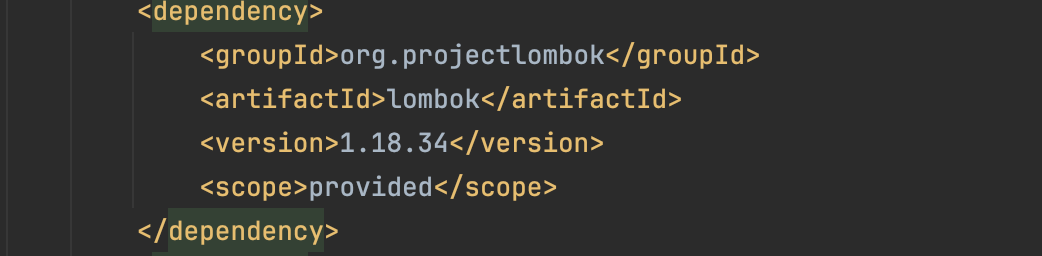


Рисунок – 2.2.10 – Внедрение зависимости **Lombok.**

1. **Jackson Databind**  
   **Назначение**: Позволяет сериализовать и десериализовать объекты Java в JSON и обратно.



Рисунок – 2.2.11 – Внедрение зависимости **Jackson Databind.**

1. **ModelMapper**  
   **Назначение**: Облегчает преобразование объектов между слоями приложения.

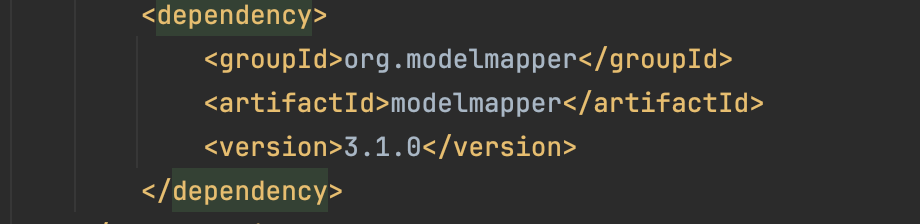


Рисунок – 2.2.12 – Внедрение зависимости **ModelMapper.**

1. **Настройка Maven плагина**

Для запуска приложения используется **spring-boot-maven-plugin**:



Рисунок – 2.2.13 – Настройка плагина Maven**.**

## 3. Настройка базы данных

Для использования PostgreSQL или H2 необходимо настроить подключение к базе данных в файле application.properties.

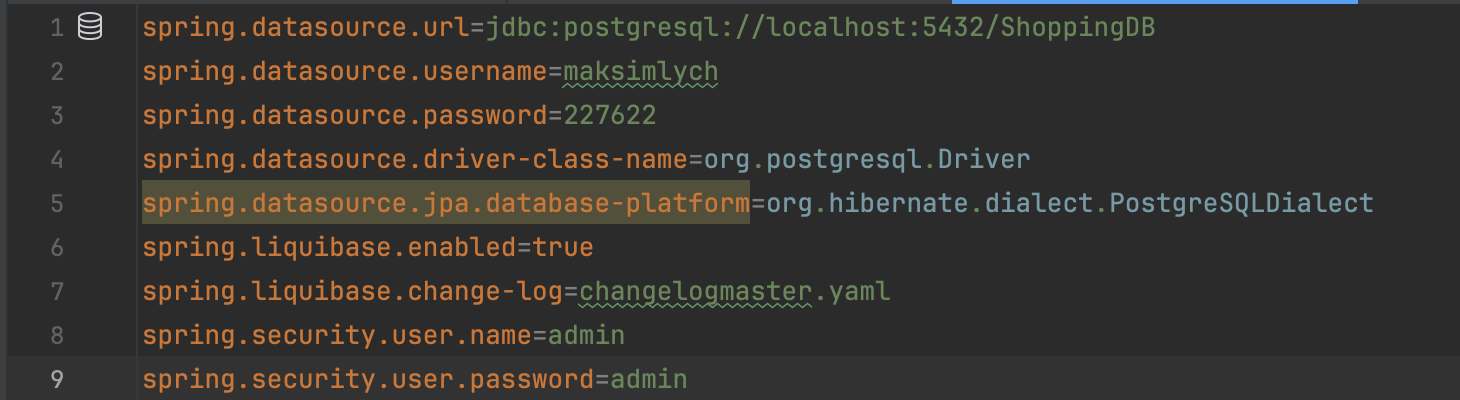


Рисунок – 2.3.1 – Содержимое файла настройки application.properties**.**

**Для** управления миграциями базы данных используем файл настройки для Liquibase change-master.yaml содержащий структуру базы данных.

databaseChangeLog:  
 - changeSet:  
 id: 1  
 author: Lych  
 changes:  
 - createTable:  
 tableName: "shopping\_list"  
 columns:  
 - column:  
 name: "id"  
 type: "bigserial"  
 constraints:  
 primaryKey: true  
 - column:  
 name: "name"  
 type: "varchar(100)"  
 constraints:  
 nullable: false  
 - column:  
 name: "created\_at"  
 type: "timestamp"  
 constraints:  
 nullable: false  
 - column:  
 name: "updated\_at"  
 type: "timestamp"  
 constraints:  
 nullable: true  
  
 - changeSet:  
 id: 2  
 author: Lych  
 changes:  
 - createTable:  
 tableName: "item"  
 columns:  
 - column:  
 name: "id"  
 type: "bigserial"  
 constraints:  
 primaryKey: true  
 - column:  
 name: "name"  
 type: "varchar(100)"  
 constraints:  
 nullable: false  
 - column:  
 name: "description"  
 type: "varchar(500)"  
 constraints:  
 nullable: true  
 - column:  
 name: "price"  
 type: "decimal(10, 2)"  
 constraints:  
 nullable: false  
 - column:  
 name: "created\_at"  
 type: "timestamp"  
 constraints:  
 nullable: false  
 - column:  
 name: "updated\_at"  
 type: "timestamp"  
 constraints:  
 nullable: true  
  
 - changeSet:  
 id: 3  
 author: Lych  
 changes:  
 - createTable:  
 tableName: "shopping\_list\_item"  
 columns:  
 - column:  
 name: "id"  
 type: "bigserial"  
 constraints:  
 primaryKey: true  
 - column:  
 name: "shopping\_list\_id"  
 type: "bigint"  
 constraints:  
 nullable: false  
 - column:  
 name: "item\_id"  
 type: "bigint"  
 constraints:  
 nullable: false  
 - column:  
 name: "quantity"  
 type: "integer"  
 constraints:  
 nullable: false  
  
 - addForeignKeyConstraint:  
 baseTableName: "shopping\_list\_item"  
 baseColumnNames: "shopping\_list\_id"  
 referencedTableName: "shopping\_list"  
 referencedColumnNames: "id"  
 constraintName: "fk\_shopping\_list\_item\_list"  
  
 - addForeignKeyConstraint:  
 baseTableName: "shopping\_list\_item"  
 baseColumnNames: "item\_id"  
 referencedTableName: "item"  
 referencedColumnNames: "id"  
 constraintName: "fk\_shopping\_list\_item\_item"

Рисунок – 2.3.2 – Содержимое файла настройки changelog-master.yaml**.**

**В данном файле описана структура и связи базы данных.**

## 4. Разработка структуры приложения

Нужно создать классы-сущности, которые будут представлять таблицы в базе данных.

package com.Shopping.app.entity;  
  
import jakarta.persistence.\*;  
import lombok.Getter;  
import lombok.Setter;  
  
import java.math.BigDecimal;  
import java.time.LocalDateTime;  
  
@Entity  
@Getter  
@Setter  
public class Item {  
 @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;  
  
 @Column(name = "created\_at")  
 private LocalDateTime createdAt;  
  
 @Column(name = "updated\_at")  
 private LocalDateTime updatedAt;  
  
 @PrePersist  
 public void prePersist() {  
 if (createdAt == null) {  
 createdAt = LocalDateTime.*now*();  
 }  
 updatedAt = createdAt;  
 }  
}

Рисунок – 2.4.1 – Содержимое класса Item**.**

package com.Shopping.app.entity;  
  
import jakarta.persistence.\*;  
import lombok.Getter;  
import lombok.Setter;  
  
import java.time.LocalDateTime;  
import java.util.List;  
  
@Entity  
@Getter  
@Setter  
public class ShoppingList {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
  
 @Column(nullable = false)  
 private String name;  
  
 @Column(name = "created\_at", nullable = false)  
 private LocalDateTime createdAt;  
  
 @Column(name = "updated\_at")  
 private LocalDateTime updatedAt;  
  
 @OneToMany(mappedBy = "shoppingList", cascade = CascadeType.*ALL*, orphanRemoval = true)  
 private List<ShoppingListItem> items;  
  
 @PrePersist  
 public void prePersist() {  
 createdAt = LocalDateTime.*now*();  
 updatedAt = createdAt;  
 }  
}

Рисунок – 2.4.2 – Содержимое класса Shopping list**.**

package com.Shopping.app.entity;  
  
import jakarta.persistence.\*;  
import lombok.Getter;  
import lombok.Setter;  
  
@Entity  
@Getter  
@Setter  
public class ShoppingListItem {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
  
 @ManyToOne  
 @JoinColumn(name = "shopping\_list\_id", nullable = false)  
 private ShoppingList shoppingList;  
  
 @ManyToOne  
 @JoinColumn(name = "item\_id", referencedColumnName = "id", nullable = false)  
 private Item item;  
  
 @Column(nullable = false)  
 private Integer quantity;  
}

Рисунок – 2.4.3 – Содержимое класса Shopping list item**.**

Эти три класса — Item, ShoppingList, и ShoppingListItem — представляют основные сущности для работы с покупками в приложении. Класс Item описывает товары, которые могут быть добавлены в списки покупок, а ShoppingList представляет сами списки покупок. Связь между ними осуществляется через промежуточную сущность ShoppingListItem, которая содержит информацию о количестве каждого товара в списке.

Вместе эти классы образуют структуру данных, позволяющую эффективно управлять списками покупок и товарами в базе данных. Использование аннотаций JPA, таких как @OneToMany, @ManyToOne, и @PrePersist, упрощает управление связями между сущностями и автоматизирует процессы, такие как установка временных меток при создании записей.

Эта модель данных позволяет легко расширять функциональность приложения, добавляя новые поля или сущности, и поддерживает все необходимые операции для создания, редактирования и удаления списков покупок и товаров.

**2. Создание DTO классов:**

DTO (Data Transfer Object) классы представляют собой объекты, используемые для передачи данных между слоями приложения, обычно между сервисами и контроллерами, а также для обмена данными между клиентом и сервером.

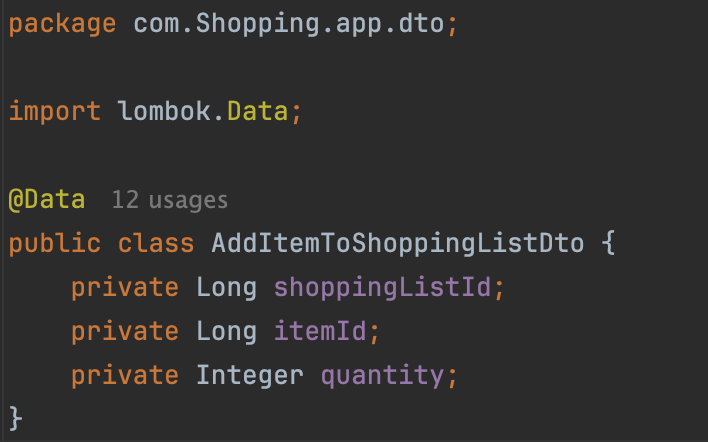
****

Рисунок – 2.4.4 – Содержимое класса AddItemToShoppongListDTO.

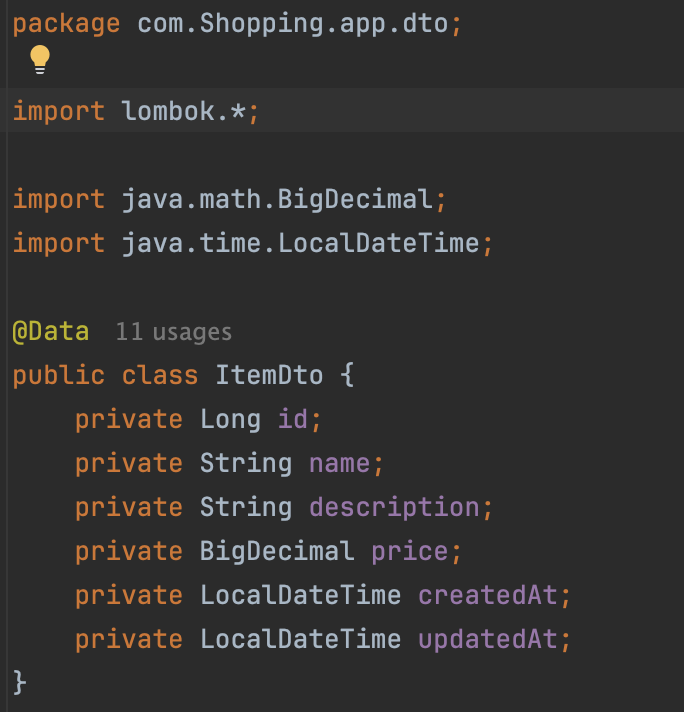
****

Рисунок – 2.4.5 – Содержимое класса ItemDTO.

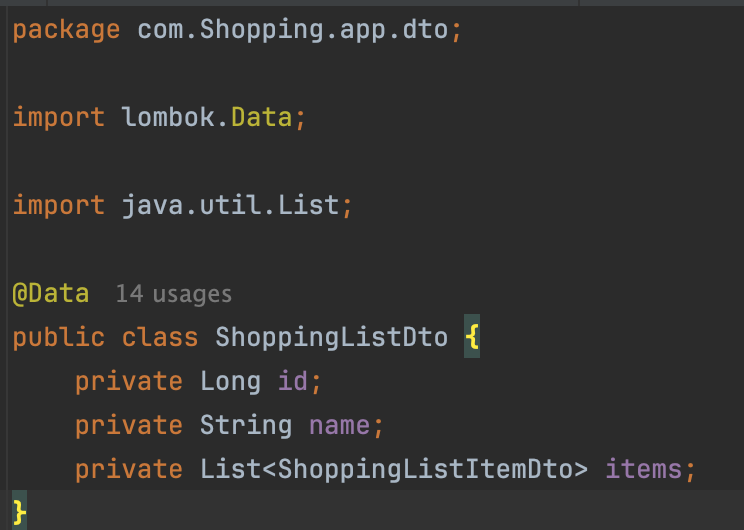
****

Рисунок – 2.4.6 – Содержимое класса ItemDTO.

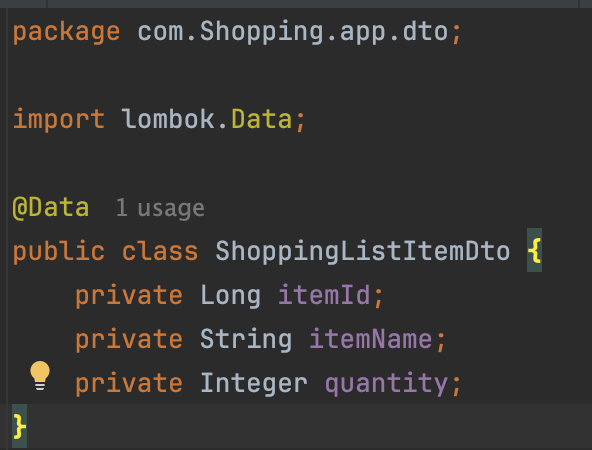


Рисунок – 2.4.7 – Содержимое класса ShoppingListDTO.

В данном проекте каждый DTO класс предназначен для упрощенной передачи данных, избегая излишних зависимостей от внутренних сущностей и логики приложения.

* **AddItemToShoppingListDto**: Используется для добавления товара в список покупок, включая идентификаторы списка, товара и количество товара.
* **ItemDto**: Представляет товар, включая его идентификатор, название, описание, цену и временные метки создания и обновления.
* **ShoppingListDto**: Описание списка покупок, включая его идентификатор, название и список элементов (ShoppingListItemDto), которые относятся к этому списку.
* **ShoppingListItemDto**: Описание элемента списка покупок, которое включает идентификатор товара, название товара и количество товара в списке.

DTO классы помогают разделить внутреннюю логику приложения от внешнего интерфейса, обеспечивая более чистую архитектуру и упрощая обработку данных между слоями.

## 5. Разработка конфигурационных классов

## Конфигурационные классы в проекте выполняют важную роль в настройке различных аспектов приложения, таких как маппинг объектов, безопасность и настройка CORS.

**ModelMapperConfig**

Этот класс конфигурирует ModelMapper, который используется для преобразования данных между сущностями (Entity) и DTO (Data Transfer Objects).

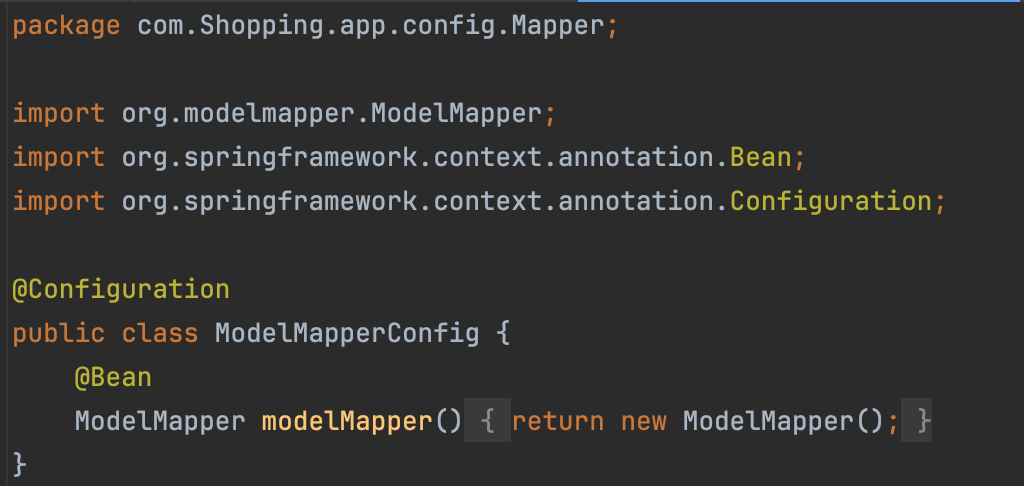


Рисунок – 2.5.1 – Содержимое класса **ModelMapperConfig**.

* Аннотация @Configuration указывает, что это класс конфигурации Spring.
* Метод modelMapper() помечен аннотацией @Bean, что позволяет Spring автоматически создать и управлять экземпляром ModelMapper.
* ModelMapper используется для упрощения преобразования данных между объектами разных типов, например, между сущностями и DTO.

**Security Config**

Этот класс конфигурирует базовую безопасность приложения, используя Spring Security. В данном случае создается простая настройка безопасности с разрешением доступа ко всем запросам и с дефолтным пользователем.

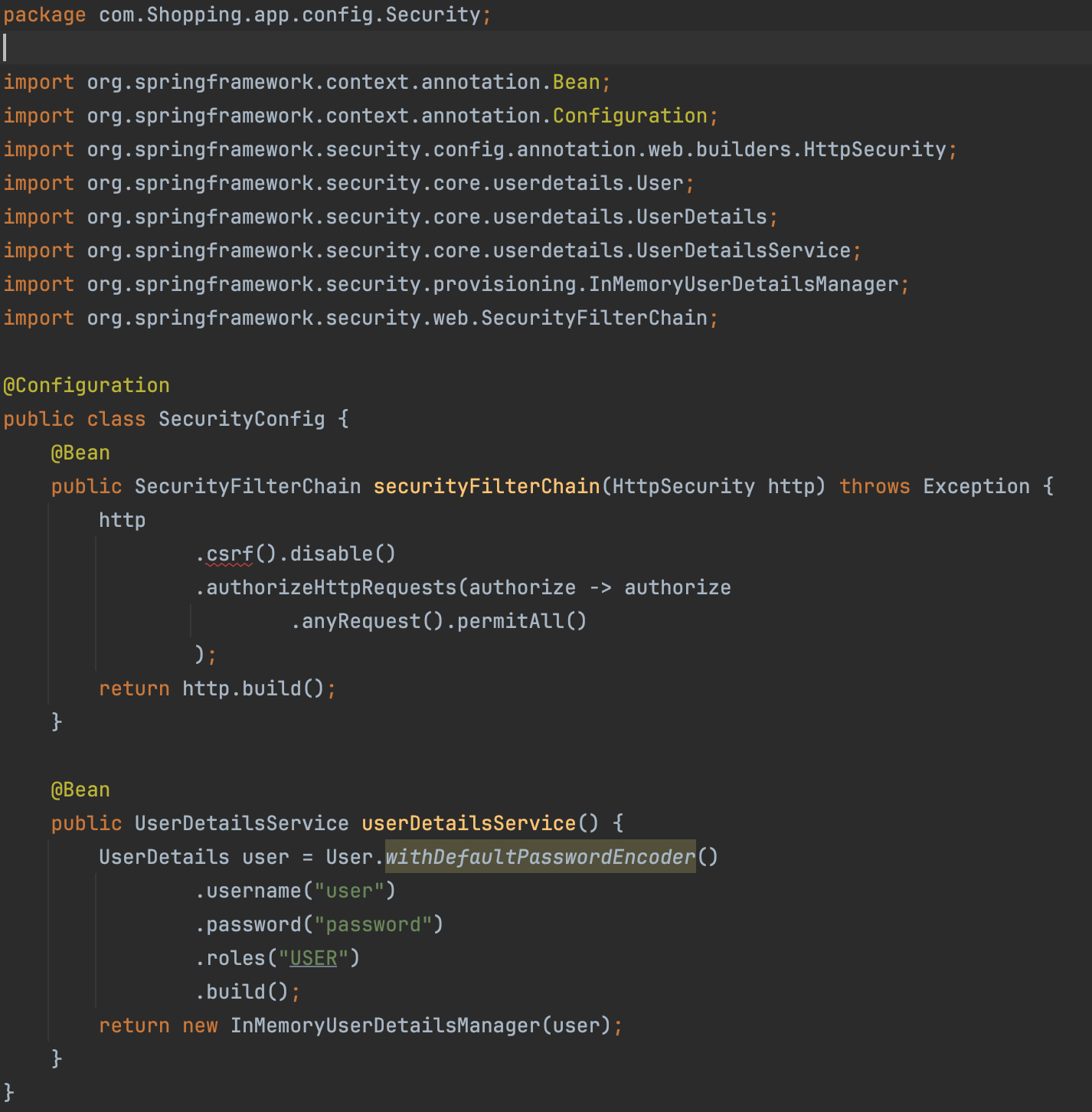


Рисунок – 2.5.2 – Содержимое класса **Security Config**.

* Метод securityFilterChain настраивает фильтры безопасности HTTP. В данном примере отключается защита от CSRF (что актуально для REST API) и разрешается доступ ко всем маршрутам (anyRequest().permitAll()).
* Метод userDetailsService создает пользователя с фиксированным логином и паролем, который используется для аутентификации в приложении. В данном случае пользователь создается в памяти с помощью InMemoryUserDetailsManager.

WebConfig

Этот класс настраивает CORS (Cross-Origin Resource Sharing), позволяя приложениям на другом домене отправлять запросы к вашему серверу. Это важно, если фронтенд и бэкенд находятся на разных доменах.

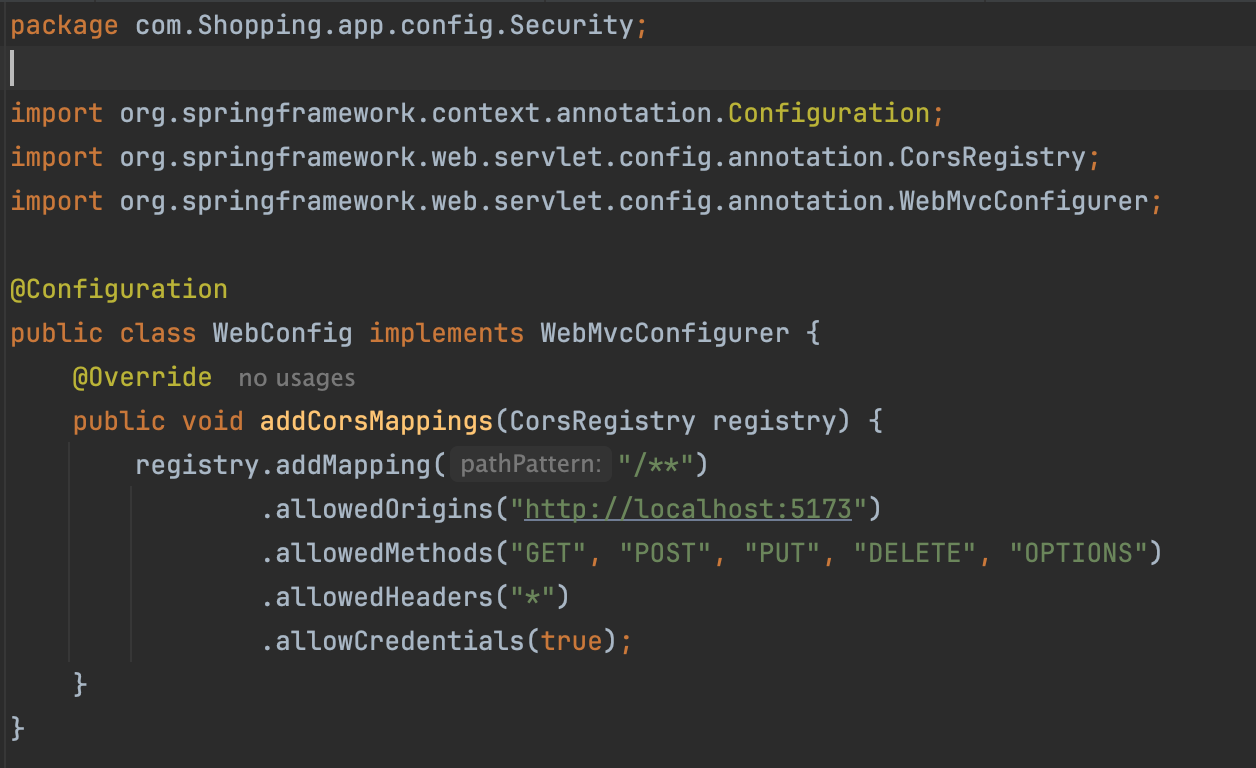


Рисунок – 2.5.3 – Содержимое класса **Web Config**.

* Аннотация @Configuration указывает, что это класс конфигурации.
* Метод addCorsMappings настраивает правила CORS, позволяя доступ с указанного домена (http://localhost:5173) и разрешая определенные HTTP методы. Это необходимо для корректного взаимодействия фронтенда и бэкенда, особенно когда они расположены на разных хостах или портах.

**Общее описание конфигураций:**

1. **ModelMapperConfig**: Конфигурация для упрощения маппинга между сущностями и DTO.
2. **SecurityConfig**: Настройка безопасности с базовой конфигурацией, позволяющей доступ ко всем запросам и создающая стандартного пользователя.
3. **WebConfig**: Настройка CORS для разрешения взаимодействия между фронтендом и бэкендом, расположенными на разных доменах.

Эти конфигурации помогают упростить работу с данными, повысить безопасность и обеспечить взаимодействие между различными частями приложения.

## 6. Разработка репозиторных классов

Репозитории в Spring Data JPA позволяют работать с сущностями, не прибегая к написанию сложных SQL-запросов, предоставляя доступ к стандартным операциям (создание, чтение, обновление и удаление данных). В нашем приложении для списков покупок, мы будем использовать репозитории для работы с такими сущностями, как Item, ShoppingList и ShoppingListItem.

**ItemRepository**

Этот репозиторий предоставляет доступ к данным о товарах. Он использует стандартный функционал JpaRepository для операций CRUD с сущностью Item, которая представляет товар в вашем приложении (с полями, такими как название, описание, цена и дата создания).

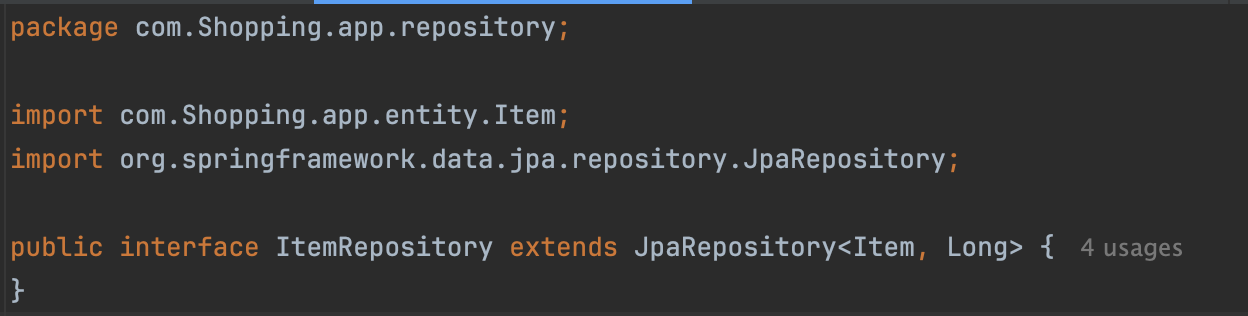


Рисунок – 2.6.1 – Содержимое класса **Item Repository**.

**ShoppingListItemRepository**

Репозиторий для работы с сущностью ShoppingListItem, которая является связующим звеном между ShoppingList и Item. Этот репозиторий расширяет функциональность JpaRepository и включает дополнительный метод для поиска элемента списка покупок по идентификаторам списка и товара.

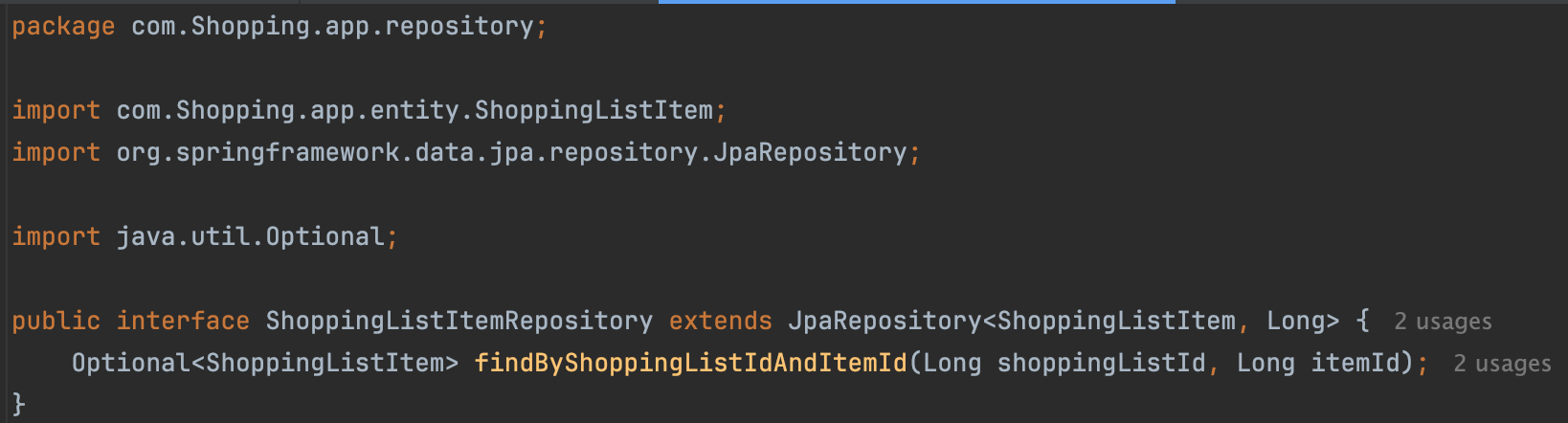


Рисунок – 2.6.2 – Содержимое класса **ShoppingListItemRepository**.

**ShoppingListRepository**

Репозиторий для работы с сущностью ShoppingList, которая представляет сам список покупок. Этот репозиторий предоставляет стандартный набор CRUD операций для работы со списками покупок.

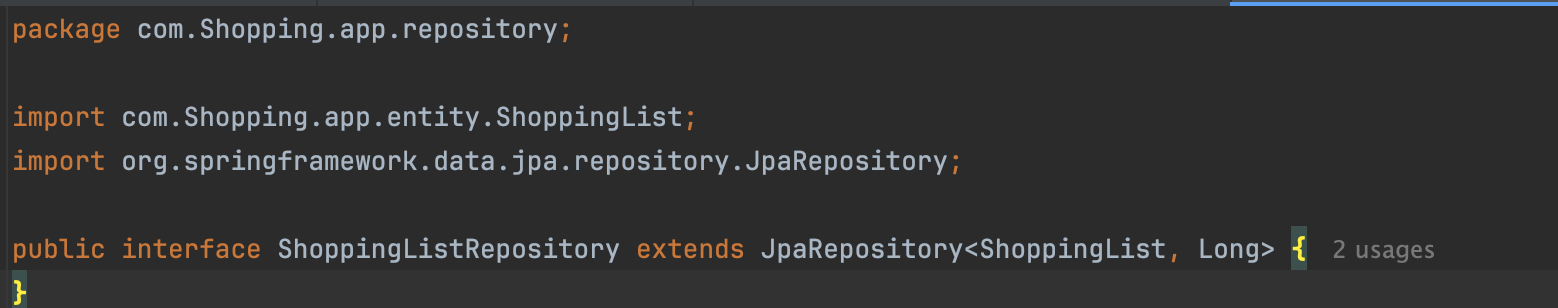


Рисунок – 2.6.3 – Содержимое класса **ShoppingListRepository**.

1. **Упрощение работы с базой данных**: Репозитории обеспечивают абстракцию над прямым взаимодействием с базой данных. Нам не нужно писать сложные SQL-запросы для выполнения базовых операций. Все основные CRUD операции выполняются с помощью простых вызовов методов.
2. **Согласованность с принципами SOLID**: Каждый репозиторий отвечает только за работу с одной сущностью, что соответствует принципу единой ответственности. Это помогает сохранять код модульным и легким для тестирования.
3. **Расширяемость**: Когда проект будет развиваться, репозитории легко расширяются. Можно добавить дополнительные методы для более сложных запросов, таких как фильтрация товаров по цене или дата создания, или же поиск списков покупок по пользователю.

Использование репозиториев в нашем проекте для списков покупок упрощает взаимодействие с базой данных и помогает сосредоточиться на бизнес-логике, а не на реализации запросов к базе данных. Благодаря Spring Data JPA, мы можем сосредоточиться на расширении функционала приложения, не беспокоясь о реализации низкоуровневых операций с базой данных.

## 7. Разработка классов репозитория

В приложении для списков покупок, сервисы выполняют важную роль, обеспечивая реализацию бизнес-логики приложения. Они используют репозитории для выполнения операций с данными и преобразуют их в соответствующие DTO (Data Transfer Objects) для передачи информации между слоями приложения.

**ItemService** — интерфейс, который определяет операции для работы с товарами:

* getAllItems(): Получение всех товаров.
* getItemById(Long id): Получение товара по его идентификатору.
* createItem(Item item): Создание нового товара.
* deleteItem(Long id): Удаление товара по его идентификатору.

Реализация этого интерфейса в классе ItemServiceImpl использует репозиторий ItemRepository для выполнения этих операций. DTO, такие как ItemDto, используются для преобразования сущностей в данные, которые отправляются на клиент.

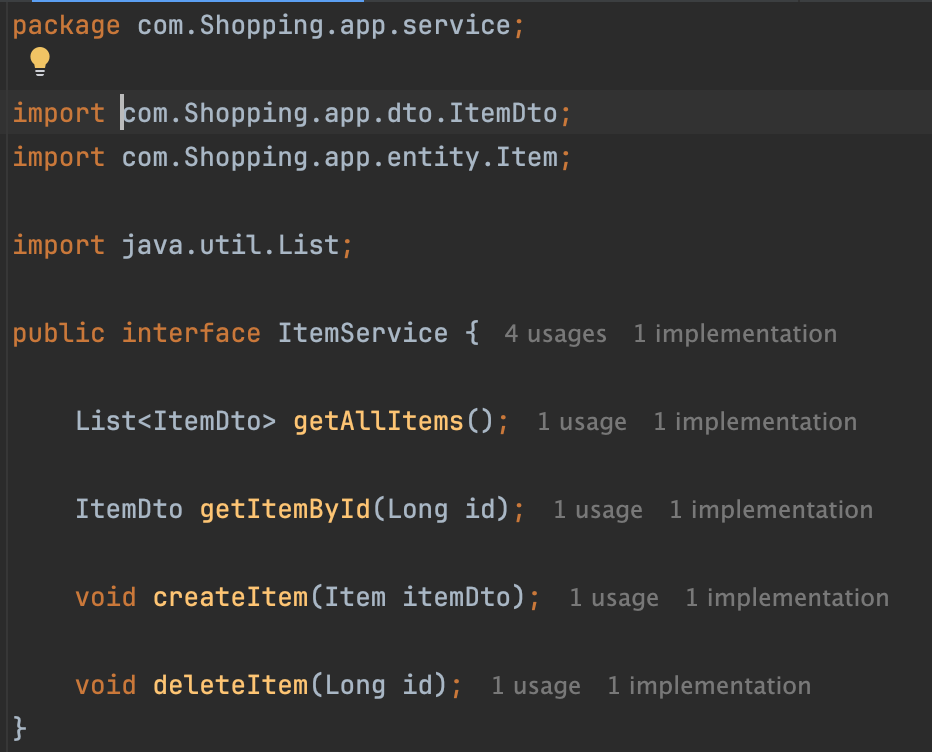


Рисунок – 2.7.1 – Содержимое интерфейса **ItemService.**

package com.Shopping.app.service.impl;  
  
import com.Shopping.app.dto.ItemDto;  
import com.Shopping.app.entity.Item;  
import com.Shopping.app.repository.ItemRepository;  
import com.Shopping.app.service.ItemService;  
import jakarta.persistence.EntityNotFoundException;  
import lombok.RequiredArgsConstructor;  
import org.modelmapper.ModelMapper;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
import java.util.stream.Collectors;  
  
@Service  
@RequiredArgsConstructor  
public class ItemServiceImpl implements ItemService {  
  
 private final ItemRepository itemRepository;  
 private final ModelMapper modelMapper;  
  
 @Override  
 public List<ItemDto> getAllItems() {  
 return itemRepository.findAll().stream()  
 .map(item -> modelMapper.map(item, ItemDto.class))  
 .collect(Collectors.*toList*());  
 }  
  
 @Override  
 public ItemDto getItemById(Long id) {  
 Item item = itemRepository.findById(id)  
 .orElseThrow(() -> new EntityNotFoundException("No item found with ID: " + id));  
 return modelMapper.map(item, ItemDto.class);  
 }  
  
 @Override  
 public void createItem(Item itemDto) {  
 Item item = modelMapper.map(itemDto, Item.class);  
 itemRepository.save(item);  
 }  
  
 @Override  
 public void deleteItem(Long id) {  
 itemRepository.deleteById(id);  
 }  
}

Рисунок – 2.7.2 – Содержимое имплементирующего класса **ItemServiceImpl.**

**ShoppingListService** — интерфейс, который определяет операции для работы со списками покупок:

* getAllShoppingLists(): Получение всех списков покупок.
* getShoppingListById(Long id): Получение списка покупок по его идентификатору.
* createShoppingList(ShoppingListDto shoppingListDto): Создание нового списка покупок.
* addItemToShoppingList(AddItemToShoppingListDto dto): Добавление товара в список покупок.
* updateItemQuantity(AddItemToShoppingListDto dto): Обновление количества товара в списке покупок.
* removeItemFromShoppingList(AddItemToShoppingListDto dto): Удаление товара из списка покупок.
* removeItemList(Long id): Удаление списка покупок.

Реализация этого интерфейса в классе ShoppingListServiceImpl использует репозитории для взаимодействия с сущностями ShoppingList и ShoppingListItem, а также DTO для преобразования данных.

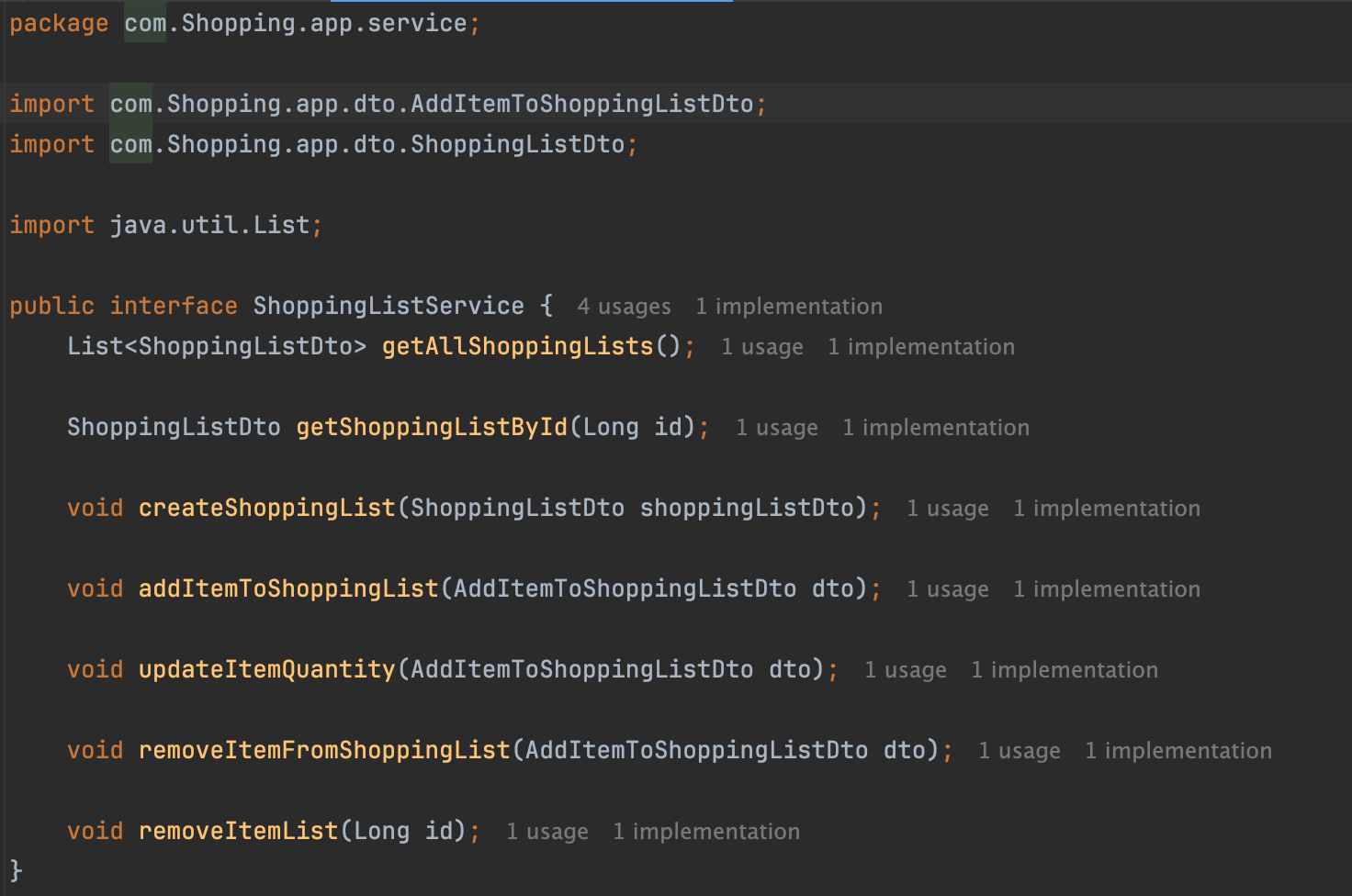


Рисунок – 2.7.3 – Содержимое интерфейса **ShoppingListService.**

package com.Shopping.app.service.impl;  
  
import com.Shopping.app.dto.AddItemToShoppingListDto;  
import com.Shopping.app.dto.ShoppingListDto;  
import com.Shopping.app.entity.Item;  
import com.Shopping.app.entity.ShoppingList;  
import com.Shopping.app.entity.ShoppingListItem;  
import com.Shopping.app.repository.ItemRepository;  
import com.Shopping.app.repository.ShoppingListItemRepository;  
import com.Shopping.app.repository.ShoppingListRepository;  
import com.Shopping.app.service.ShoppingListService;  
import jakarta.persistence.EntityNotFoundException;  
import lombok.RequiredArgsConstructor;  
import org.modelmapper.ModelMapper;  
import org.springframework.stereotype.Service;  
  
import java.util.List;  
import java.util.stream.Collectors;  
  
@Service  
@RequiredArgsConstructor  
public class ShoppingListServiceImpl implements ShoppingListService {  
  
 private final ShoppingListRepository shoppingListRepository;  
 private final ShoppingListItemRepository shoppingListItemRepository;  
 private final ItemRepository itemRepository;  
 private final ModelMapper modelMapper;  
  
 @Override  
 public List<ShoppingListDto> getAllShoppingLists() {  
 return shoppingListRepository.findAll().stream()  
 .map(shoppingList -> modelMapper.map(shoppingList, ShoppingListDto.class))  
 .collect(Collectors.*toList*());  
 }  
  
 @Override  
 public ShoppingListDto getShoppingListById(Long id) {  
 ShoppingList shoppingList = shoppingListRepository.findById(id)  
 .orElseThrow(() -> new EntityNotFoundException("Shopping list not found with ID: " + id));  
  
 return modelMapper.map(shoppingList, ShoppingListDto.class);  
 }  
  
 @Override  
 public void createShoppingList(ShoppingListDto shoppingListDto) {  
 ShoppingList shoppingList = modelMapper.map(shoppingListDto, ShoppingList.class);  
 shoppingListRepository.save(shoppingList);  
 }  
  
 @Override  
 public void addItemToShoppingList(AddItemToShoppingListDto dto) {  
 ShoppingList shoppingList = shoppingListRepository.findById(dto.getShoppingListId())  
 .orElseThrow(() -> new EntityNotFoundException("Shopping list not found with ID: " + dto.getShoppingListId()));  
  
 Item item = itemRepository.findById(dto.getItemId())  
 .orElseThrow(() -> new EntityNotFoundException("Item not found with ID: " + dto.getItemId()));  
  
 ShoppingListItem shoppingListItem = new ShoppingListItem();  
 shoppingListItem.setShoppingList(shoppingList);  
 shoppingListItem.setItem(item);  
 shoppingListItem.setQuantity(dto.getQuantity());  
  
 shoppingListItemRepository.save(shoppingListItem);  
 }  
  
 @Override  
 public void updateItemQuantity(AddItemToShoppingListDto dto) {  
 ShoppingListItem shoppingListItem = shoppingListItemRepository.findByShoppingListIdAndItemId(dto.getShoppingListId(), dto.getItemId())  
 .orElseThrow(() -> new EntityNotFoundException("Item not found in shopping list" + dto.getItemId()));  
  
 shoppingListItem.setQuantity(dto.getQuantity());  
 shoppingListItemRepository.save(shoppingListItem);  
 }  
  
 @Override  
 public void removeItemFromShoppingList(AddItemToShoppingListDto dto) {  
 ShoppingListItem shoppingListItem = shoppingListItemRepository.findByShoppingListIdAndItemId(dto.getShoppingListId(), dto.getItemId())  
 .orElseThrow(() -> new EntityNotFoundException("Item not found in shopping list"));  
  
 shoppingListItemRepository.delete(shoppingListItem);  
 }  
  
 @Override  
 public void removeItemList(Long id) {  
 shoppingListRepository.deleteById(id);  
 }  
}

Рисунок – 2.7.4 – Содержимое имплементирующего класса **ShoppingListServiceImpl.**

Сервисы в проекте отвечают за бизнес-логику и взаимодействие с репозиториями. Они обеспечивают выполнение операций с сущностями и преобразование данных в удобный формат (DTO) для передачи между слоями приложения. Сервисы позволяют изолировать логику работы с данными от контроллеров и других частей приложения, что делает код более структурированным, поддерживаемым и тестируемым.

## 8. Разработка классов контроллера

Контроллеры в проекте отвечают за обработку HTTP-запросов и передачу данных между клиентом и сервисами. В Spring, контроллеры используют аннотации для определения маршрутов и типов запросов, а также инжектируют сервисы для выполнения бизнес-логики. Контроллеры служат как посредники между клиентом и сервисами, обрабатывая запросы и возвращая ответы.

**ShoppingListController**

Контроллер ShoppingListController обрабатывает HTTP-запросы, связанные со списками покупок. Он использует сервис ShoppingListService для выполнения операций с данными. Контроллер принимает и отправляет данные в формате JSON.

**Основные операции контроллера:**

* **Получение всех списков покупок**: обработка GET-запроса на /api/shopping-lists, который возвращает список всех списков покупок.
* **Получение списка покупок по ID**: обработка GET-запроса на /api/shopping-lists/{id}, который возвращает список покупок по его ID.
* **Создание нового списка покупок**: обработка POST-запроса на /api/shopping-lists, который создает новый список покупок.
* **Добавление товара в список покупок**: обработка POST-запроса на /api/shopping-lists/add-item, который добавляет товар в список покупок.
* **Обновление количества товара в списке покупок**: обработка PUT-запроса на /api/shopping-lists/update-item-quantity, который обновляет количество товара в списке покупок.
* **Удаление товара из списка покупок**: обработка DELETE-запроса на /api/shopping-lists/remove-item, который удаляет товар из списка покупок.
* **Удаление списка покупок**: обработка GET-запроса на /api/shopping-lists/remove-list/{id}, который удаляет список покупок.

package com.Shopping.app.controller;  
  
import com.Shopping.app.dto.AddItemToShoppingListDto;  
import com.Shopping.app.dto.ShoppingListDto;  
import com.Shopping.app.service.ShoppingListService;  
import lombok.RequiredArgsConstructor;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
@RequestMapping("/api/shopping-lists")  
@CrossOrigin(origins = "http://localhost:5173")  
@RequiredArgsConstructor  
public class ShoppingListController {  
  
 private final ShoppingListService shoppingListService;  
  
 @GetMapping  
 public List<ShoppingListDto> getAllShoppingLists() {  
 return shoppingListService.getAllShoppingLists();  
 }  
  
 @GetMapping("/{id}")  
 public ShoppingListDto getShoppingListById(@PathVariable Long id) {  
 return shoppingListService.getShoppingListById(id);  
 }  
  
 @PostMapping  
 public void createShoppingList(@RequestBody ShoppingListDto shoppingListDto) {  
 shoppingListService.createShoppingList(shoppingListDto);  
 }  
  
 @PostMapping("/add-item")  
 public void addItemToShoppingList(@RequestBody AddItemToShoppingListDto dto) {  
 shoppingListService.addItemToShoppingList(dto);  
 }  
  
 @PutMapping("/update-item-quantity")  
 public void updateItemQuantity(@RequestBody AddItemToShoppingListDto dto) {  
 shoppingListService.updateItemQuantity(dto);  
 }  
  
 @DeleteMapping("/remove-item")  
 public void removeItemFromShoppingList(@RequestBody AddItemToShoppingListDto dto) {  
 shoppingListService.removeItemFromShoppingList(dto);  
 }  
  
 @GetMapping("/remove-list/{id}")  
 public void removeItemList(@PathVariable Long id) {  
 shoppingListService.removeItemList(id);  
 }  
}

Рисунок – 2.8.1 – Содержимое класса **ShoppingListController.**

* **@RestController**: Указывает, что этот класс является REST контроллером, и все методы будут возвращать данные в формате JSON или XML.
* **@RequestMapping("/api/shopping-lists")**: Устанавливает базовый путь для всех маршрутов в этом контроллере. Например, для получения всех списков покупок путь будет /api/shopping-lists.
* **@GetMapping**, **@PostMapping**, **@PutMapping**, **@DeleteMapping**: Аннотации для обработки соответствующих HTTP-методов (GET, POST, PUT, DELETE). Каждая аннотация указывает на маршрут, который будет обрабатываться этим методом.
* **@PathVariable**: Аннотация для извлечения значений из пути URL. Например, в маршруте /api/shopping-lists/{id}, id будет извлечено из пути и передано в метод как аргумент.
* **@RequestBody**: Указывает, что тело запроса должно быть связано с параметром метода. Это используется для передачи данных, например, DTO, в POST или PUT запросах.
* **@CrossOrigin(origins = "[http://localhost:5173](http://localhost:5173" \t "_new)")**: Разрешает доступ к API с указанных источников (в данном случае с фронтенда, который работает на порту 5173).

**ItemController**

Контроллер ItemController обрабатывает HTTP-запросы, связанные с операциями над товарами. Он использует сервис ItemService для выполнения операций с товарами, таких как создание, получение и удаление товаров. Контроллер принимает и отправляет данные в формате JSON.

**Основные операции контроллера:**

* **Получение всех товаров**: обработка GET-запроса на /api/items, который возвращает список всех товаров.
* **Получение товара по ID**: обработка GET-запроса на /api/items/{id}, который возвращает товар по его ID.
* **Создание нового товара**: обработка POST-запроса на /api/items, который создает новый товар.
* **Удаление товара**: обработка DELETE-запроса на /api/items/{id}, который удаляет товар по его ID.

package com.Shopping.app.controller;  
  
import com.Shopping.app.dto.ItemDto;  
import com.Shopping.app.entity.Item;  
import com.Shopping.app.service.ItemService;  
import lombok.RequiredArgsConstructor;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.List;  
  
@RestController  
@RequestMapping("/api/items")  
@RequiredArgsConstructor  
@CrossOrigin(origins = "http://localhost:5173")  
public class ItemController {  
  
 private final ItemService itemService;  
  
 @GetMapping  
 public List<ItemDto> getAllItems() {  
 return itemService.getAllItems();  
 }  
  
 @GetMapping("/{id}")  
 public ItemDto getItemById(@PathVariable Long id) {  
 return itemService.getItemById(id);  
 }  
  
 @PostMapping  
 public void createItem(@RequestBody Item itemDto) {  
 itemService.createItem(itemDto);  
 }  
  
 @DeleteMapping("/{id}")  
 public void delete(@PathVariable("id") Long id) {  
 itemService.deleteItem(id);  
 }  
}

Рисунок – 2.8.2 – Содержимое класса **ItemController.**

## 9. Дерево проекта

Таким образом после разработки мы получаем финальное дерево проекта:

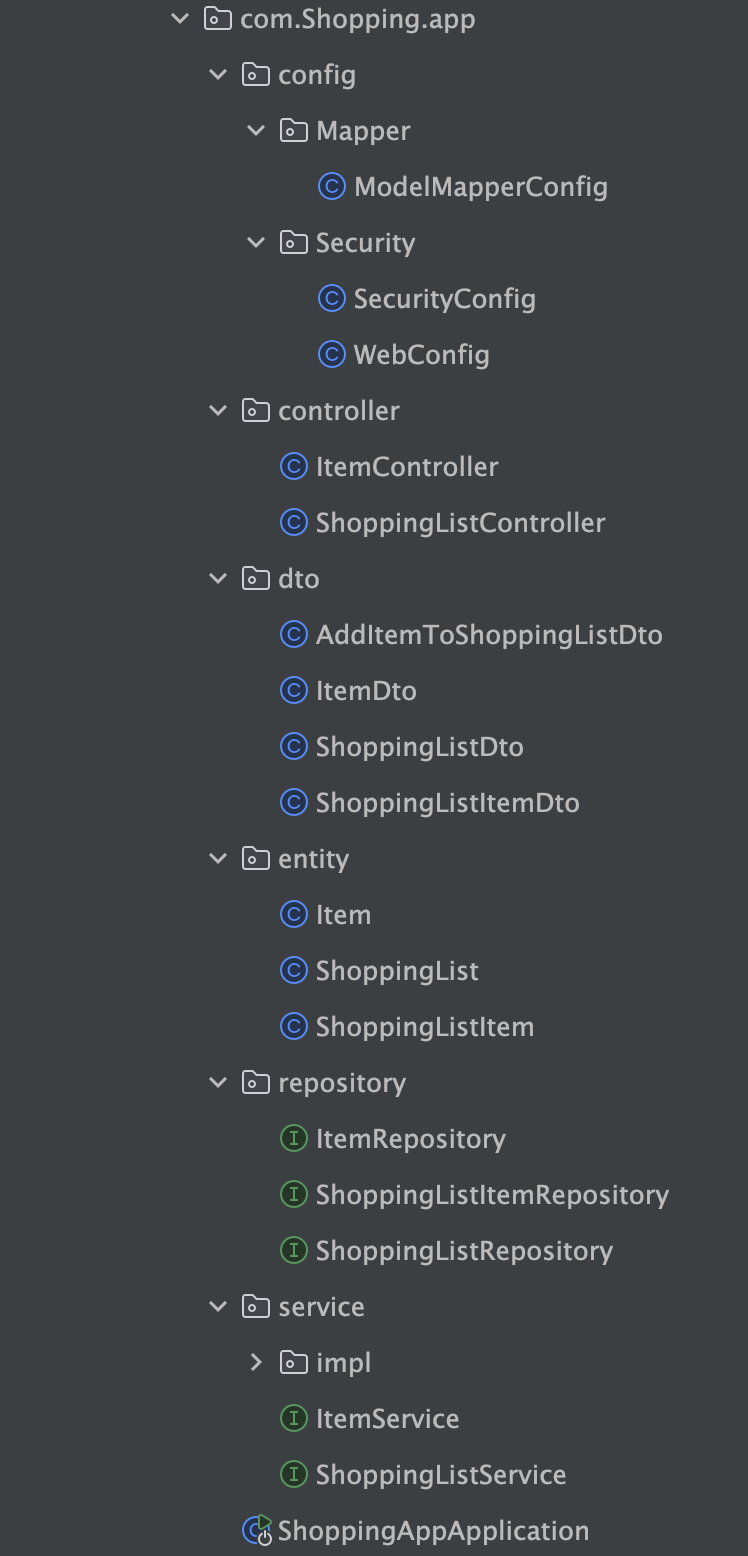


Рисунок – 2.9.1 – Дерево проекта**.**

# ГЛАВА 3. РАЗРАБОТКА FRONTEND ЧАСТИ ВЕБ-ПРИЛОЖЕНИЯ

## 1. Исследование структуры проекта

Проект состоит из нескольких директорий и файлов, организованных следующим образом:

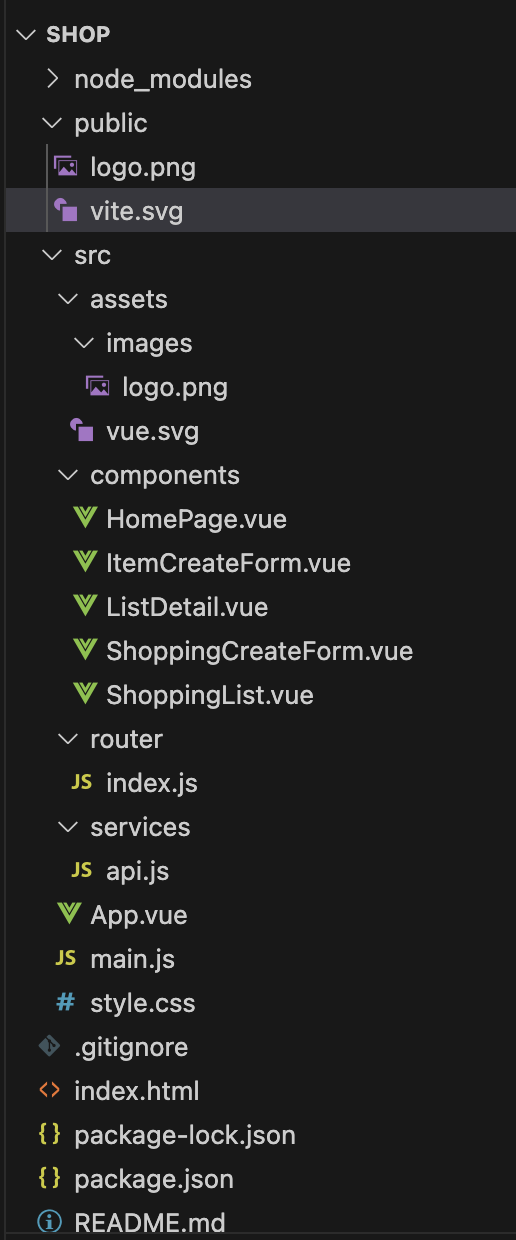
****

Рисунок – 3.1.1 – Структура проекта

**Директория node\_modules**:  
Содержит все зависимости и модули, необходимые для работы проекта.

**Директория public**:  
Содержит статические файлы, доступные напрямую.

* logo.png
* vite.svg

**Директория src**:  
Содержит исходный код приложения, включая несколько поддиректорий и файлов.

* **assets**:  
  Содержит директорию images с двумя файлами:
  + logo.png
  + vue.svg
* **components**:  
  Содержит пять Vue компонентов:
  + HomePage.vue: Главная страница приложения.
  + ItemCreateForm.vue: Форма для создания новых товаров.
  + ListDetail.vue: Компонент для отображения подробностей списка покупок.
  + ShoppingCreateForm.vue: Форма для создания нового списка покупок.
  + ShoppingList.vue: Компонент для отображения списка покупок.
* **router**:  
  Содержит один файл:
  + index.js: Конфигурация маршрутов приложения.
* **services**:  
  Содержит один файл:
  + api.js: Настройки для взаимодействия с сервером через API.

**Корневые файлы**:

* App.vue: Главный компонент приложения.
* main.js: Главный файл для инициализации и запуска приложения.
* style.css: Файл стилей для приложения.
* .gitignore: Файл для игнорирования файлов и директорий в Git.
* index.html: Основной HTML файл.
* package-lock.json: Файл блокировки версий зависимостей.
* package.json: Файл с метаданными проекта и зависимостями.
* README.md: Документация проекта.

Такая структура проекта позволяет легко управлять кодом, обеспечивать его модульность и повторное использование компонентов. Каждая директория и файл имеют свое предназначение, что упрощает навигацию и обслуживание проекта.

## 2. Маршрутизация c использованием Vue Router

Маршрутизация — это важный компонент в разработке веб-приложений на Vue, который управляет тем, какие компоненты отображаются в зависимости от текущего URL-адреса в браузере. Приведённый код демонстрирует использование Vue Router — стандартной библиотеки для реализации маршрутизации в приложениях Vue.

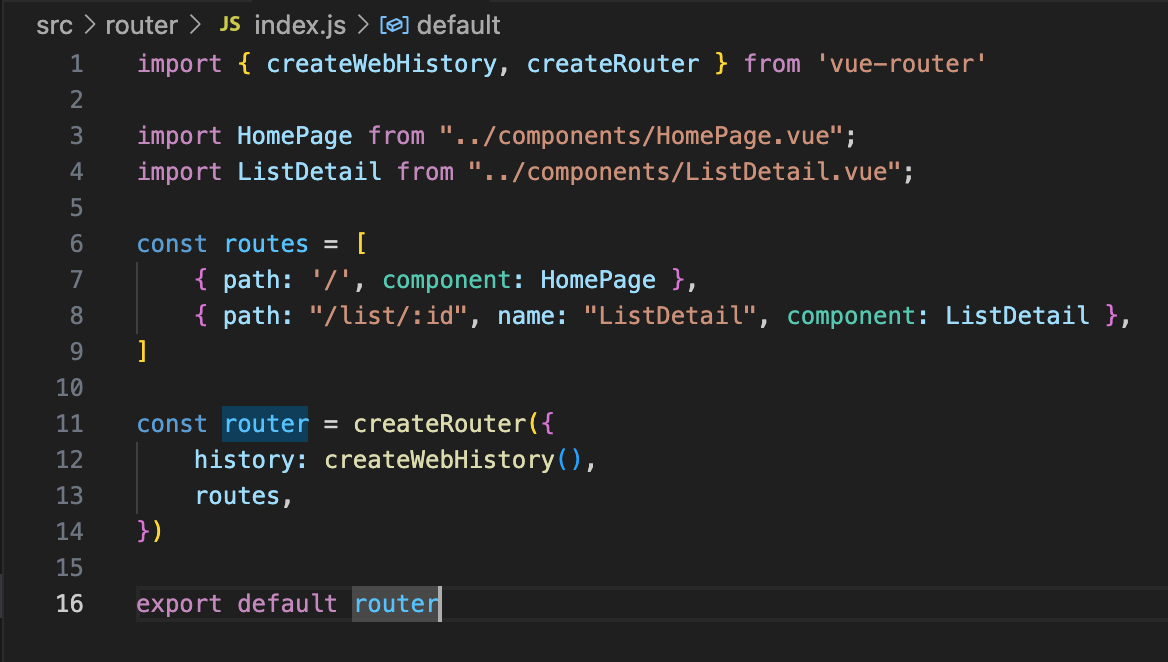


Рисунок – 3.2.1 – Фрагмент кода для реализации маршрутов.

1. **Использование createRouter и createWebHistory**: Эти методы используются для создания маршрутизатора, который управляет навигацией в приложении. Метод createWebHistory позволяет использовать HTML5 History API, что позволяет создавать чистые URL без хеш-символа (#).
2. **Маршруты**:
   * **path: '/'** — отображает компонент HomePage на главной странице приложения.
   * **path: "/list/:id"** — динамический маршрут, где :id является параметром, который передается в компонент ListDetail для отображения подробной информации о списке, выбранном по id.
3. **Динамические маршруты**: Параметры, передаваемые в URL, такие как :id, позволяют динамически загружать контент в компоненте. Это важно, когда требуется отображать различные данные, в зависимости от того, какой объект был выбран.
4. **Экспорт маршрутизатора**: После создания маршрутов, экземпляр router экспортируется для использования в основном приложении, что позволяет Vue управлять навигацией между различными компонентами.

## 3. Проектная структура и архитектура

Проектная структура и компоненты веб-приложения для создания и управления списками покупок на основе Vue.js и API, выполнены с учетом принципов модульности и простоты использования. Основные компоненты включают:

1. **ShoppingList.vue** – отвечает за отображение всех доступных списков покупок. Он использует данные, передаваемые через props, и предоставляет возможность удалять списки.
2. **ShoppingCreateForm.vue** – позволяет пользователю создавать новый список покупок, проверяя на уникальность имя списка и отправляя запрос на сервер для его добавления.
3. **ItemCreateForm.vue** – реализует создание новых товаров с проверкой на корректность введенной цены. После создания товара данные отправляются на сервер для добавления в базу данных.
4. **ListDetail.vue** – отображает детали конкретного списка покупок, включая товары, их количество, цену и общую сумму. Этот компонент также позволяет добавлять и удалять товары из списка, а также управлять ценой через вычисления.
5. **HomePage.vue** – главный компонент приложения, который агрегирует компоненты для отображения всех списков покупок и формы для создания нового списка.

Компонент ShoppingList.vue — отображение списка покупок

<script setup>

import { onMounted, ref } from "vue";

import apiClient from "../services/api.js";

const props = defineProps(["data"]);

const deleteList = async (id, index) => {

await apiClient.get(`/shopping-lists/remove-list/${id}`);

props.data.splice(index, 1);

};

</script>

<template>

<div>

<h2>Ваши списки покупок:</h2>

<div class="list">

<div class="card" v-for="(item, index) in props.data" :key="item.id">

<h3>{{ item.name }}</h3>

<p>Количество товаров: {{ item.items.length }}</p>

<router-link :to="{ name: 'ListDetail', params: { id: item.id } }">

Открыть список

</router-link>

<button @click="deleteList(item.id, index)">Удалить</button>

</div>

</div>

</div>

</template>

<style scoped>

.list {

display: flex;

flex-wrap: wrap;

gap: 20px;

justify-content: center;

}

.card {

background-color: #1e1e1e;

padding: 15px;

border-radius: 10px;

box-shadow: 0 2px 5px rgba(0, 0, 0, 0.5);

width: 250px;

text-align: center;

color: #e0e0e0;

}

.card h3 {

margin-bottom: 10px;

color: #646cff;

}

button {

margin-top: 10px;

}

</style>

Рисунок – 3.3.1 – Код для реализации компонента ShoppingList.vue.

Компонент ShoppingCreateForm.vue — создание нового списка

<script setup>

import { ref } from "vue";

import apiClient from "../services/api.js";

const emit = defineEmits(["create"]);

const nameShoppingList = ref("");

const errorMessage = ref("");

const createShoppingListRequest = async () => {

if (!nameShoppingList.value.trim()) {

errorMessage.value = "Название списка не может быть пустым!";

return;

}

const existingLists = await apiClient.get("/shopping-lists");

const isDuplicate = existingLists.data.some(

(list) => list.name.toLowerCase() === nameShoppingList.value.toLowerCase()

);

if (isDuplicate) {

errorMessage.value = "Список с таким именем уже существует!";

return;

}

await apiClient.post("/shopping-lists", { name: nameShoppingList.value });

nameShoppingList.value = "";

errorMessage.value = "";

emit("create");

};

</script>

<template>

<form @submit.prevent="createShoppingListRequest" class="form">

<input

type="text"

v-model="nameShoppingList"

placeholder="Введите название списка"

/>

<button type="submit">Создать список</button>

</form>

</template>

<style scoped>

.form {

display: flex;

justify-content: center;

gap: 10px;

margin-top: 20px;

}

input {

padding: 10px;

border: 1px solid #424242;

border-radius: 15px;

background-color: #1e1e1e;

color: white;

}

button {

padding: 10px 20px;

}

</style>

Рисунок – 3.3.2 – Код для реализации компонента ShoppingCreateForm.vue.

Компонент для создания товара ItemCreateForm.vue

<script setup>

import { ref } from "vue";

import apiClient from "../services/api.js";

const emit = defineEmits(["create"]);

const itemName = ref("");

const itemDescription = ref("");

const itemPrice = ref("");

const createItemRequest = async () => {

if (itemPrice.value < 0) {

alert("Цена не может быть отрицательной");

return;

}

await apiClient.post("/items", {

name: itemName.value,

description: itemDescription.value,

price: itemPrice.value,

});

itemName.value = "";

itemDescription.value = "";

itemPrice.value = "";

emit("create");

};

</script>

<template>

<form @submit.prevent="createItemRequest" class="form-container">

<input type="text" v-model="itemName" placeholder="Название" required class="input-field" />

<input type="text" v-model="itemDescription" placeholder="Описание" class="input-field" />

<input

type="number"

v-model="itemPrice"

placeholder="Цена"

min="0"

step="0.01"

required

class="input-field"

/>

<button type="submit" class="btn-submit">Создать</button>

</form>

</template>

<style scoped>

.form-container {

display: flex;

flex-direction: column;

gap: 15px;

max-width: 400px;

margin: 0 auto;

padding: 20px;

background: #2c2c2c;

border-radius: 8px;

box-shadow: 0 4px 8px rgba(0, 0, 0, 0.3);

}

.input-field {

padding: 10px;

border: 1px solid #ccc;

border-radius: 5px;

font-size: 14px;

color: #ffffff;

}

.input-field:focus {

outline: none;

border-color: #6b63ff;

box-shadow: 0 0 5px #6b63ff;

}

.btn-submit {

background-color: #6b63ff;

color: #fff;

padding: 10px 15px;

font-size: 16px;

font-weight: bold;

border: none;

border-radius: 5px;

cursor: pointer;

transition: background 0.3s ease;

}

.btn-submit:hover {

background-color: #6b63ff;

}

</style>

Рисунок – 3.3.3 – Код для реализации компонента ItemCreateForm.vue.

Компонент для отображения деталей списка покупок ListDetail.vue

<script setup>

import { onMounted, ref } from "vue";

import { useRoute, useRouter } from "vue-router";

import apiClient from "../services/api.js";

import ItemCreateForm from "./ItemCreateForm.vue";

const route = useRoute();

const router = useRouter();

const listDetailData = ref(null);

const items = ref([]);

const totalPrice = ref(0);

const getAllItems = async () => {

const response = await apiClient.get("/items");

items.value = response.data;

};

const calculateTotalPrice = (itemId, quantity) => {

const item = items.value.find(i => i.id === itemId);

return item ? item.price \* quantity : 0

};

const calculateTotal = () => {

totalPrice.value = listDetailData.value.items.reduce((sum, item) => {

return sum + calculateTotalPrice(item.itemId, item.quantity);

}, 0).toFixed(2);

};

const addItem = async (listItem) => {

const itemData = {

shoppingListId: route.params.id,

itemId: listItem.id,

quantity: listItem.q,

};

const existingItem = listDetailData.value.items.find(item => item.itemId === listItem.id);

if (existingItem) {

existingItem.quantity = listItem.q;

await updateItem(existingItem);

} else {

const response = await apiClient.post("/shopping-lists/add-item", itemData);

listDetailData.value.items.push({

...itemData,

itemName: listItem.name,

});

}

calculateTotal();

};

const deleteItem = async (listItem, index) => {

const itemData = {

shoppingListId: route.params.id,

itemId: listItem.itemId,

};

await apiClient.delete('/shopping-lists/remove-item', { data: itemData });

listDetailData.value.items.splice(index, 1);

calculateTotal();

};

const deleteFromItems = async (itemId) => {

await apiClient.delete(`/items/${itemId}`);

items.value = items.value.filter(item => item.id !== itemId);

};

const goToHomePage = () => {

router.push("/");

};

onMounted(async () => {

const response = await apiClient.get(`/shopping-lists/${route.params.id}`);

listDetailData.value = response.data;

await getAllItems();

calculateTotal();

});

</script>

<template>

<div v-if="listDetailData" class="shopping-list-container">

<h1 class="title">{{ listDetailData.name }}</h1>

<div class="list-items">

<h2>Список товаров</h2>

<div v-for="(item, index) in listDetailData.items" :key="index" class="item-card">

<p>

<span class="item-name">{{ item.itemName }}</span> |

<span>Количество: {{ item.quantity }}</span> |

<span>Цена: {{ calculateTotalPrice(item.itemId, item.quantity) }} BYN</span>

</p>

<button class="btn delete-btn" @click="deleteItem(item, index)">Удалить</button>

</div>

</div>

<div class="available-items">

<h2>Все доступные товары</h2>

<div v-for="listItem in items" :key="listItem.id" class="item-card">

<p>

<span class="item-name">{{ listItem.name }}</span> |

<span>Цена: {{ listItem.price }} BYN</span>

</p>

<input type="number" v-model.number="listItem.q" min="1" class="quantity-input" placeholder="Количество" />

<button class="btn add-btn" @click="addItem(listItem)">Добавить</button>

<button class="btn delete-btn" @click="deleteFromItems(listItem.id)">Удалить</button>

</div>

</div>

<div class="add-items">

<h2>Добавить новый товар</h2>

<ItemCreateForm @create="getAllItems" />

</div>

<div class="total-price">

<h2>Общая сумма: <span>{{ totalPrice }} BYN</span></h2>

<div class="action-buttons">

<button class="btn back-btn" @click="goToHomePage">Вернуться</button>

</div>

</div>

</div>

</template>

<style scoped>

.shopping-list-container {

background: #1e1e1e;

color: #fff;

padding: 20px;

border-radius: 8px;

max-width: 800px;

margin: 0 auto;

box-shadow: 0 4px 8px rgba(0, 0, 0, 0.3);

}

.title {

text-align: center;

color: #6b63ff;

}

.action-buttons {

display: flex;

justify-content: flex-end, right;

margin: 20px 0;

}

.btn {

background: #6b63ff;

color: white;

border: none;

padding: 10px 15px;

margin: 5px;

border-radius: 5px;

cursor: pointer;

font-size: 14px;

}

.btn.delete-btn {

background: #e74c3c;

}

.btn:hover {

opacity: 0.8;

}

.list-items, .available-items {

margin: 20px 0;

}

.item-card {

background: #2c2c2c;

border-radius: 5px;

margin: 10px 0;

padding: 10px;

display: flex;

justify-content: space-between;

align-items: center;

}

.item-name {

font-weight: bold;

color: #6b63ff;

}

.quantity-input {

padding: 5px;

width: 50px;

margin: 0 10px;

}

.total-price {

text-align: center;

font-size: 18px;

margin-top: 20px;

font-weight: bold;

}

h2 {

color: #6b63ff;

text-align: center;

}

</style>

Рисунок – 3.3.4 – Код для реализации компонента ListDetails.vue.

Основной компонент (HomePage.vue)

<script setup>

import ShoppingList from "./ShoppingList.vue";

import ShoppingCreateForm from "./ShoppingCreateForm.vue";

import { onMounted, ref } from "vue";

import apiClient from "../services/api.js";

const shoppingLists = ref([]);

const getShoppingList = async () => {

const response = await apiClient.get("/shopping-lists");

shoppingLists.value = response.data;

};

onMounted(async () => {

await getShoppingList();

console.log(shoppingLists.value);

});

</script>

<template>

<div class="app-container">

<h1>Создайте и управляйте своими покупками!</h1>

<ShoppingList :data="shoppingLists" />

<ShoppingCreateForm @create="getShoppingList" />

</div>

<div class="logo-container">

<img src="/Users/maksimlych/Desktop/курсач/shop/src/assets/images/logo.png" alt="Logo" class="logo" />

</div>

</template>

<style scoped>

body {

background-color: #121212;

color: #e0e0e0;

font-family: Arial, sans-serif;

margin: 0;

padding: 0;

}

.app-container {

max-width: 800px;

margin: 0 auto;

padding: 20px;

}

h1 {

text-align: center;

color: #646cff;

margin-bottom: 100px;

}

button {

background-color: #0d4ba8;

color: white;

border: none;

padding: 10px 15px;

border-radius: 5px;

cursor: pointer;

margin: 5px 0;

transition: background-color 0.3s ease;

}

button:hover {

background-color: #646cff;

}

button:active {

background-color: #0d47a1;

}

input[type="text"],

input[type="number"] {

padding: 10px;

margin: 5px 0;

border: 1px solid #424242;

border-radius: 5px;

background-color: #1e1e1e;

color: white;

width: calc(100% - 22px);

box-sizing: border-box;

}

input::placeholder {

color: #9e9e9e;

}

.list {

display: flex;

flex-wrap: wrap;

gap: 20px;

justify-content: center;

margin-top: 20px;

}

.card {

background-color: #1e1e1e;

padding: 5px;

border-radius: 10px;

color: #e0e0e0;

width: 250px;

text-align: center;

box-shadow: 0 2px 5px rgba(0, 0, 0, 0.5);

}

.card p {

margin: 5px 0;

}

.router-link {

display: inline-block;

margin-top: 10px;

color: #00bcd4;

text-decoration: none;

font-weight: bold;

}

.router-link:hover {

text-decoration: underline;

}

.logo-container {

position: fixed;

top: 50px;

left: 50px;

}

.logo {

width: 150px;

height: auto;

}

</style>

Рисунок – 3.3.5 – Код для реализации компонента HomePage.vue.

Архитектура приложения построена на компонентном подходе с четким разделением логики и представления. Каждый компонент отвечает за конкретную задачу, что упрощает поддержку и расширение приложения. Компоненты связаны с сервером через API, что позволяет эффективно работать с данными на серверной стороне, обеспечивая динамичное обновление пользовательского интерфейса.

Такой подход к структуре и архитектуре приложения позволяет легко добавлять новые функции, поддерживать масштабируемость и обеспечивать удобный интерфейс для пользователя.

## 4. Api Client

Для взаимодействия с сервером в приложении используется клиент **Axios**, который настраивается с базовым URL и необходимыми заголовками для отправки запросов.

import axios from 'axios';

const apiClient = axios.create({

baseURL: 'http://localhost:8080/api',

headers: {

'Content-Type': 'application/json',

},

});

export default apiClient;

Рисунок – 3.4.1 – Код для реализации компонента HomePage.vue.

1. **baseURL** – Указывает на основной адрес сервера API, с которым приложение будет взаимодействовать. В данном случае это локальный сервер на порту 8080. Это позволяет сократить дублирование адресов при отправке различных запросов, так как они будут автоматически направляться на указанный сервер.
2. **headers** – Устанавливает заголовки HTTP-запросов. В данном примере указывается, что отправляемые данные имеют формат **JSON**. Это важно для правильного взаимодействия с сервером, который ожидает данные в данном формате.

Этот клиент используется в различных компонентах приложения для выполнения операций с API, таких как:

* Получение данных (например, списков покупок, товаров и т. д.),
* Отправка данных на сервер (например, создание нового списка покупок или товара),
* Удаление данных (например, удаление товаров или списков).

## 5. Интерфейс

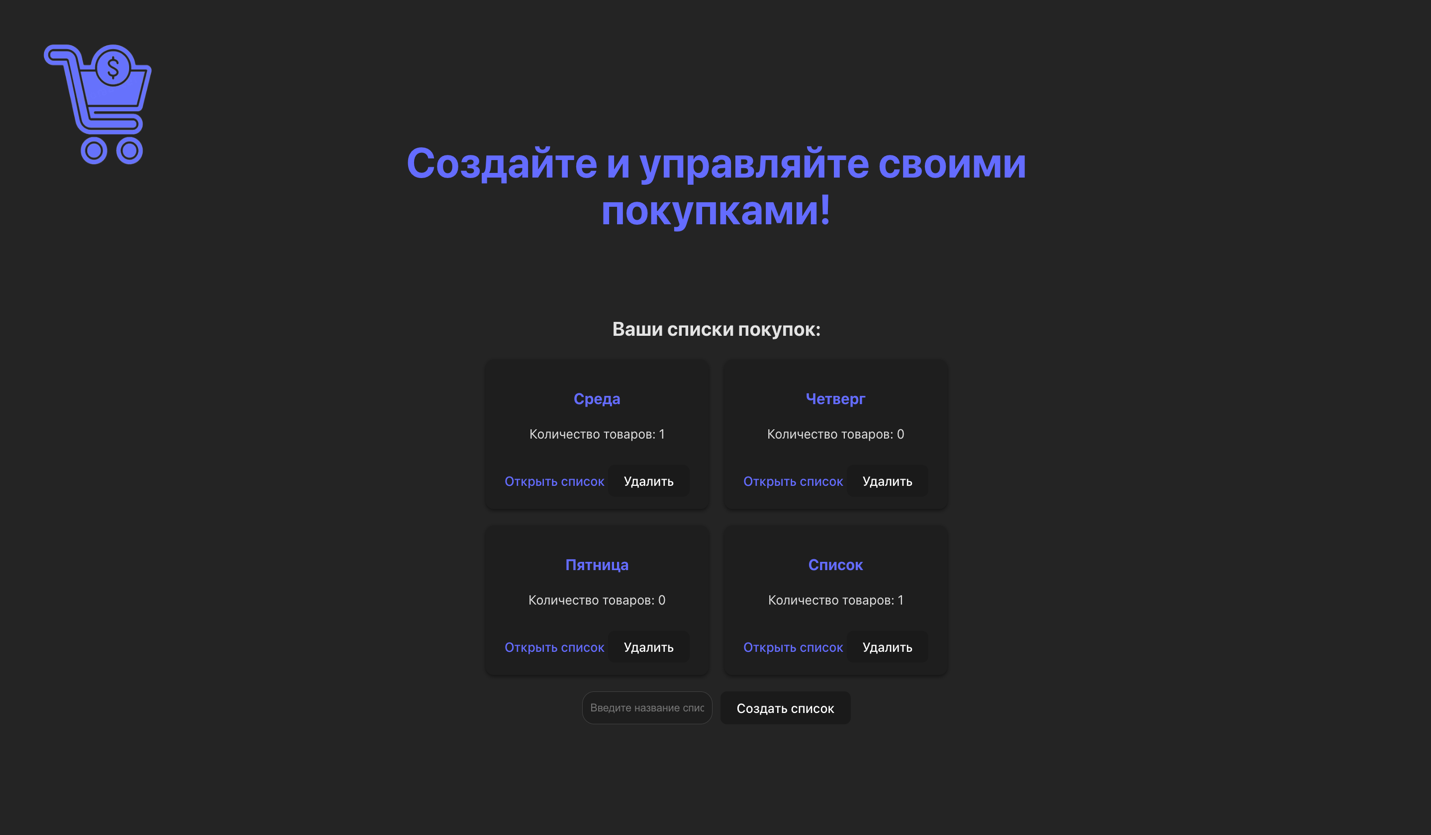


Рисунок – 3.5.1 – Внешний вид домашней страницы.

Пользовательский интерфейс имеет различные функции:

1. Создание нового списка покупок.
2. Удаление существующего списка покупок.
3. Открыть уже существующий список покупок.

Данный интерфейс содержит необходимые функции для удобного и быстрого взаимодействия со списком покупок.



Рисунок – 3.5.2 – Внешний вид страницы взаимодействия со списком покупок.

Интерфейс взаимодействия со списком покупок имеет функции:

1. Выбрать товар из существующих в список покупок
2. Добавить свой товар
3. Удалить товар из существующих
4. Удалить товар из списка покупок
5. Добавить товар в список покупок
6. Вернуться на домашнюю страницу

Данное решение позволяет быстро и удобно взаимодействовать с выбранным списком покупок. На страницы можно увидеть общую сумма товара добавленного в список, данная сумма учитывает количество товара в списке.

# 

# ЗАКЛЮЧЕНИЕ

В результате выполненного курсового проекта было разработано веб-приложение, позволяющее пользователям создавать, редактировать и удалять списки покупок, а также управлять содержимым этих списков. Реализован удобный пользовательский интерфейс с функциями добавления новых товаров и выбора из предложенного списка.

Благодаря использованию современных технологий и инструментов, таких как Java, Spring Boot, Vue и PostgreSQL, удалось создать стабильное и интуитивно понятное приложение для эффективного управления повседневными покупками.

Возможные направления дальнейшего развития проекта  
Данный проект может быть развит в следующих направлениях:

• **Расширение функциональности**: Добавление функций напоминаний о покупках, автоматического расчета стоимости и создания шаблонов для повторяющихся списков.  
• **Интеграция с внешними сервисами**: Внедрение возможности импорта и экспорта списков покупок, а также интеграция с онлайн-магазинами для проверки цен и оформления заказов.  
• **Рекомендательная система**: Разработка рекомендаций на основе предыдущих покупок или популярных товаров среди пользователей.  
• **Мобильная адаптация**: Создание мобильного приложения или оптимизация интерфейса для работы на мобильных устройствах для повышения удобства использования.  
• **Интернационализация**: Добавление поддержки нескольких языков и валют для привлечения международной аудитории.  
• **Масштабируемость**: Оптимизация приложения для работы с большим числом пользователей и списков при сохранении производительности и надежности.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Официальная документация Java 17 – <https://docs.oracle.com/javase/17/>
2. Официальная документация Spring Boot – <https://spring.io/projects/spring-boot>
3. Документация Hibernate – https://hibernate.org/orm/documentation/
4. PostgreSQL Documentation – <https://www.postgresql.org/docs/>
5. Официальная документация H2 Database – <https://www.h2database.com/>
6. Liquibase Documentation – https://www.liquibase.org/documentation/
7. Lombok Documentation – <https://projectlombok.org/>
8. ModelMapper Documentation – <http://modelmapper.org/>
9. Jackson Databind Documentation – [https://github.com/FasterXML/jackson-databind](https://github.com/FasterXML/jackson-databind" \t "_new)
10. Официальная документация Vue.js – <https://vuejs.org/>
11. Axios Documentation – <https://axios-http.com/>
12. Официальная документация Maven – <https://maven.apache.org/>