

Лабораторная работа № 7

Разработка API и серверной части приложения

Цель работы: получить навыки проектирования и разработки серверного приложения, реализации бизнес-логики, REST API, а также настройки CI/CD и инструментов контроля качества кода.

В рамках лабораторной работы было разработано серверное приложение для интернет-магазина техники "**TechStore**". Приложение обеспечивает полный цикл работы интернет-магазина: от регистрации пользователей и просмотра каталога до оформления заказов и получения уведомлений на почту.

Техническое задание и выбор инструментов

Для реализации серверной части был выбран язык **Java 21** и фреймворк **Spring Boot 3.3.5**, так как этот стек является стандартом для разработки высоконагруженных корпоративных систем.

Используемый стек технологий:

- **Core:** Java 21, Spring Boot (Web, Data JPA, Security).
- **Database:** PostgreSQL (основное хранилище), Redis (кэширование и корзина).
- **Messaging:** Apache Kafka (событийно-ориентированная архитектура).
- **Migrations:** Liquibase.
- **Docs:** OpenAPI (Swagger).
- **Build & Deploy:** Maven, Docker, Docker Compose.
- **Quality:** SonarQube, Jacoco.

Архитектура приложения

Приложение построено по **монолитной архитектуре** с четким разделением на слои (Layered Architecture):

1. **Controller Layer:** Обработка HTTP-запросов (@RestController).
2. **Service Layer:** Бизнес-логика (@Service).
3. **Repository Layer:** Взаимодействие с БД (Spring Data JPA).

4. DTO Layer: Объекты передачи данных, маппинг через ModelMapper.

Структура проекта:

```
src/main/java/com/example/techstore
└── cart      // Модуль корзины (Redis)
└── catalog   // Товары, категории, бренды, фильтры
└── common    // Общие утилиты и исключения
└── config    // Конфигурация Security, Swagger, Cache
└── notification // Сервис уведомлений (Email)
└── order     // Заказы и события Kafka
└── review    // Отзывы и рейтинги
└── user      // Аутентификация и пользователи
└── TechStoreApplication.java
```

Реализация серверной части (Backend)

4.1. База данных и миграции (PostgreSQL + Liquibase)

В качестве СУБД используется PostgreSQL. Для управления схемой базы данных используется **Liquibase**. Это позволяет версионировать изменения БД и накатывать их автоматически при старте приложения.

Основные таблицы: users, roles, products, orders, reviews, carts (в Redis).

Пример скрипта миграции (001-initial-schema.sql):

```
CREATE TABLE products (
    product_id BIGINT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    title VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    ...
);
```

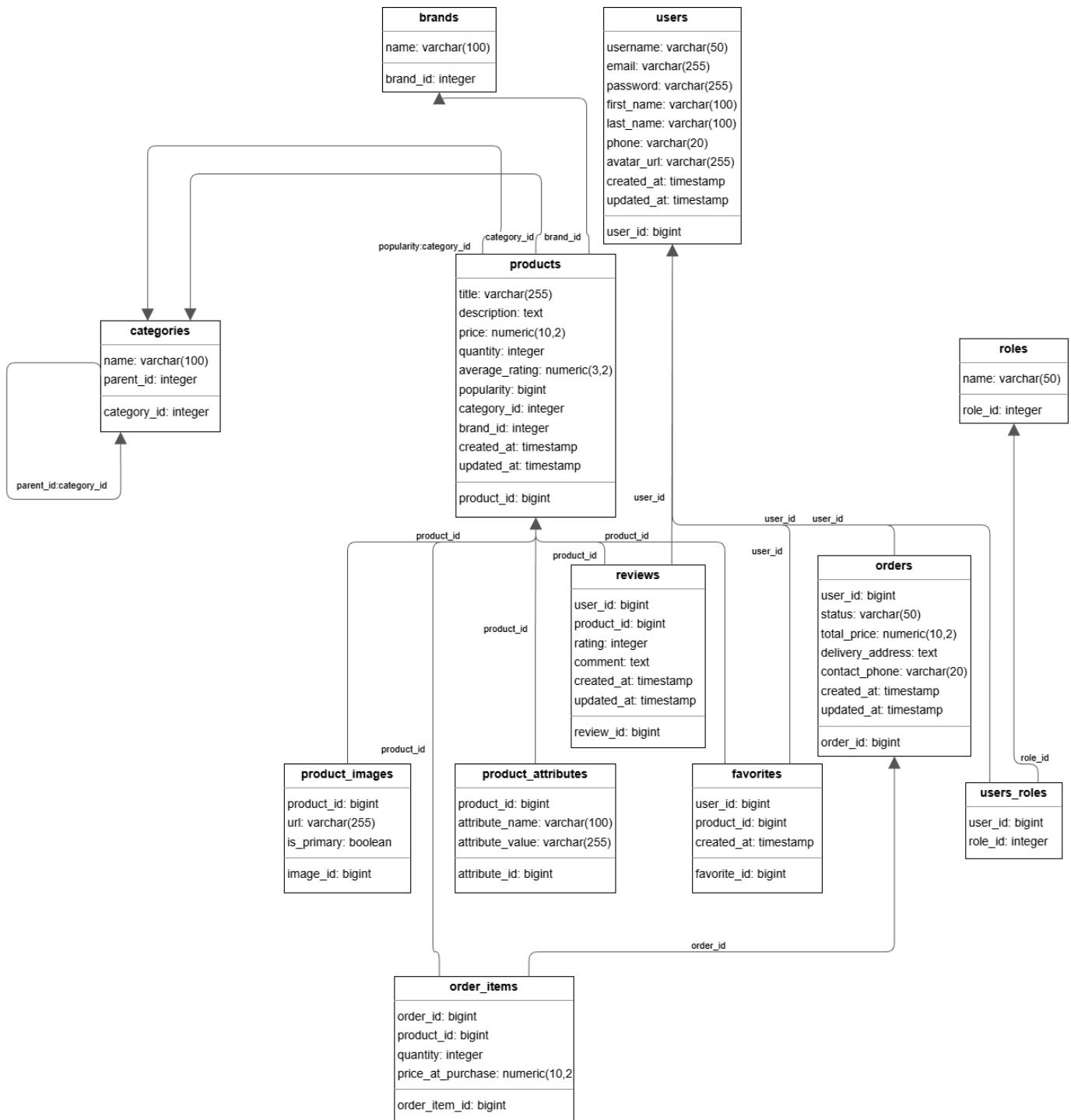


Рис. 1. Схема базы данных.

4.2. Безопасность и JWT (Spring Security)

Реализована Stateless-аутентификация на основе **JWT (JSON Web Tokens)**.

- При входе (/auth/sign-in) пользователю выдается токен.
- Фильтр JwtRequestFilter перехватывает запросы, валидирует токен и устанавливает контекст безопасности.
- Пароли хранятся в зашифрованном виде (BCryptPasswordEncoder).

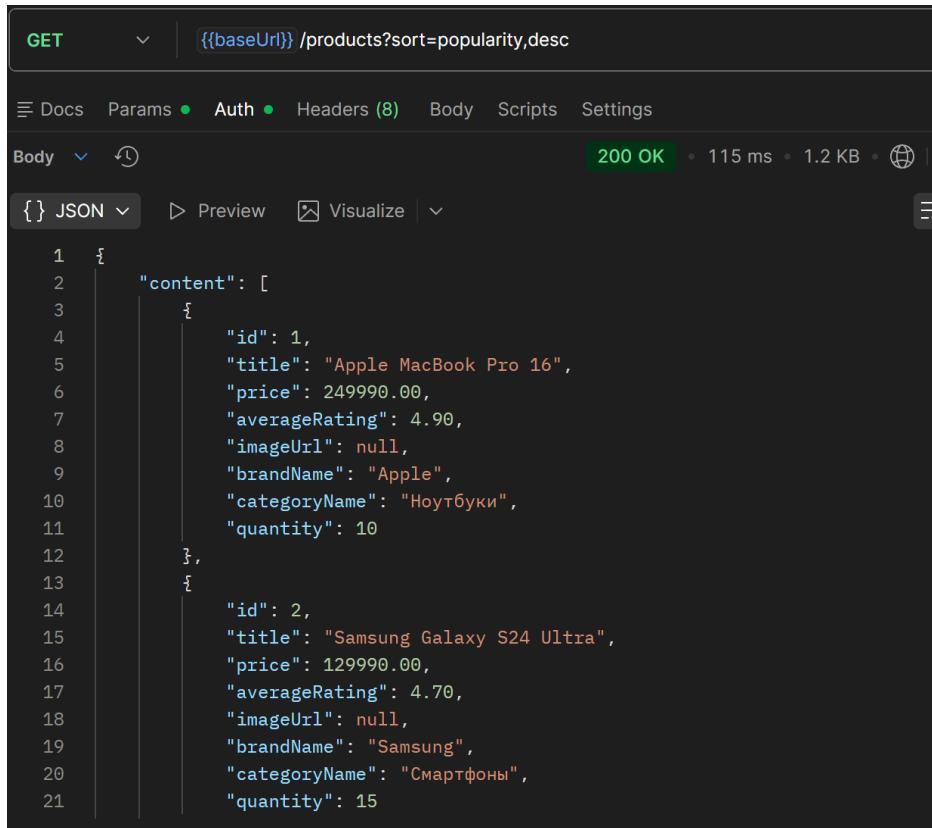
4.3. Каталог товаров и фильтрация

Реализован гибкий поиск товаров с использованием **JPA Specifications**. Это позволяет фильтровать товары по множеству параметров одновременно:

- Поиск по названию/описанию.
- Диапазон цен (min/max).
- Бренд и Категория.
- Наличие на складе.
- Рейтинг.

Класс ProductSpecifications динамически собирает SQL-запрос WHERE условий.

Postman запрос GET /api/v1/products с параметрами фильтрации:



The screenshot shows a Postman interface with a successful API call. The URL is `GET {{baseUrl}} /products?sort=popularity,desc`. The response status is 200 OK, with a duration of 115 ms and a size of 1.2 KB. The response body is a JSON object with two items, each representing a product. The first item is an Apple MacBook Pro 16 with ID 1, price 249990.00, and quantity 10. The second item is a Samsung Galaxy S24 Ultra with ID 2, price 129990.00, and quantity 15.

```
1 {  
2   "content": [  
3     {  
4       "id": 1,  
5       "title": "Apple MacBook Pro 16",  
6       "price": 249990.00,  
7       "averageRating": 4.90,  
8       "imageUrl": null,  
9       "brandName": "Apple",  
10      "categoryName": "Ноутбуки",  
11      "quantity": 10  
12    },  
13    {  
14      "id": 2,  
15      "title": "Samsung Galaxy S24 Ultra",  
16      "price": 129990.00,  
17      "averageRating": 4.70,  
18      "imageUrl": null,  
19      "brandName": "Samsung",  
20      "categoryName": "Смартфоны",  
21      "quantity": 15  
22  }]  
23}
```

Рис. 2. Результат фильтрации товаров через Postman.

4.4. Корзина и кэширование (Redis)

Для высокой производительности корзина пользователя хранится не в PostgreSQL, а в Redis. Это снижает нагрузку на основную БД.

- TTL корзины: 7 дней.
- Сущности: CartDTO, CartItem.

Также настроено кэширование справочных данных (роли, категории) с помощью аннотации `@Cacheable`.

4.5. Заказы и асинхронные уведомления (Kafka)

Реализована событийно-ориентированная архитектура (EDA) для обработки заказов.

1. Пользователь создает заказ (POST /orders).
2. Сервис сохраняет заказ в PostgreSQL.

3. **Producer** отправляет событие OrderPlacedEvent в топик Kafka order-placed.
4. **Consumer** (NotificationService) читает событие и отправляет Email пользователю.

Используется **Avro Schema** для строгой типизации сообщений Kafka.

Схема события (order-placed.avsc):

```
{  
    "type": "record",  
    "name": "OrderPlacedEvent",  
    "namespace": "com.example.techstore.order.event",  
    "fields": [  
        { "name": "orderId", "type": "long" },  
        { "name": "email", "type": "string" },  
        { "name": "firstName", "type": "string" },  
        { "name": "totalPrice", "type": "string" },  
        { "name": "itemsCount", "type": "int" }  
    ]  
}
```

API Документация (Swagger)

Для документирования REST API подключена библиотека **SpringDoc OpenAPI**. По адресу /swagger-ui.html доступен интерактивный интерфейс для тестирования запросов.

Реализованы контроллеры:

- AuthController: Регистрация, вход.
- CatalogController: Товары, бренды, категории.
- CartController: Управление корзиной.
- OrderController: Оформление заказов.
- ReviewController: Отзывы.

Страница Swagger UI с открытым списком контроллеров:

The screenshot displays the Swagger UI interface, which provides a visual representation of the API's endpoints and their details. The API is organized into five main controllers, each with its own section and a list of endpoints.

- user-controller**: Contains seven endpoints:
 - DELETE /api/v1/users/{id}** (red background)
 - GET /api/v1/users/{id}** (blue background)
 - GET /api/v1/users** (light blue background)
 - GET /api/v1/users/{id}/exists** (light blue background)
 - GET /api/v1/users/profile** (light blue background)
 - PATCH /api/v1/users/{id}** (light green background)
 - POST /api/v1/users/{id}/assign-admin** (light green background)
- review-controller**: Contains three endpoints:
 - DELETE /api/v1/reviews/{reviewId}** (red background)
 - GET /api/v1/reviews/product/{productId}** (blue background)
 - POST /api/v1/reviews** (light green background)
- order-controller**: Contains four endpoints:
 - GET /api/v1/orders** (blue background)
 - GET /api/v1/orders/{id}** (blue background)
 - POST /api/v1/orders** (light green background)
 - POST /api/v1/orders/{id}/pay** (light green background)
- favorite-controller**: Contains two endpoints:
 - GET /api/v1/favorites** (blue background)
 - POST /api/v1/favorites/{productId}** (light green background)
- comparison-controller**: Contains three endpoints:
 - DELETE /api/v1/comparison/clear** (red background)
 - GET /api/v1/comparison** (blue background)
 - POST /api/v1/comparison/{productId}** (light green background)

cart-controller

- DELETE** /api/v1/cart/remove/{productId}
- DELETE** /api/v1/cart/clear
- GET** /api/v1/cart
- PATCH** /api/v1/cart/update
- POST** /api/v1/cart/add

auth-controller

- POST** /api/v1/auth/sign-up
- POST** /api/v1/auth/sign-in

catalog-controller

- GET** /api/v1/products
- GET** /api/v1/products/{id}
- GET** /api/v1/categories
- GET** /api/v1/brands

Рис. 3. Swagger UI документация API.

Качество кода и тестирование

В проекте настроены инструменты статического анализа и покрытия тестами:

- Unit-тесты:** Написаны тесты для сервисного слоя (AuthServiceTest, OrderServiceTest, UserServiceTest) с использованием **JUnit 5** и **Mockito**.
- Jacoco:** Плагин для проверки покрытия кода тестами.
- SonarQube:** Подключен плагин для анализа уязвимостей и дублирования.

Результаты прогона тестов в IDE:

The screenshot shows the IntelliJ IDEA test runner interface. The left pane lists test classes and their methods, each with a green checkmark indicating success. The right pane displays the command line output of the test execution. Key logs include:

- Tests passed: 47 of 47 tests – 3 sec 25 ms
- C:\Users\PC-1\.jdks\openjdk-21.0.1\bin\java.exe ...
- дек. 02, 2025 6:17:56 PM org.junit.platform.core.LauncherConfigurat...
WARNING: Discovered 2 'junit-platform.properties' configuration files in the...
- дек. 02, 2025 6:17:56 PM org.junit.platform.core.LauncherConfigurat...
WARNING: Discovered 2 'junit-platform.properties' configuration files in the...
- WARNING: A Java agent has been loaded dynamically (C:\Users\PC-1\.m2\repository...
WARNING: If a serviceability tool is in use, please run with -XX:+EnableDyna...
- WARNING: If a serviceability tool is not in use, please run with -Djdk.instru...
- WARNING: Dynamic loading of agents will be disallowed by default in a future...
- OpenJDK 64-Bit Server VM warning: Sharing is only supported for boot loader...
- 18:17:59.572 [main] INFO org.springframework.test.context.support.Annotation...
- 18:17:59.802 [main] INFO org.springframework.boot.test.context.SpringBootTest...
- Spring Boot :: (v3.3.5)
- 2025-12-02T18:18:00.310+03:00 INFO 3668 --- [tech-store] [main] o...

Рис. 4. Успешное прохождение Unit-тестов.

Развертывание (Docker)

Для контейнеризации приложения создан Dockerfile (Multistage build) и docker-compose.yml.

Среда включает в себя:

- backend-app: Приложение (порт 8080).
- postgres-main: База данных.
- redis: Кэш.
- zookeeper & kafka: Брокер сообщений.
- schema-registry: Регистр схем Avro.
- kafka-ui: Интерфейс для управления Kafka.

Вывод команды docker ps:

CONTAINER ID	IMAGE	NAMES	COMMAND	CREATED	STATUS	PORTS
bcb10ed6009a	provectuslabs/kafka-ui:latest	kafka-ui	"/bin/sh -c 'java --..."	21 minutes ago	Up 21 minutes	0.0.0.0:
e721e7420832	90jer/tech-store:latest		"java -cp @/app/jib-..."	21 minutes ago	Up 21 minutes	0.0.0.0:
8080->8080/tcp		project12a-backend-bimbimbambam-backend-app-1				
b4a5c527d396	confluentinc/cp-schema-registry:7.3.0	schema-registry	"/etc/confluent/dock..."	21 minutes ago	Up 21 minutes	0.0.0.0:
8081->8081/tcp						
970c49e1d3a0	confluentinc/cp-kafka:7.3.0	kafka	"/etc/confluent/dock..."	21 minutes ago	Up 21 minutes	0.0.0.0:
9092->9092/tcp, 0.0.0.0:29092->29092/tcp	postgres:16	postgres-main	"docker-entrypoint.s..."	21 minutes ago	Up 21 minutes (healthy)	0.0.0.0:
21e40af8ed3f						
5433->5432/tcp	redis:alpine	redis	"docker-entrypoint.s..."	21 minutes ago	Up 21 minutes	0.0.0.0:
6379->6379/tcp		zookeeper	"/etc/confluent/dock..."	21 minutes ago	Up 21 minutes	2888/tcp
ca911la25430						
, 0.0.0.0:2181->2181/tcp, 3888/tcp						

Рис. 5. Запущенные контейнеры приложения.