|  |  |  |
| --- | --- | --- |
|  |  |  |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |

Институт Информационных технологий

Кафедра Инструментального и прикладного программного обеспечения

**ПРАКТИЧЕСКИЕ РАБОТЫ №5-7**

по дисциплине «Разработка серверных частей интернет-ресурсов»

**Студент группы** ИКБО-01-21 Кузнецов А.А.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись студента)

**Руководитель практической работы** Благирев М.М.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись руководителя)

Работа представлена «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_ 2023 г.

Допущен к работе «\_\_\_ » \_\_\_\_\_\_\_\_\_\_\_ 2023 г.

Москва 2023

# ПРАКТИЧЕСКАЯ РАБОТА №5

# ЦЕЛЬ РАБОТЫ

**Задание**

Студенту предлагается создать свое серверное CRUD приложение. Приложение должно стать основанием для создания маркетплейса. В работе должны быть реализованы такие модели:

1. Book – модель, которая описывает сущность книги, с полями: автор, номер продавца, тип продукта (подразумевается электроника, книги, сантехника и т. п.), стоимость, название.
2. Client – модель, которая описывает сущность клиента сайта. У модели должны быть поля: имя, электронная почта, логин, пароль.
3. Telephone – модель, описывающая сущность телефона. Обязательные поля: производитель, объем аккумулятора, номер продавца, тип продукта (подразумевается электроника, книги, сантехника и т. п.), стоимость, название.
4. WashingMachine – модель, которая описывает сущность стиральной машины, с такими полями: производитель, объем бака, номер продавца, тип продукта (подразумевается электроника, книги, сантехника и т. п.), стоимость, название.

Так же должны быть контроллеры, которые позволят манипулировать модели (CRUD). Вся настройка приложения должна быть реализована в директории configuration. Приложение должно запускаться с помощью docker-compose.

# ХОД РАБОТЫ

Для создания образа необходимого веб-сервера, был использован Dockerfile, изображенный на рисунке 1.

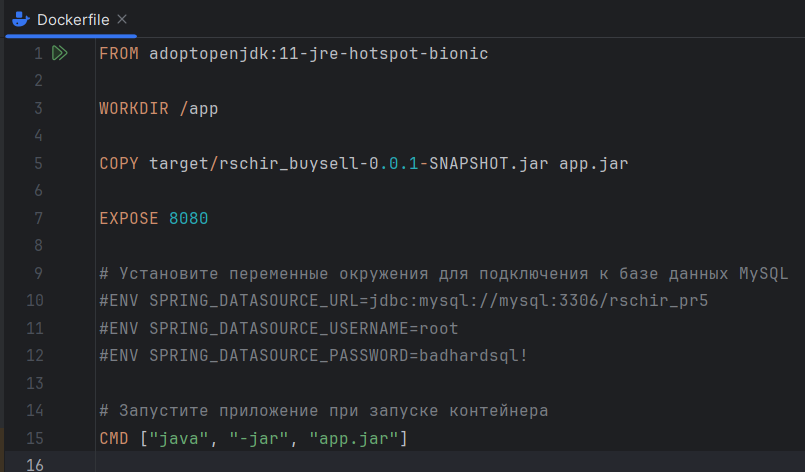


Рисунок 1 – Dockerfile для проекта Buysell

Тут мы устанавливаем 11 версию JDK для корректной работы приложения и копируем содержимое сервера, находящееся в текущей директории, в файловую систему веб-сервера.

Также для проекта был использован следующий docker-compose.yaml, изображенный на рисунке 2.



Рисунок 2 – docker-compose.yaml

Здесь есть два сервиса: веб-сервер и база данных. При этом веб-сервер зависит от базы данных. В настройках сервера мы указываем Dockerfile, который создаст нужный нам образ, а также порты и тома для работы с нужными нам файлами в файловой системе сервера.

Далее на рисунке 3 изображена файловая система со списком моделей в приложении по ТЗ.

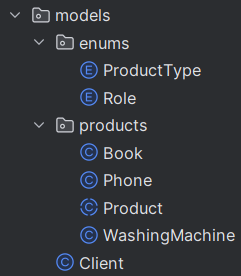


Рисунок 3 – Файловое дерево сущностей в приложении

На листинге 1 изображен абстрактный класс, являющейся сущностью продукта Product.class.

Листинг 1 – Кода класса-модели Product.class

@Entity  
@Table  
@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public abstract class Product {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 @Column  
 private Long id;  
  
 @Column  
 private String name;  
  
 @Column  
 private Integer quantity;  
  
 @Column  
 private Double price;  
  
 @ManyToOne(cascade = CascadeType.*REFRESH*, fetch = FetchType.*LAZY*)  
 @JoinColumn(name = "seller\_id")  
 private Client seller;  
  
 @Column  
 @Enumerated(EnumType.*STRING*)  
 private ProductType productType;  
  
 @OneToMany(cascade = CascadeType.*ALL*, fetch = FetchType.*LAZY*,  
 mappedBy = "product")  
 private List<Image> images = new ArrayList<>();  
 private Long previewImageId;  
  
 public void addImageToProduct(Image *image*) {  
 *image*.setProduct(this);  
 images.add(*image*);  
 }  
  
 public boolean hasPreview() {  
 return previewImageId != null;  
 }  
  
 public abstract String toControllerProductType();  
}

Файловое дерево всех контроллеров для приложения изображена на рисунке 4.

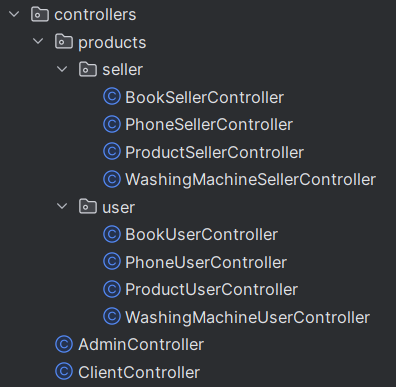


Рисунок 4 – Файловое дерево контроллеров

Пример нескольких эндпоинтов контроллера для работы пользователя с продуктами ProductSellerController.class изображен на листинге 2.

Листинг 2 – Фрагмент кода класса-контроллера ProductSellerController

@Controller  
@RequiredArgsConstructor  
@RequestMapping("/product")  
@PreAuthorize("hasAuthority('ROLE\_SELLER') or hasAuthority('ROLE\_ADMIN')")  
public class ProductSellerController {  
  
 private final ProductService productService;  
  
 @GetMapping("/typeSelect")  
 public String productTypeSelect(Model *model*, Principal *principal*) {  
 List<Product> *products* = productService.getAllProducts();  
 *model*.addAttribute("products", *products*);  
 *model*.addAttribute("user", productService.getClientByPrincipal(*principal*));  
 *model*.addAttribute("types", ProductType.*values*());  
 return "/products/selling/productTypeSelect";  
 }  
  
 @GetMapping("/delete/{id}")  
 public String deleteProduct(@PathVariable Long *id*, @AuthenticationPrincipal Client *client*) {  
 productService.deleteProduct(*id*, *client*);  
 return "redirect:/";  
 }  
}

Как мы видим к данному контроллеру могут обращаться только пользователи с ролями продавца и администратора. Аналогично и для других классов-контроллеров.

Файловая система репозиториев изображена на рисунке 5.



Рисунок 5 – Файловое дерево репозиториев

Файловая система классов-сервисов изображена на рисунке 6.

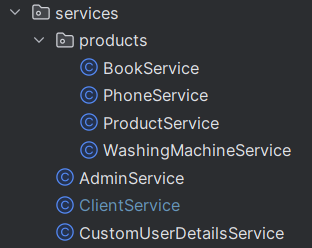


Рисунок 6 – Файловое дерево классов-сервисов

# ТЕСТИРОВАНИЕ

При попытке неавторизованными обратиться к какому-либо энд-поинту нас отправляет на страницу авторизации, изображенную на рисунке 7.



Рисунок 7 – Страница авторизации

Но для начала необходимо авторизироваться, как изображено на рисунке 8.

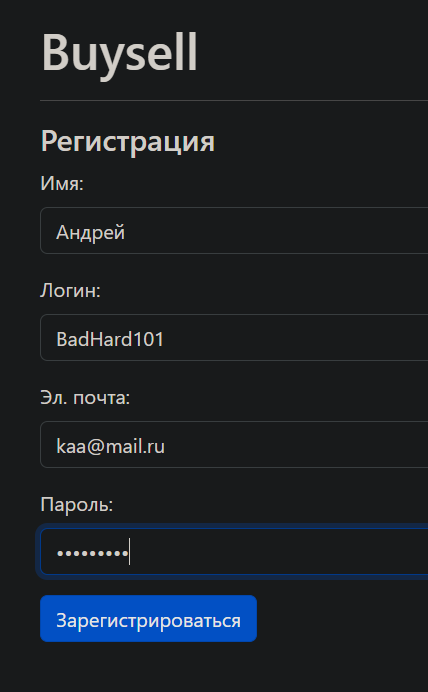


Рисунок 8 – Регистрация аккаунта

После авторизации мы попадаем на главную страницу приложения, как изображено на рисунке 9.

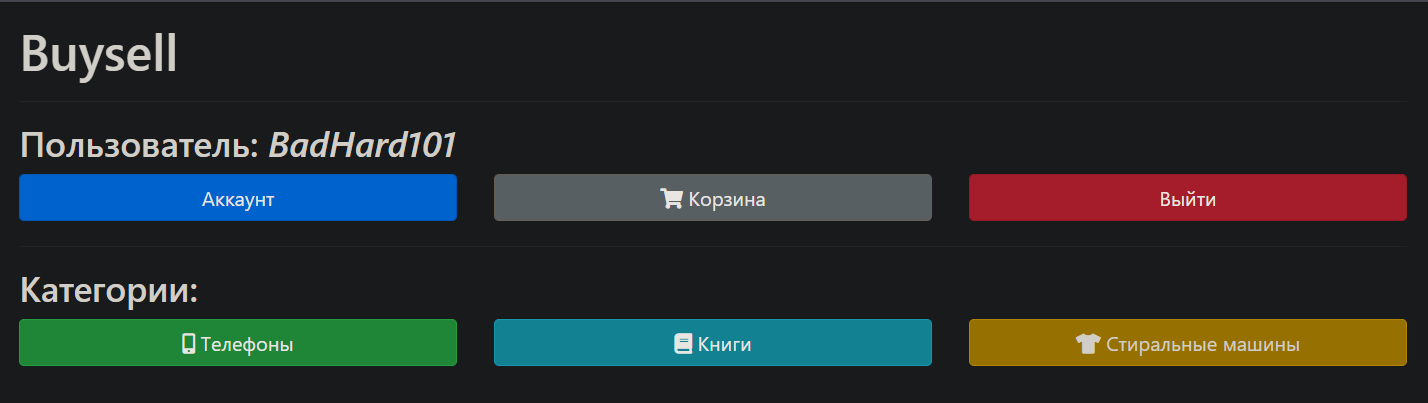


Рисунок 9 – Главная страница

При выборе какой-то категории товаров мы видим товары на продажу, которые уже есть от других пользователей в базе данных, как изображено на рисунке 10.

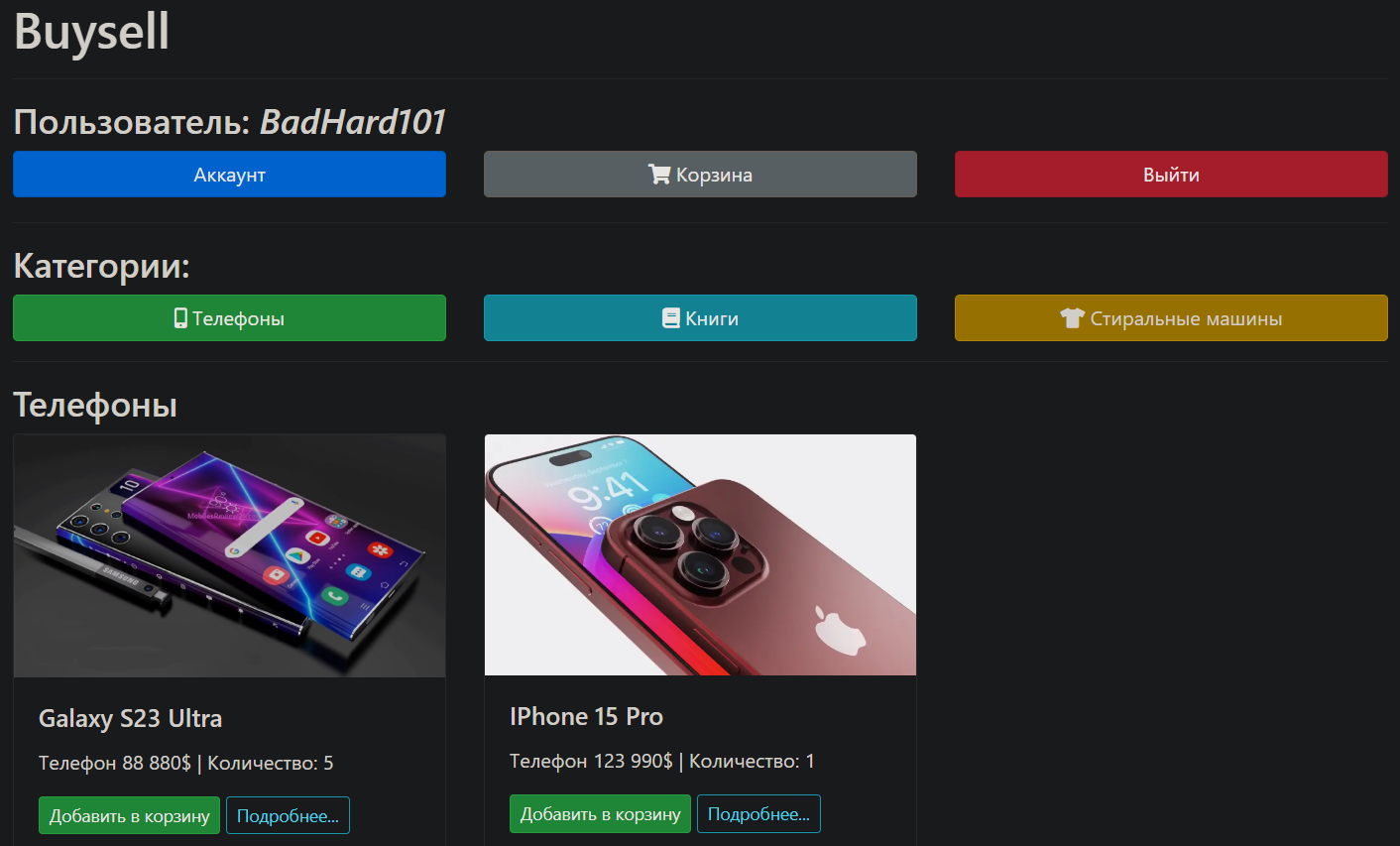


Рисунок 10 – Товары категории «Телефоны»

Чтобы выставить какой-то свой телефон на продажу нам необходимо получить роль продавца. Пока что это может сделать только администратор. Зайдем за администратора и выдадим роль. Главная страница администратора изображена на рисунке 11.

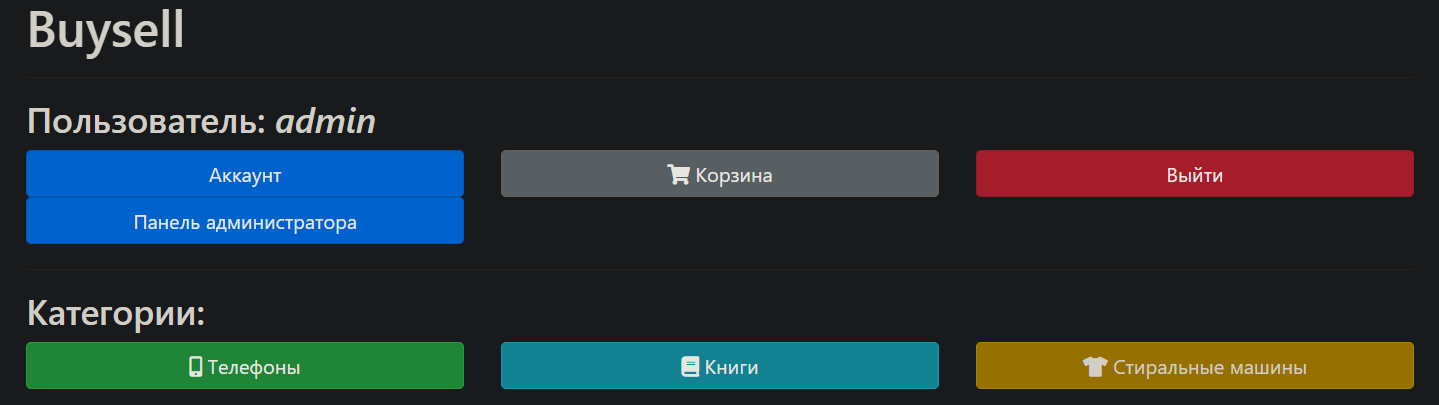


Рисунок 11 – Главная страница администратора

Как мы видим у него есть панель администратора. Войдя в нее, попадем в панель управления пользователями и увидим то, что изображено на рисунке 12.

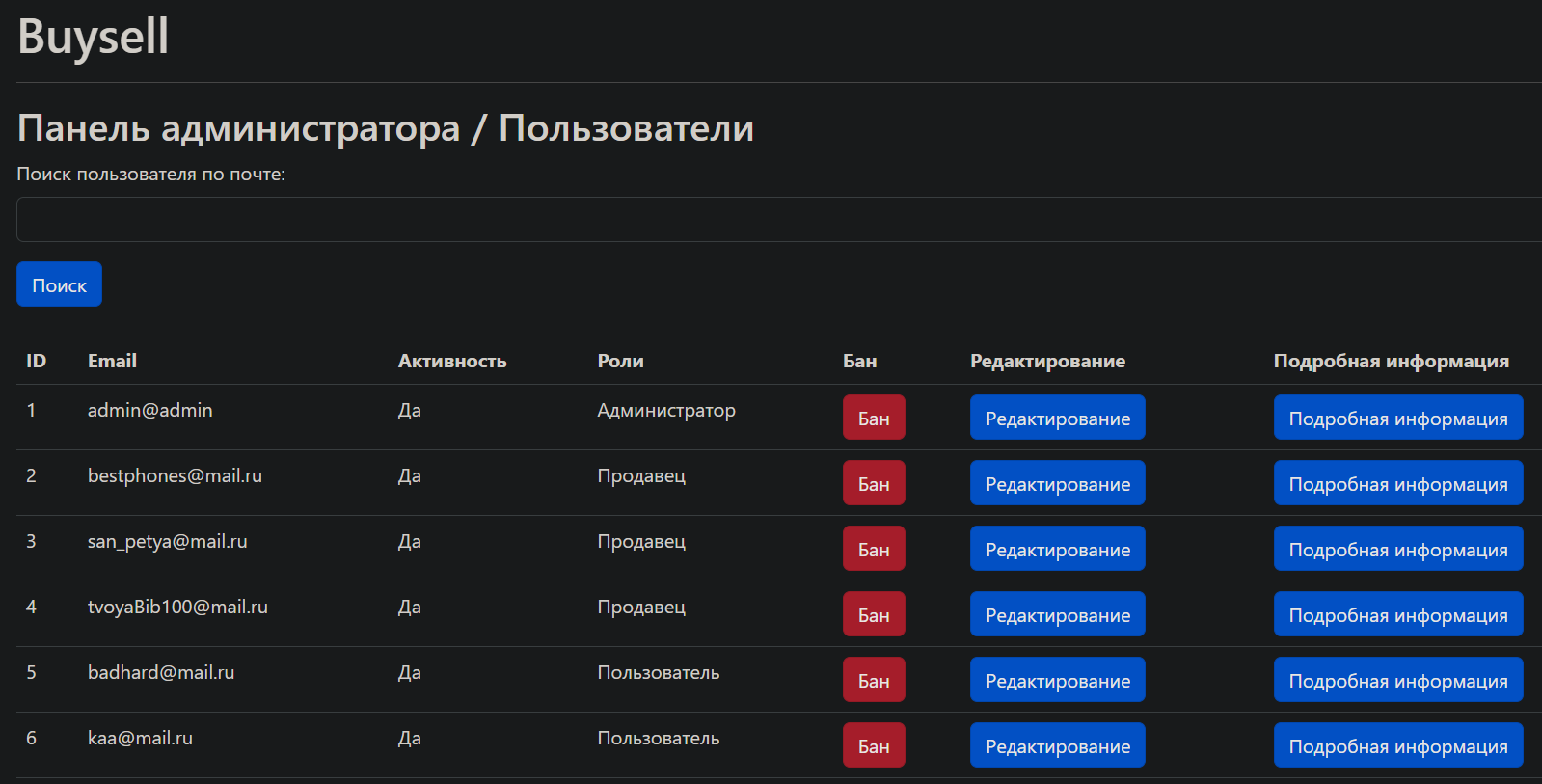


Рисунок 12 – Панель управления пользователями

Выдадим роль продавца для пользователя с почтой kaa@mail.ru, как на рисунке 13.

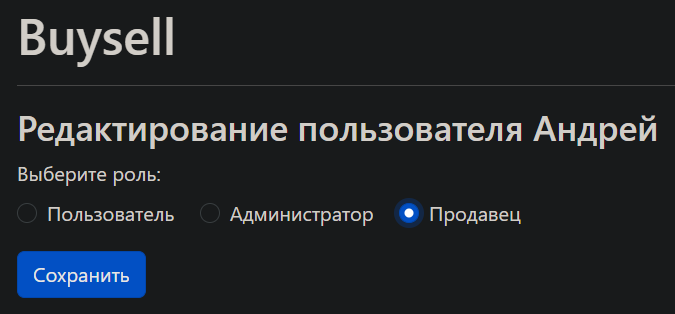


Рисунок 13 – Изменяем роль пользователя

Войдем обратно и на рисунке 14 увидим, что теперь нам стала доступна кнопка для продажи.

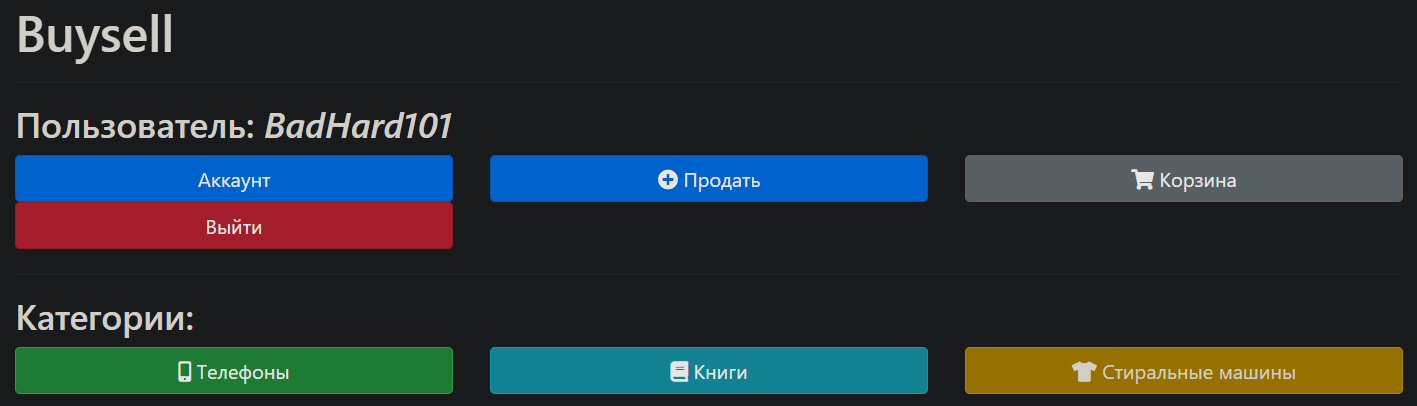


Рисунок 14 – Обновленная главная страница

Выставим на продажу наш телефон как изображено на рисунке 15.

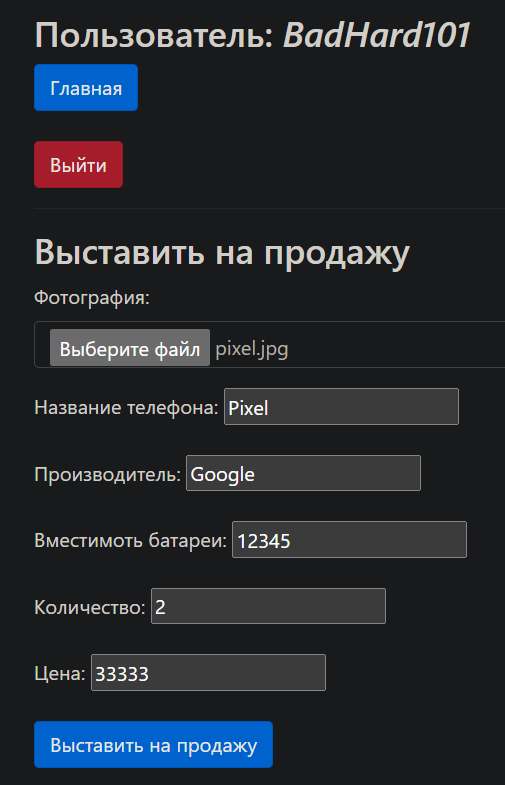


Рисунок 15 – Регистрация телефона на продажу

Теперь увидим, что наш телефон теперь в каталоге телефонов, как на рисунке 16.

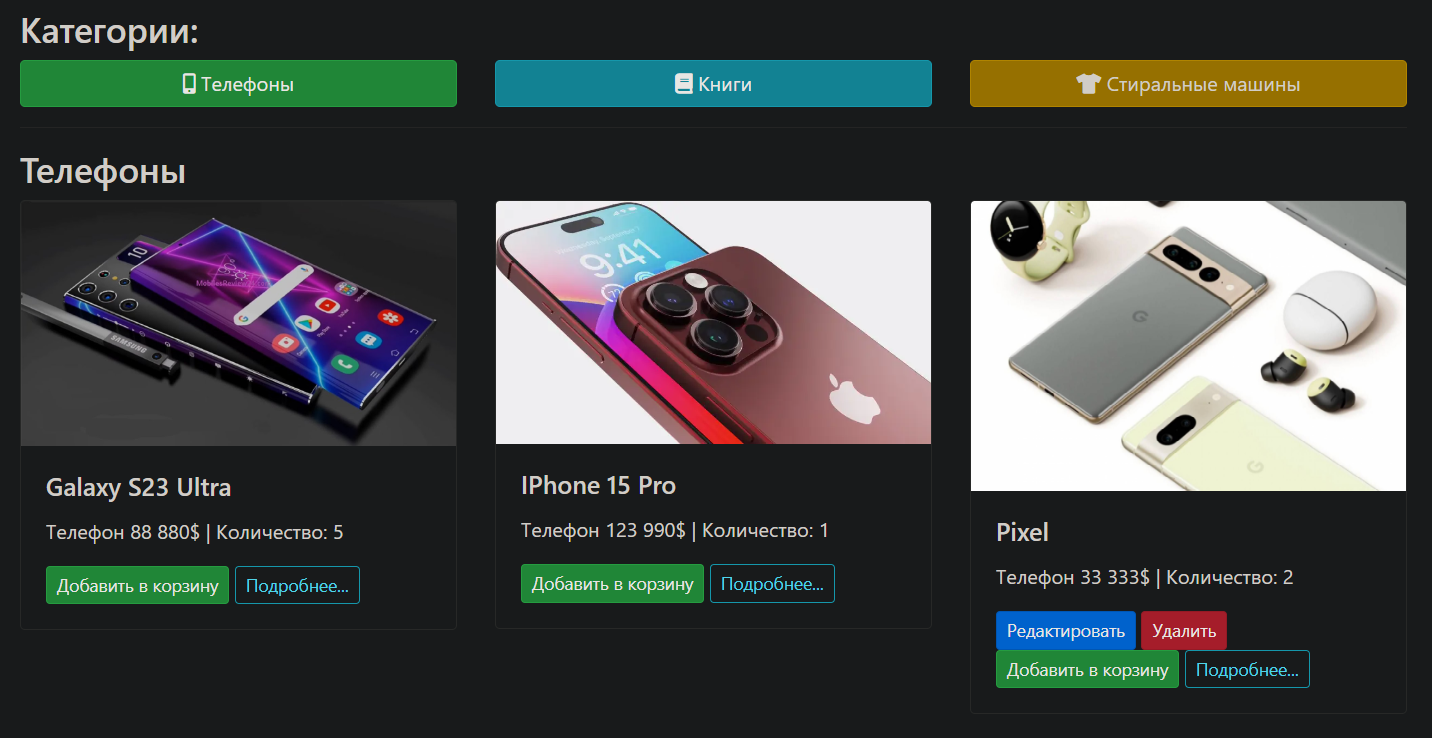


Рисунок 16 – Добавленный телефон в каталоге

Также, так как мы владелец телефона, нам доступны функции редактирования и удаления товара. Попробуем изменить цену телефона, как на рисунке 17.

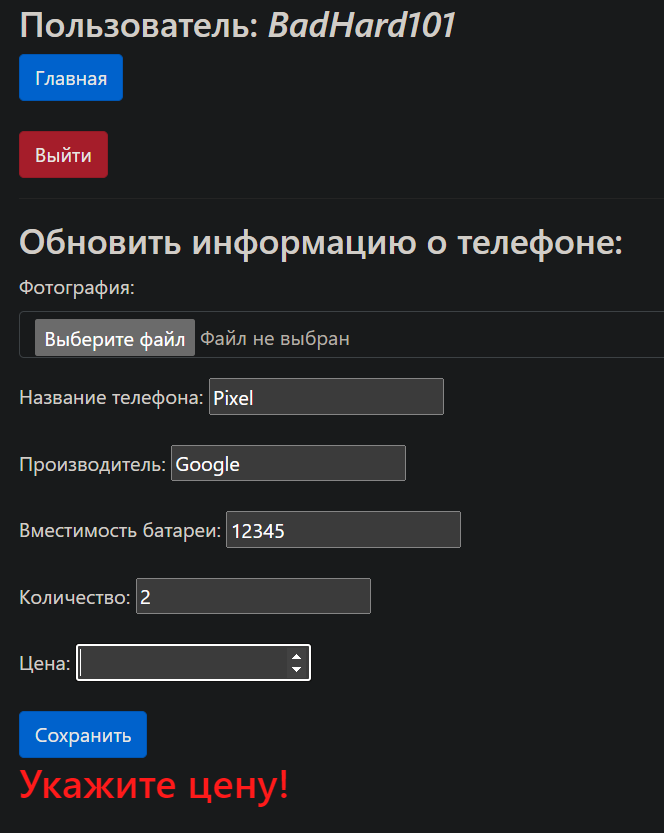


Рисунок 17 – Некорректное изменение цены товара

Если мы забыли указать какое-либо поле или оно будет введено неправильно, приложения уведомит об этом нам, так как в приложении есть методы валидации. Попробуем указать цену, рисунок 18.

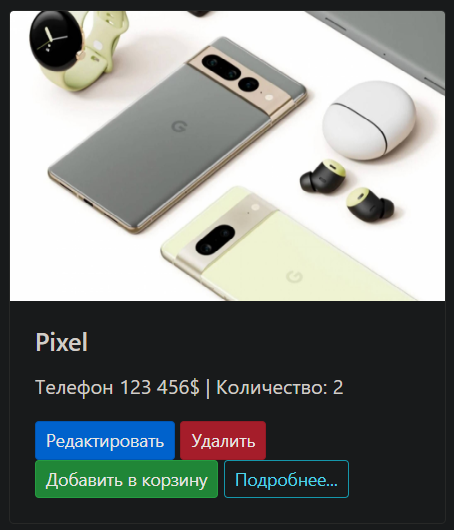


Рисунок 18 – Изменение произошло успешно

Попробуем удалить товар, результат изображен на рисунке 19.

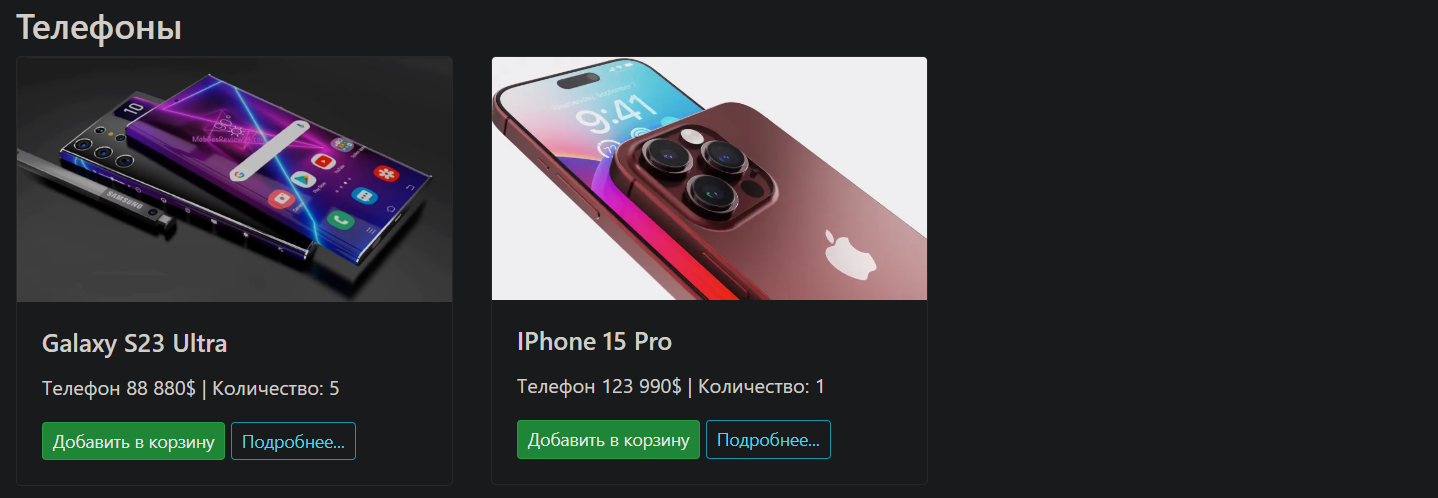


Рисунок 19 – Результат удаления

Мы убедились, что все функции работают корректно. Аналогично и с другими товарами. Категории «Книги» и «Стиральные машины» изображены на рисунках 20-21 соответственно.

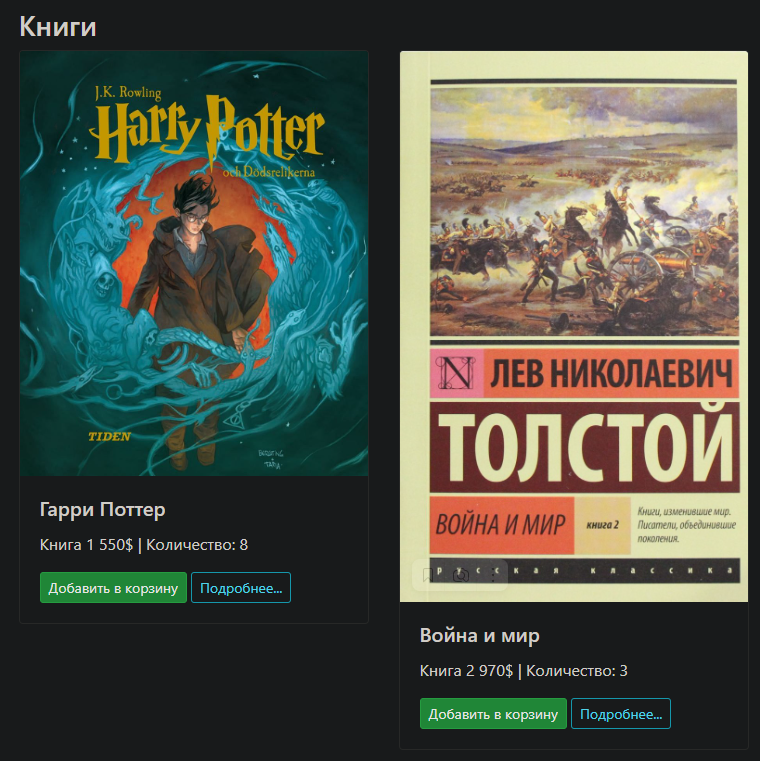


Рисунок 20 – Каталог книг

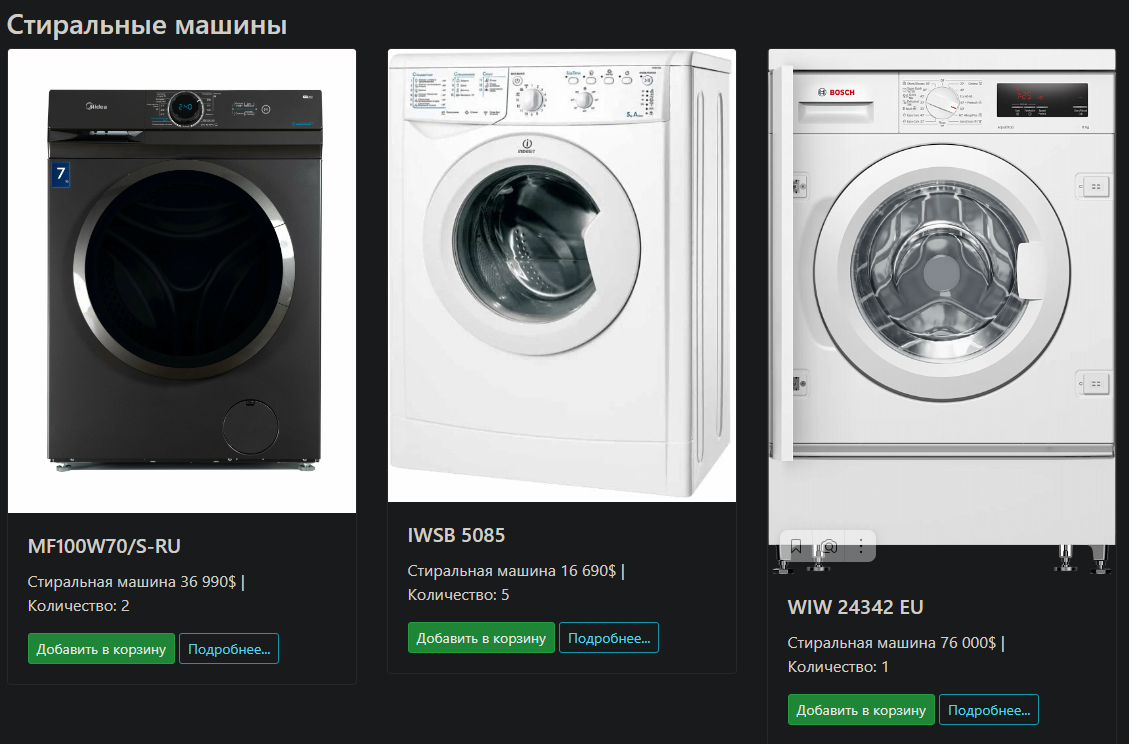


Рисунок 21 – Каталог стиральных машин

# ВЫВОД

Таким образом было создано CRUD приложение на языке Java при помощи Spring фреймворка и его дополнительных модулей, были реализованы и успешно проверены методы добавления, редактирования, получения и удаления всех категорий товаров. Проект был развернут в Docker при помощи docker-compose и Dockerfile. Реализовано разграничение ролей при помощи Spring Security, так что методы приложения успешно работают с ограничениями безопасности.

# ОТВЕТЫ НА ВОПРОСЫ

1. Что такое JDBC и JPA, в чем отличия.

JDBC (Java Database Connectivity): Это API для взаимодействия с базами данных из языка Java. Он предоставляет набор классов и методов для выполнения SQL-запросов и манипулирования данными в базах данных.

JPA (Java Persistence API): Это стандарт Java EE для управления реляционными данными в приложениях, использующих платформу Java. Он предоставляет более высокоуровневый и объектно-ориентированный подход к взаимодействию с базами данных.

1. Основные интерфейсы для реализации Spring Data.

Основные интерфейсы в Spring Data включают CrudRepository, PagingAndSortingRepository, и JpaRepository. Они предоставляют стандартные операции для работы с базами данных, такие как создание, чтение, обновление и удаление (CRUD).

1. Аннотации @Entity, @Table. В чем разница.

@Entity: Эта аннотация помечает класс как сущность JPA, что означает, что объекты этого класса могут быть сохранены в базе данных.

@Table: Эта аннотация позволяет настраивать детали таблицы базы данных, такие как ее имя и другие характеристики.

1. Что такое Lombok. Приведите несколько примеров использования.

Lombok - это библиотека для уменьшения шаблонного кода в проектах на Java. Она добавляет аннотации для автоматической генерации методов, конструкторов, геттеров и сеттеров, улучшая читаемость и поддерживаемость кода. Примеры: @Data, @Getter, @Setter, @NoArgsConstructor, @AllArgsConstructor.

1. Назовите способы инициализации бинов в Java.

Использование XML-конфигурации.

Использование аннотаций в коде (например, @Component, @Service, @Repository).

Использование JavaConfig с классами конфигурации.

1. Что такое миграции и как они работаю в библиотеке Flyway.

Миграции: Это способ эволюции базы данных путем применения последовательных изменений с сохранением версионности.

Flyway: Это библиотека для управления миграциями базы данных. Она позволяет автоматизировать и упростить процесс изменения схемы базы данных в соответствии с изменениями в приложении.

1. Что такое сущность и модуль. В чем разница.

Сущность (Entity): В контексте JPA, это объект, представляющий данные в базе данных. Он соответствует записи в таблице базы данных.

Модуль: Это более широкий термин, обозначающий логически связанный блок кода в проекте. Модуль может включать в себя несколько сущностей, сервисов и других компонентов, не обязательно связанных с базой данных.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Документация Java: <https://docs.oracle.com/en/java/> (дата обращения: 15.11.2023). – Текст: электронный.

2. Документация Spring: <https://spring.io/quickstart> (дата обращения: 15.11.2023).

3. Методические указания по выполнению практической работы: <https://online-edu.mirea.ru/mod/resource/view.php?id=517436> (дата обращения: 15.11.2023).

4. Официальная документация докера: <https://docs.docker.com/> (дата обращения: 15.11.2023).

5. Более сложная и подробная статья про докер: <https://habr.com/ru/post/277699/> (дата обращения: 15.11.2023).

# ПРАКТИЧЕСКАЯ РАБОТА №6

# Задание

Студенту предлагается реализовать бизнес логику ко второй практике по Java. Нужно добавить к существующему приложению реализацию «корзины».

Пользователь должен иметь такие возможности:

1. Добавить товар в корзину.

2. Удалить товар из корзины.

3. Изменить количество товара в корзине.

4. Посмотреть всю корзину.

5. Оформить заказ и очистить корзину.

При добавлении товара в корзину должна быть проверка, что товар есть в наличии. Так же обработать случай, что товар добавили в корзину, после товар закончился на складе и оформить заказ невозможно.

Сервис должен с помощью docker-compose. Должно быть использовано паттерны проектирования: Чистая архитектура, MVC.

# ХОД РАБОТЫ

Для реализации возможности работы пользователя с корзиной была добавлена модель корзины, содержащая информацию о клиенте, массив продуктов и метка, показывающая является ли корзина сейчас активной, а также методы для работы с массивом продуктов и подсчет стоимости всех продуктов в корзине. Модель корзины изображена на листинге 4.

Листинг 4 – Код класса-модели ShoppingCart

@Entity  
@Table  
@Data  
public class ShoppingCart {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
  
 @OneToOne  
 @JoinColumn(name = "client")  
 private Client client;  
  
 @Column(name = "is\_active")  
 private boolean active;  
  
 @ElementCollection  
 @CollectionTable(name = "cart\_items", joinColumns = @JoinColumn(name = "cart\_id"))  
 private Map<Product, Integer> items = new HashMap<>();  
  
 public void addItem(Product *product*) {  
 items.put(*product*, items.getOrDefault(*product*, 0) + 1);  
 }  
  
 public void removeItem(Product *product*) {  
 int *quantity* = items.getOrDefault(*product*, 0);  
 if (*quantity* > 1) {  
 items.put(*product*, *quantity* - 1);  
 } else {  
 items.remove(*product*);  
 }  
 }  
  
 public void deleteItem(Product *product*) {  
 items.remove(*product*);  
 }  
  
 public Double calculateTotal() {  
 Double *total* = 0.0;  
 for (Map.Entry<Product, Integer> *entry* : items.entrySet()) {  
 Product *product* = *entry*.getKey();  
 Integer *quantity* = *entry*.getValue();  
 *total* += *product*.getPrice() \* *quantity*;  
 }  
 return *total*;  
 }  
}

Соответственно и были добавлены энд-поинты в класс ProductUserController.class, изображенный на листингах 4-5, для работы с нашей созданной корзиной,.

Листинг 4 – Код класса-контроллера ProductUserController часть 1

@Controller  
@RequiredArgsConstructor  
@RequestMapping("/product")  
public class ProductUserController {  
  
 private final ProductService productService;  
  
 @GetMapping("/addToCart/{id}")  
 public String addProductToCart(@PathVariable Long *id*, @AuthenticationPrincipal Client *client*, Model *model*) {  
 String *st* = productService.addProductToCart(*id*, *client*);  
 if (!*st*.equals("Success")) *model*.addAttribute("errorMessage", *st*);  
 return "redirect:/product/shoppingCart";  
 }   
  
@GetMapping("/shoppingCart")  
public String showShoppingCart(@AuthenticationPrincipal Client *client*, Model *model*) {  
 ShoppingCart *cart* = productService.getOrCreateShoppingCartByClient(*client*);  
 Map<Product, Integer> *items* = *cart*.getItems();  
 // Преобразование Map в список для сортировки  
 List<Map.Entry<Product, Integer>> *sortedItems* = new ArrayList<>(*items*.entrySet());  
 // Сортировка списка. Пример: сортировка по имени продукта  
 *sortedItems*.sort(Comparator.*comparing*(*entry* -> *entry*.getKey().getPrice()));  
  
 // Теперь, когда у вас есть отсортированный список, добавьте его в модель  
 *model*.addAttribute("user", *client*);  
 *model*.addAttribute("items", *sortedItems*); // Используйте отсортированный список здесь  
 *model*.addAttribute("cart", *cart*); // Используйте отсортированный список здесь  
  
 return "user/shoppingCart";  
}

Листинг 5 – Код класса-контроллера ProductUserController часть 2

@GetMapping("/shoppingCart/addItem/{id}")  
 public String addItemToShoppingCart(@PathVariable Long *id*, @AuthenticationPrincipal Client *client*, Model *model*) {  
 String *st* = productService.addProductToCart(*id*, *client*);  
 if (!*st*.equals("Success")) *model*.addAttribute("errorMessage", *st*);  
  
 ShoppingCart *cart* = productService.getOrCreateShoppingCartByClient(*client*);  
 Map<Product, Integer> *items* = *cart*.getItems();  
 List<Map.Entry<Product, Integer>> *sortedItems* = new ArrayList<>(*items*.entrySet());  
 *sortedItems*.sort(Comparator.*comparing*(*entry* -> *entry*.getKey().getPrice()));  
  
 *model*.addAttribute("user", *client*);  
 *model*.addAttribute("items", *sortedItems*);   
 *model*.addAttribute("cart", *cart*   
 return "user/shoppingCart";  
 }  
  
 @GetMapping("/shoppingCart/removeItem/{id}")  
 public String removeItemFromShoppingCart(@PathVariable Long *id*, @AuthenticationPrincipal Client *client*, Model *model*) {  
 productService.removeProductToCart(*id*, *client*);  
 return "redirect:/product/shoppingCart";  
 }  
  
 @GetMapping("/shoppingCart/checkout")  
 public String checkoutShoppingCart(@AuthenticationPrincipal Client *client*, Model *model*) {  
 String *st* = productService.checkoutShoppingCart(*client*);  
 if (!*st*.equals("Success")) *model*.addAttribute("errorMessage", *st*);  
 else {  
 *model*.addAttribute("successLabel", "Вы успешно оформили заказ!");  
 *model*.addAttribute("user", *client*);  
 return "user/orderPage";  
 }  
  
 ShoppingCart *cart* = productService.getOrCreateShoppingCartByClient(*client*);  
 Map<Product, Integer> *items* = *cart*.getItems();  
 List<Map.Entry<Product, Integer>> *sortedItems* = new ArrayList<>(*items*.entrySet());  
 *sortedItems*.sort(Comparator.*comparing*(*entry* -> *entry*.getKey().getPrice()));  
  
 *model*.addAttribute("user", *client*);  
 *model*.addAttribute("items", *sortedItems*);   
 *model*.addAttribute("cart", *cart*);   
  
 return "user/shoppingCart";  
 }  
}

И конечно же был добавлен репозиторий для работы с корзинами пользователей из базы данных.

# ТЕСТИРОВАНИЕ

Добавим несколько товаров из каталога и посмотрим их в корзине, рисунок 22.

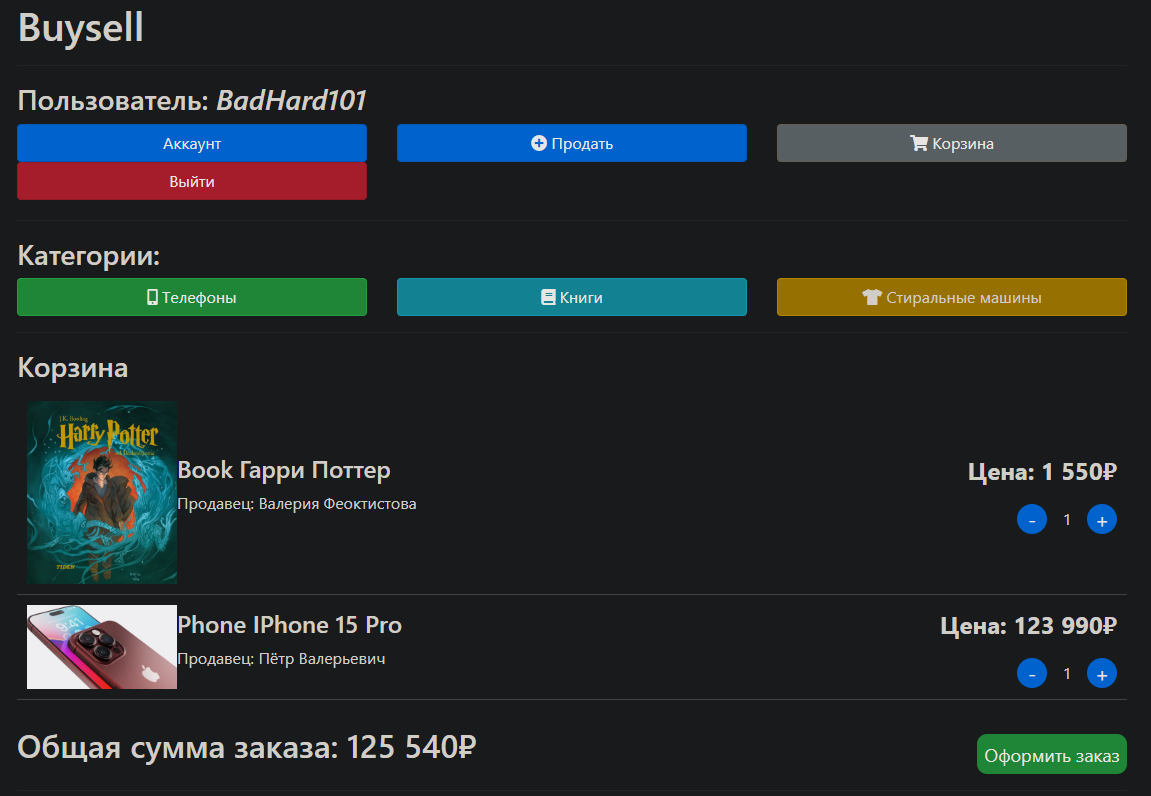


Рисунок 22 – Страница корзины

Мы добавили одну книгу и один телефон. Попробуем добавить еще одну книгу и посмотрим изменится ли цена, рисунок 23.

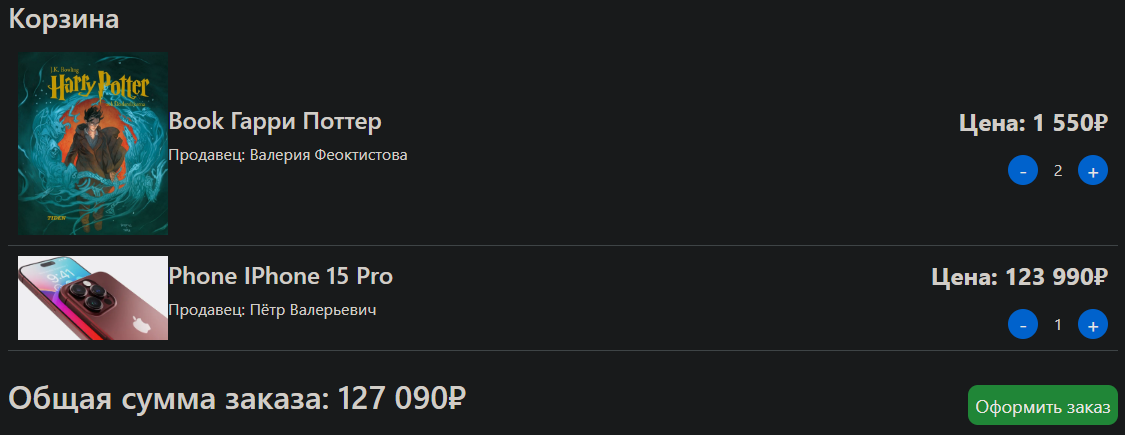


Рисунок 23 – Изменение кол-ва товаров и цены

Как видим кол-во товаров и цена действительно изменились. Но такой телефон на продажу был всего один, попробуем его добавить, рисунок 24.

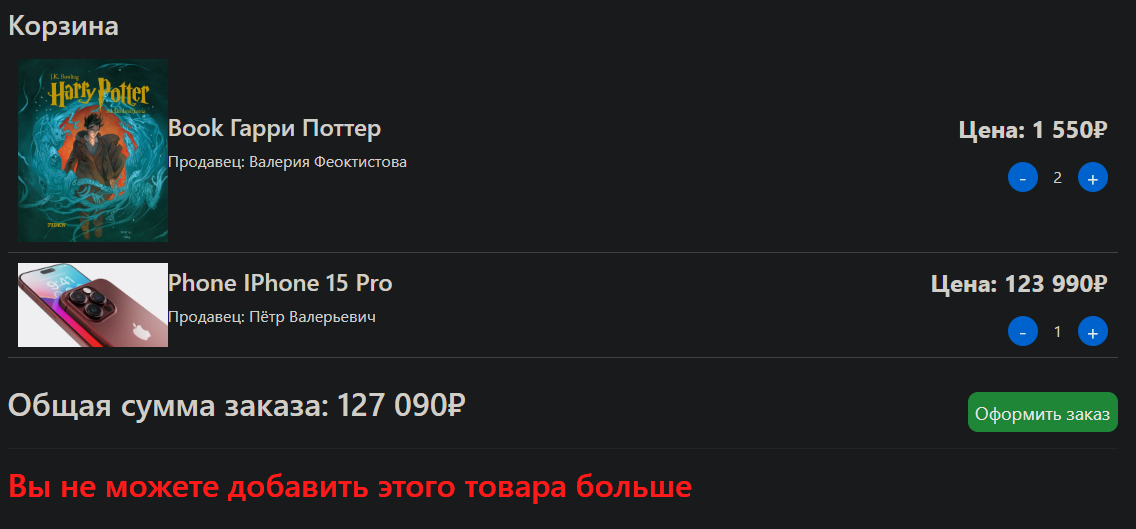


Рисунок 24 – Добавление товара кол-ва больше, чем есть в наличии

Как мы видим, приложение не позволило нам это сделать. Проверим, а можем ли мы оформить заказ если другой пользователь закажет книги, пока они у нас были в корзине. Оформим заказ на 7 книг из 8 в наличии, как на рисунках 25-26.

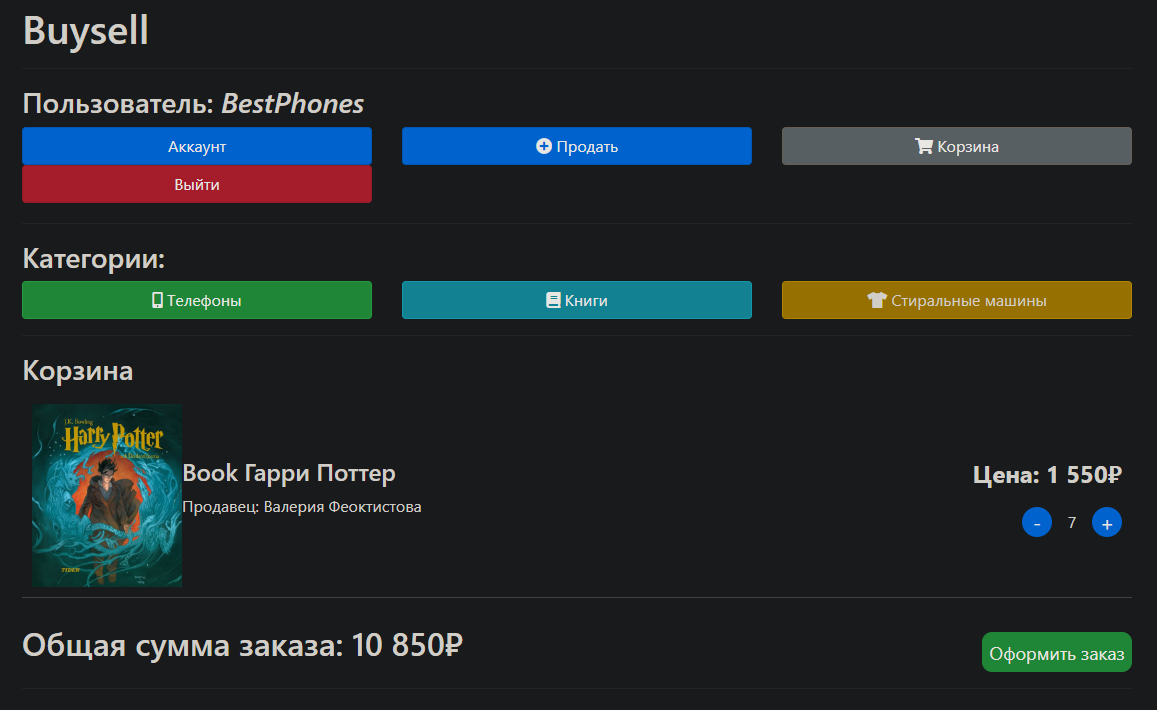


Рисунок 25 – Оформление заказа на книги за другого пользователя часть 1

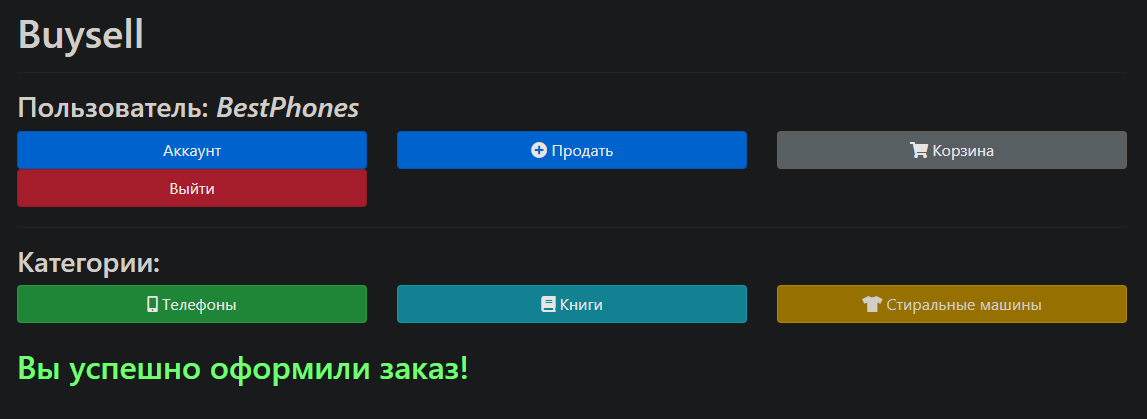


Рисунок 26 – Оформление заказа на книги за другого пользователя часть 2

Мы успешно сделали заказ. Как мы знаем в наличии осталось всего одна такая книга, рисунок 27.

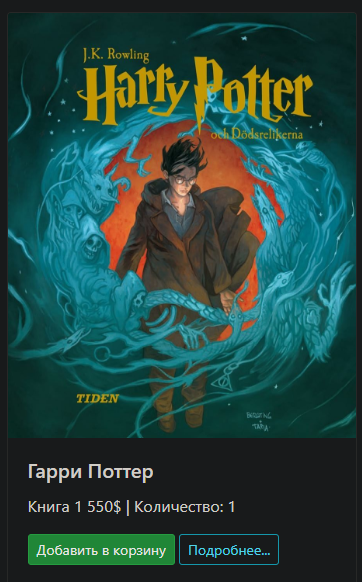


Рисунок 28 – Кол-во книг в наличии

Попробуем сделать заказ с двумя книгами в корзине, рисунок 29.

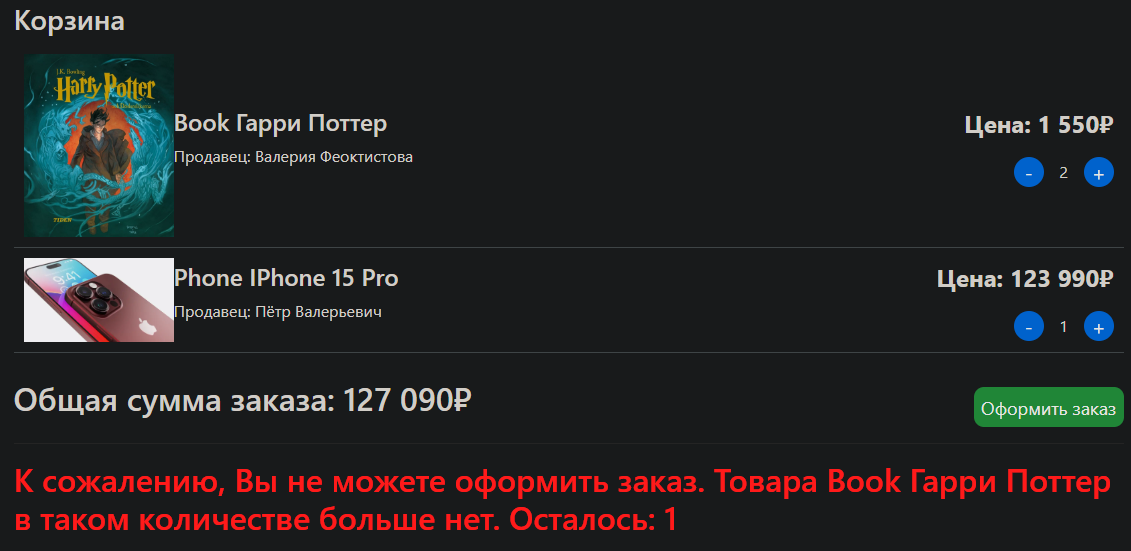


Рисунок 29 – Заказ книг больше чем в наличии

Даже с небольшой хитростью, но приложение не позволило нам заказать книг больше, чем есть в наличии.

Исходя из тестирования, можно сказать, что все методы работают корректно.

# Вывод

В приложении была реализована корзина, в которую можно собирать товары и оформлять заказ. Корзина прошла все тесты на наличие ошибок в одновременной работе с различными пользователями. Репозиторий проекта доступен по ссылке: <https://github.com/BadHard101/Buysell>

# ОТВЕТЫ НА ВОПРОСЫ

1. Аннотация Service.

@Service - это аннотация в Spring Framework, которая помечает класс как службу (service). Сервисы в Spring используются для реализации бизнес-логики и обеспечения сервисов приложения.

1. Паттерн MVC.

MVC (Model-View-Controller): Это архитектурный паттерн, разделяющий приложение на три основных компонента: Model (модель), View (представление) и Controller (контроллер). Этот подход упрощает управление, обновление и расширение кода приложения.

1. Паттерн «Чистая архитектура».

Этот паттерн, предложенный Робертом Мартином, ставит акцент на разделении приложения на независимые слои, где внутренние слои не зависят от внешних. В чистой архитектуре выделяются такие слои, как Entities, Use Cases, Interface Adapters и Frameworks & Drivers. Этот подход способствует независимости слоев и улучшает тестируемость, поддерживаемость и расширяемость кода.

1. Описать все слои RestFull приложения и описать зачем каждый из них.

Controller Layer (Контроллер): Обрабатывает входящие HTTP-запросы, управляет потоком управления и делегирует запросы соответствующим слоям.

Service Layer (Сервис): Содержит бизнес-логику приложения. Отвечает за обработку запросов, выполняет необходимые операции и взаимодействует с репозиториями или другими сервисами.

Repository Layer (Репозиторий): Отвечает за взаимодействие с базой данных или другими источниками данных. Выполняет операции чтения и записи.

Model Layer (Модель): Представляет объекты данных, используемые в приложении. Эти объекты могут быть сущностями базы данных, DTO (Data Transfer Objects) и другими.

View Layer (Представление): В контексте RESTful приложения, представление часто означает формат ответа, например, JSON или XML. Может также включать в себя клиентскую сторону для веб-приложений.

Security Layer (Слой безопасности): Обеспечивает безопасность приложения, включая аутентификацию, авторизацию и управление доступом к ресурсам.

Exception Handling Layer (Слой обработки исключений): Отвечает за обработку исключений, возникающих во время выполнения приложения, и предоставляет клиентам информацию об ошибках в понятной форме.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Документация Spring Security: <https://spring.io/projects/spring-security> (дата обращения: 20.11.2023). – Текст: электронный.

2. Документация Java: <https://docs.oracle.com/en/java/> (дата обращения: 20.11.2023). – Текст: электронный.

3. Документация Spring: <https://spring.io/quickstart> (дата обращения: 20.11.2023).

4. Методические указания по выполнению практической работы: <https://online-edu.mirea.ru/mod/resource/view.php?id=517436> (дата обращения: 20.11.2023).

5. Официальная документация докера: <https://docs.docker.com/> (дата обращения: 20.11.2023).

6. Более сложная и подробная статья про докер: <https://habr.com/ru/post/277699/> (дата обращения: 20.11.2023).

# ПРАКТИЧЕСКАЯ РАБОТА №7

# Задание

Студенту предлагается дополнить задание шестой практики по Java. Нужно создать приложение на микросервисной архитектуре, где один сервис — это приложение шестой практики, а второй сервис — это сервис авторизации.

Сервис авторизации должен быть написан с помощью Spring Security и содержать JWT token. В качестве СУБД во втором сервисе должен быть использован Redis. Должны быть созданы три роли: USER, SELLER, ADMINISTRATOR.

USER не имеет доступ к какому-либо методу, который связан с редактированием или просмотром информации по другому пользователю, не имеет возможности добавлять или удалять товары.

SELLER имеет те же ограничения, что и USER, за исключением, что продавец может добавлять и удалять свои товары.

ADMINISTRATOR не имеет каких-либо ограничений.

В сервис с приложением маркетплейса добавить бизнес-логику, позволяющую пользователь с ролью USER получить роль SELLER.

Практика должна запускаться с помощью docker-compose. Каждый микросервис должен запускаться в отдельном потоке.

# ХОД РАБОТЫ

Для реализации работы аутентификации при помощи Bearer и JWT токенов, для начала необходимо реализовать в проекте сервер, в котором будем хранить данные авторизации пользователей. В нашем случае это будет Redis, изображенные в листинге 6.

Листинг 6 – Код класса-конфигурации RedisConfig

@Configuration  
@EnableRedisRepositories  
public class RedisConfig {  
  
 @Bean  
 public JedisConnectionFactory connectionFactory() {  
 RedisStandaloneConfiguration *configuration* = new RedisStandaloneConfiguration();  
 *configuration*.setHostName("redis"); // Конфигурация для подключения к Redis серверу  
 *configuration*.setPort(6379);  
 return new JedisConnectionFactory(*configuration*); // Создание фабрики подключения Jedis  
 }  
  
 @Bean  
 public RedisTemplate<String, Object> template() {  
 RedisTemplate<String, Object> *template* = new RedisTemplate<>(); // Создание объекта RedisTemplate  
  
 *template*.setConnectionFactory(connectionFactory()); // Установка фабрики подключения  
  
 // Установка сериализаторов для ключей и значений  
 *template*.setKeySerializer(new StringRedisSerializer());  
 *template*.setHashKeySerializer(new StringRedisSerializer());  
 *template*.setHashKeySerializer(new JdkSerializationRedisSerializer());  
 *template*.setValueSerializer(new JdkSerializationRedisSerializer());  
  
 *template*.setEnableTransactionSupport(true); // Включение поддержки транзакций  
  
 *template*.afterPropertiesSet(); // Завершение настройки шаблона Redis  
  
 return *template*; // Возвращаем готовый шаблон Redis  
 }  
}

Таже нам необходимо реализовать конфигурационный файл JwtAuthentificationFilter.class, изображенный в листинге 7.

Листинг 7 – Код класса-конфигурации JwtAuthentificationFilter

@Component  
@Slf4j  
public class JwtAuthenticationFilter extends OncePerRequestFilter {  
  
 @Autowired  
 private JwtService jwtService;  
 @Autowired  
 private UserDetailsService userDetailsService;  
  
 @Override  
 protected void doFilterInternal(HttpServletRequest *request*,  
 HttpServletResponse *response*,  
 FilterChain *filterChain*) throws ServletException, IOException {  
 final String *authHeader* = *request*.getHeader("Authorization"); // Считываем хэдер авторизации  
 final String *jwt*;  
 final String *userEmail*;  
  
 if (StringUtils.*isEmpty*(*authHeader*) || !StringUtils.*startsWith*(*authHeader*, "Bearer ")) {  
 *filterChain*.doFilter(*request*, *response*); // Если хэдер отсутствует или не начинается с "Bearer", пропускаем запрос дальше  
 return;  
 }  
  
 *jwt* = *authHeader*.substring(7); // Извлекаем JWT токен, убирая "Bearer "  
 *userEmail* = jwtService.extractUserName(*jwt*); // Извлекаем email пользователя из токена  
  
 *log*.debug(*userEmail*); // Выводим email в логи для отладки  
  
 if (StringUtils.*isNotEmpty*(*userEmail*) && SecurityContextHolder.*getContext*().getAuthentication() == null) {  
 UserDetails *userDetails* = userDetailsService.loadUserByUsername(*userEmail*); // Загружаем UserDetails по email  
  
 if (jwtService.isTokenValid(*jwt*, *userDetails*)) { // Проверяем валидность токена  
 SecurityContext *context* = SecurityContextHolder.*createEmptyContext*();  
 UsernamePasswordAuthenticationToken *authenticationToken* = new UsernamePasswordAuthenticationToken(  
 *userDetails*, null, *userDetails*.getAuthorities()  
 );  
 *authenticationToken*.setDetails(new WebAuthenticationDetailsSource().buildDetails(*request*));  
 *context*.setAuthentication(*authenticationToken*);  
 SecurityContextHolder.*setContext*(*context*); // Устанавливаем аутентификацию в SecurityContextHolder  
 }  
 }  
  
 *filterChain*.doFilter(*request*, *response*); // Передаём запрос дальше в цепочку фильтров  
 }  
}

Также необходимо создать некоторые dao-классы. Файловая система необходимых dao-классов в проекте изображена на рисунке 30.



Рисунок 31 – Файловая система dao-классов

Не забываем по саму модель для нашего Redis сервера для аутентификации, изображенную в листинге 8.

Листинг 7 – Код класса-модели RedisEntity

@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@RedisHash("Entity")  
@Builder  
public class RedisEntity implements Serializable {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 private int id;  
 private String userEmail;  
 private String token;  
}

И наконец нам необходимы следующие классы-сервисы для реализации всех необходимых методов, изображенные в файловой системе на рисунке 32.

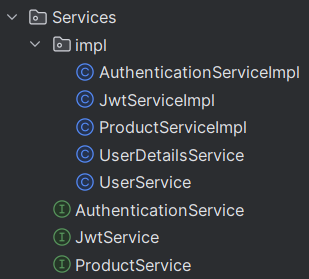


Рисунок 32 – Файловая система классов-сервисов

Чтобы создать все необходимые Docker-контейнеры, нам будут необходимы следующие Dockerfile и docker-compose.yaml, изображенные на рисунках 33 и 34 соответственно.

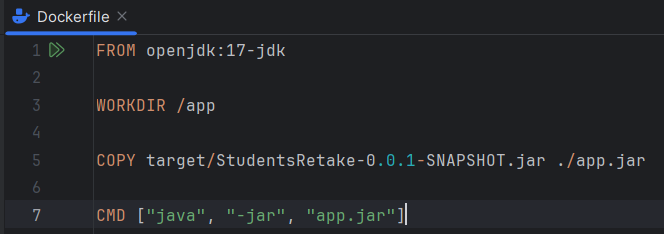


Рисунок 33 – Содержимое Dockerfile

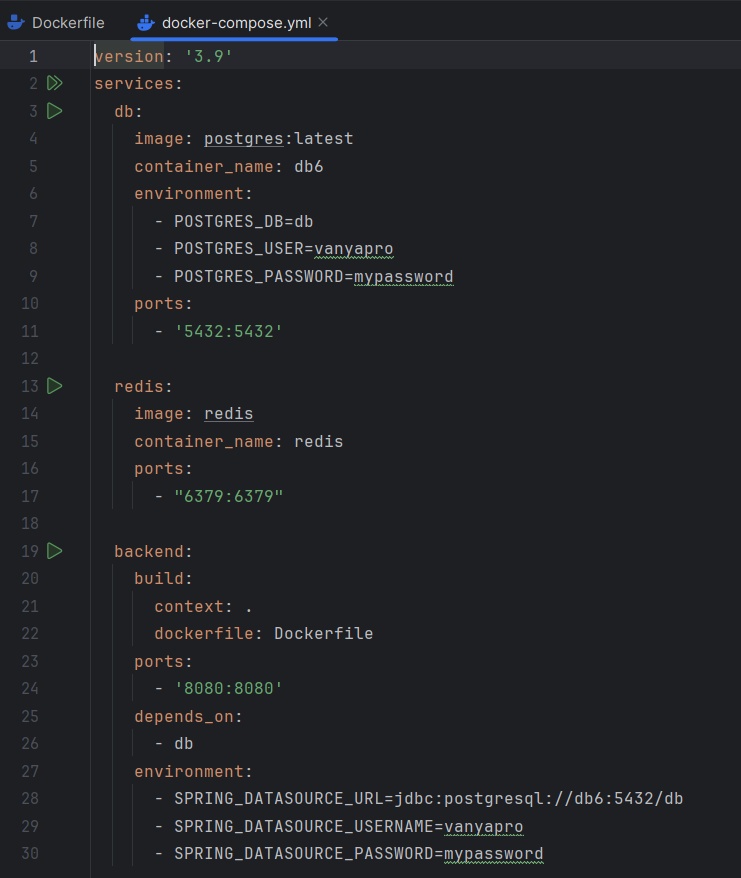


Рисунок 34 – Содержимое docker-compose.yaml

После выполнения команд “mvn clean install” для компилирования нашего проекта и “docker-compose up” для создания контейнеров, приложение будет развернуто в Docker, рисунок 35.



Рисунок 35 – Созданные контейнеры в Docker

# ТЕСТИРОВАНИЕ

Через Postman зарегистрируем пользователя-продавца с соответствующей ролью и получим его токен, как изображено на рисунке 36.

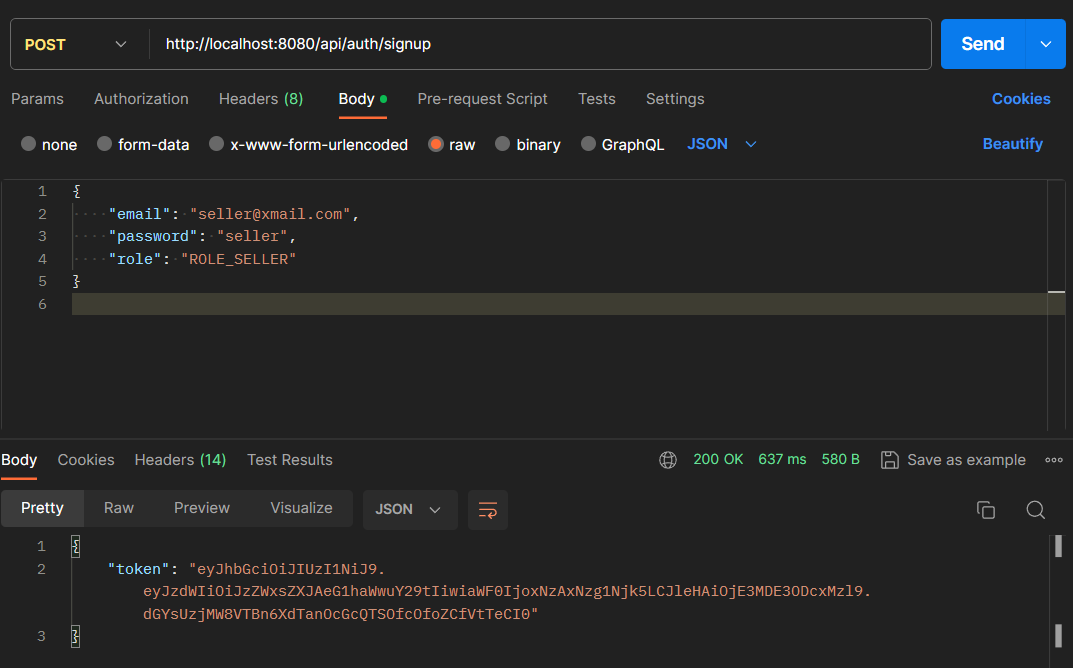


Рисунок 36 – Регистрация продавца

Теперь то же самое с пользователем-администратором, как изображено на рисунке 37.

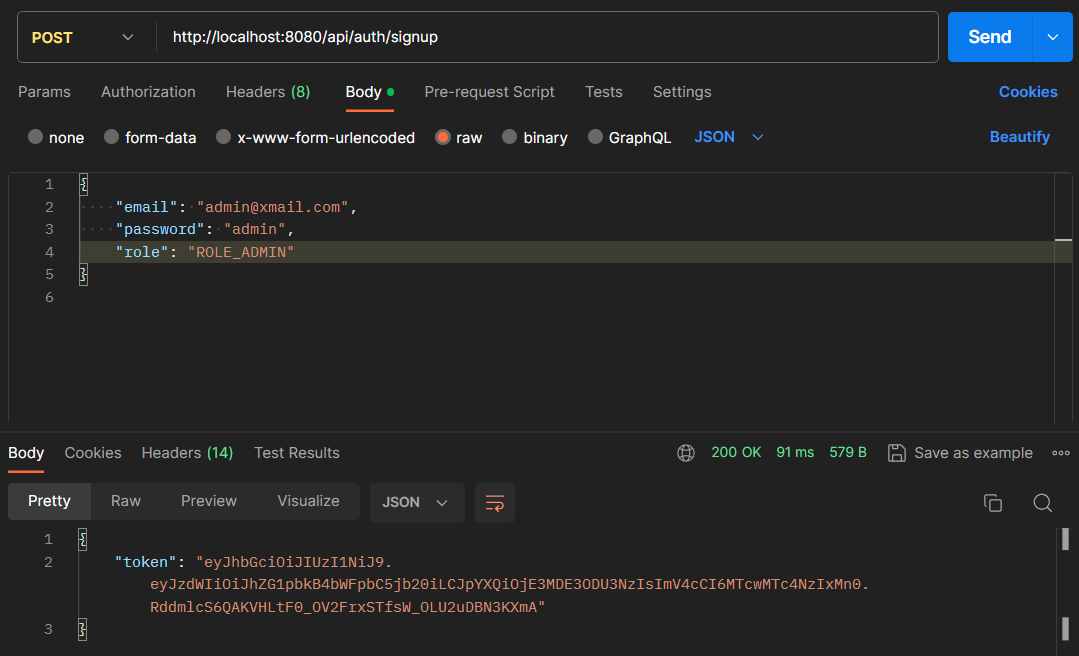


Рисунок 37 – Регистрация администратора

Выбрав Bearer Token и вставив в соответствующее поле токен администратора, мы можем посмотреть информацию о любом пользователе как на рисунке 38.

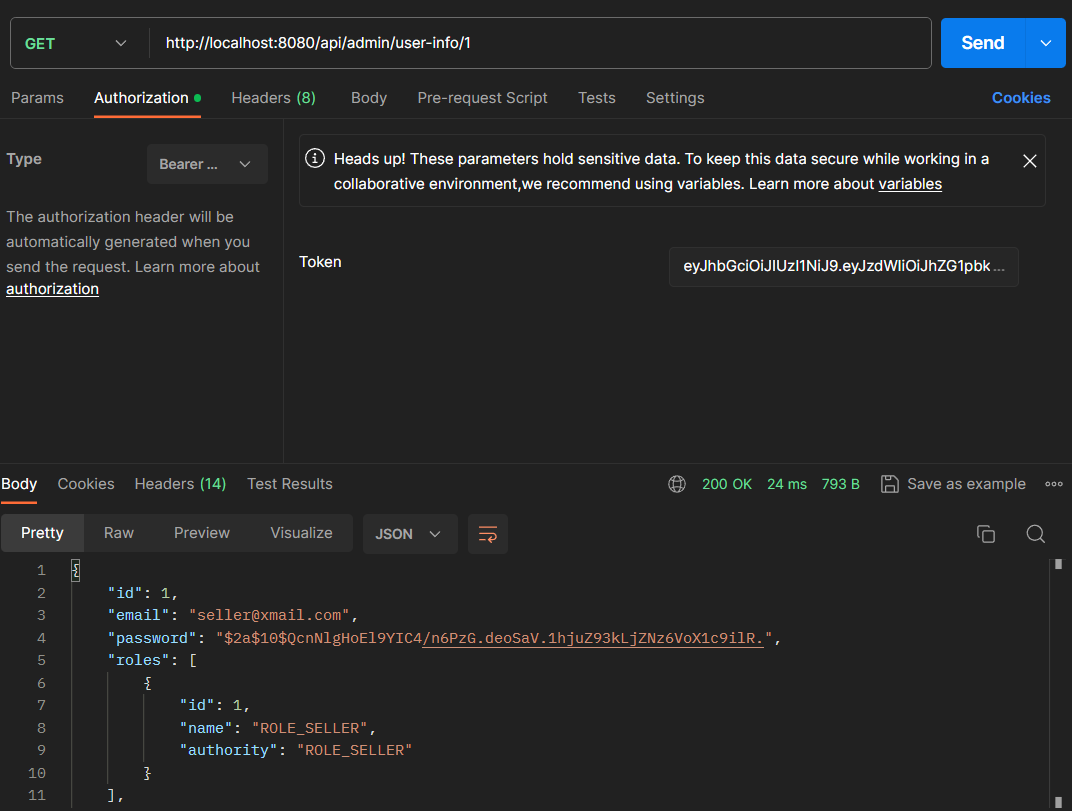


Рисунок 38 – Просмотр информации о пользователе с токеном админа

Чтобы добавить продуктов, залогинимся в качестве продавца, как на рисунке 39, и скопируем токен.

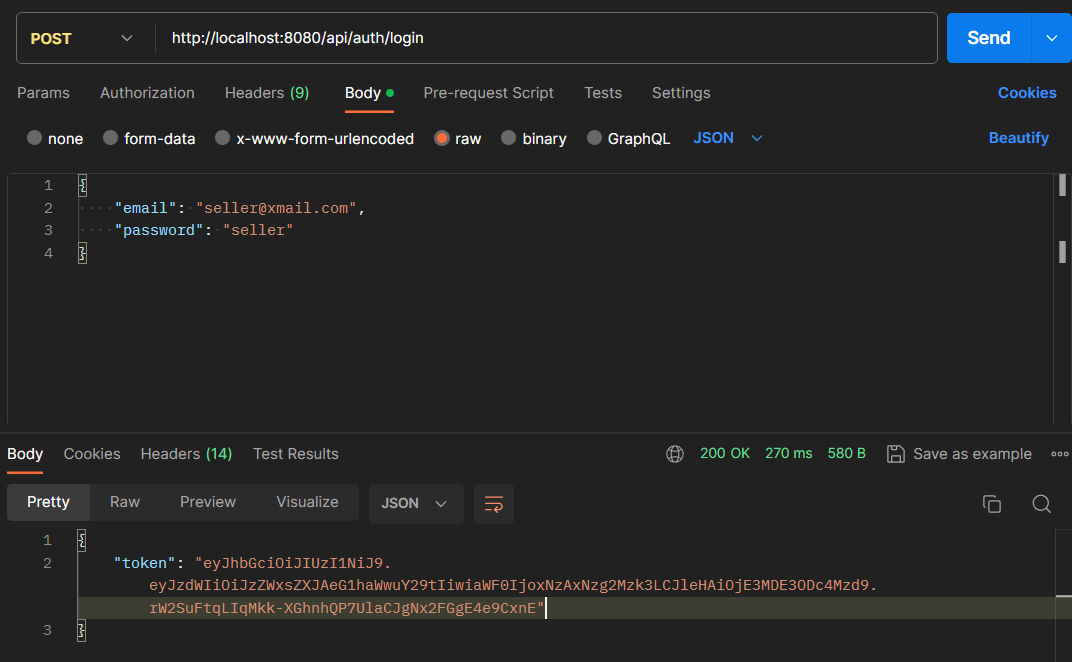


Рисунок 39 – Авторизация в качестве продавца

Предварительно не забыв вставить токен продавца, попробуем добавить продуктов, пример добавления изображен на рисунке 40.

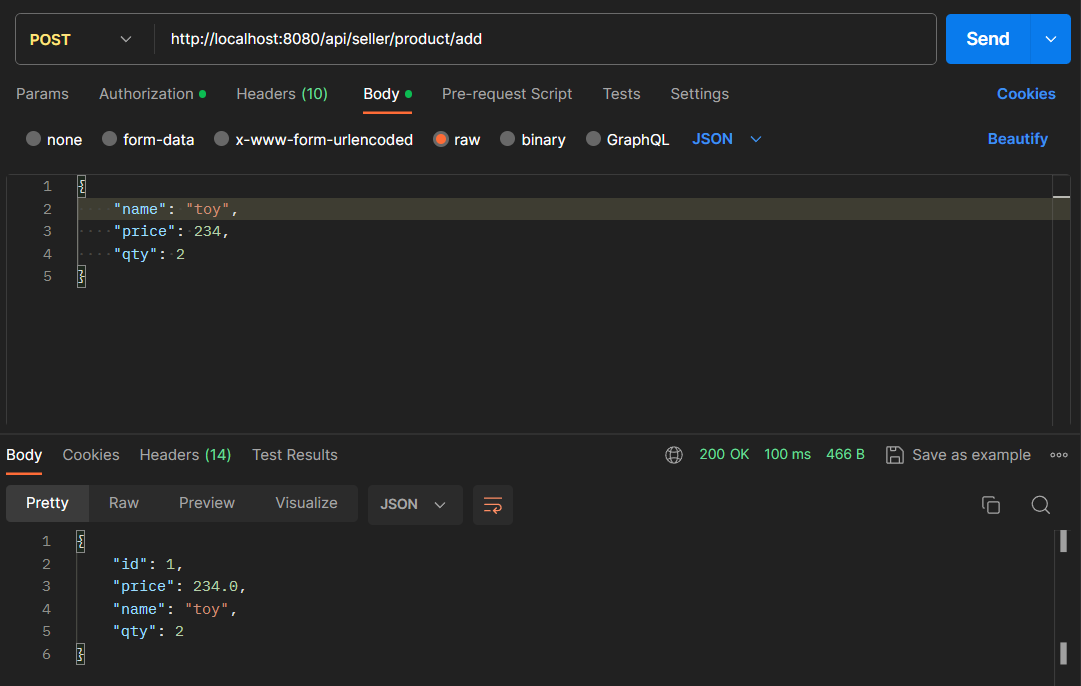


Рисунок 40 – Добавление продукта за продавца

Опять же, вставив токен, можем посмотреть продукты, которые мы добавили в систему, рисунок 41.

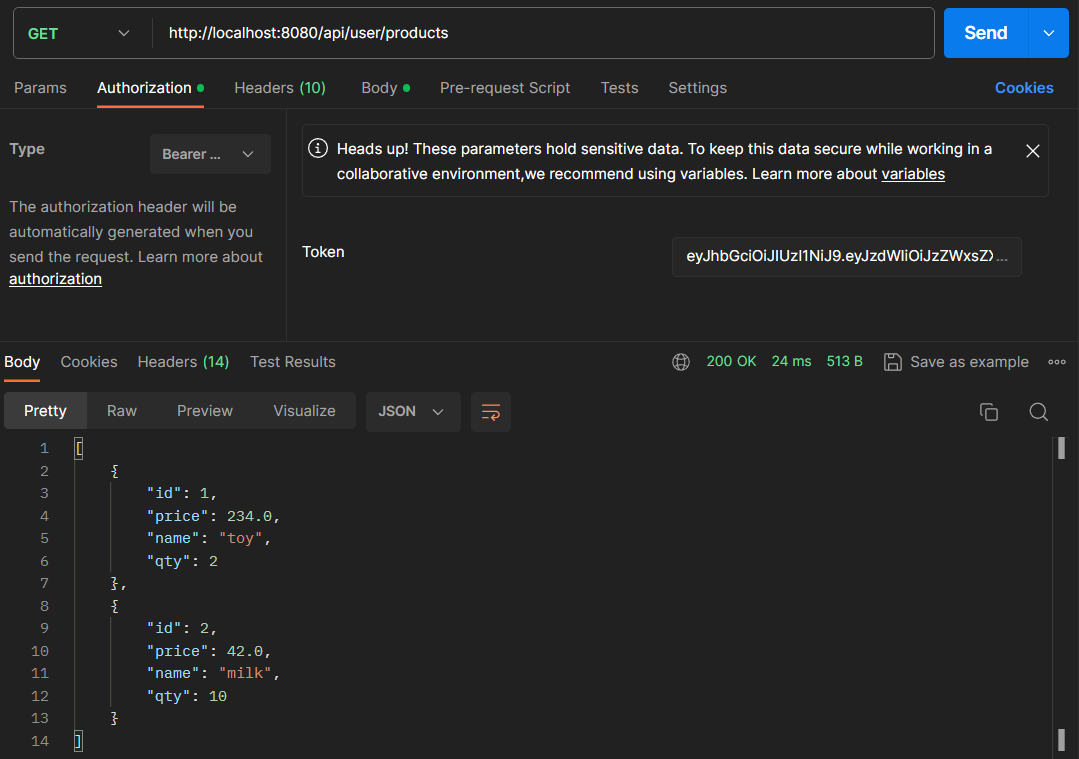


Рисунок 41 – Список всех продуктов

Вставив токен, попробуем удалить продукт с id равным 1, рисунок 42.

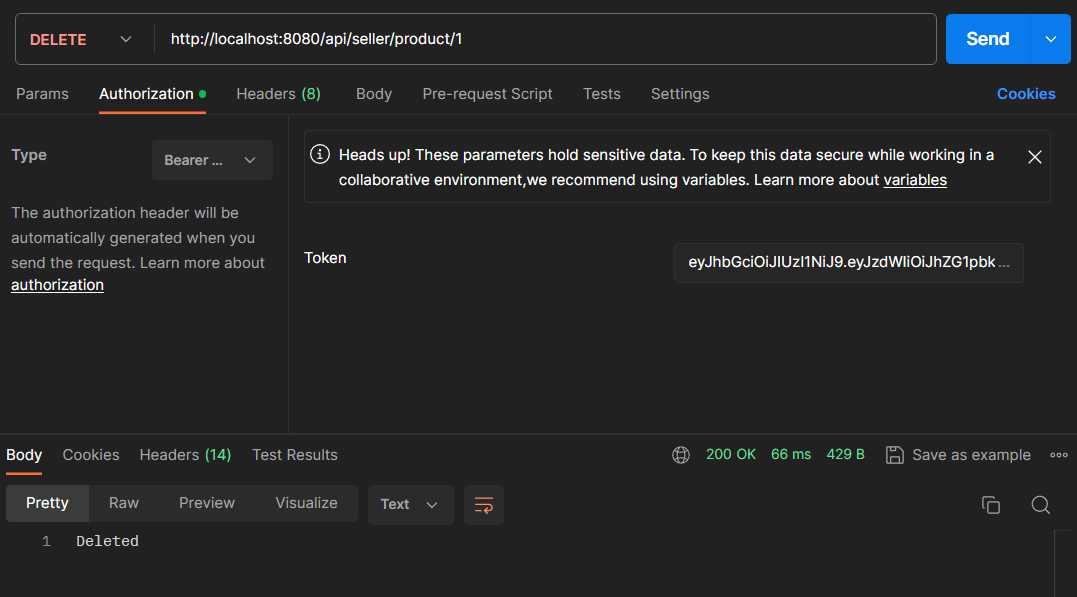


Рисунок 42 – Удаление продукта

Посмотрим все продукты еще раз, проверив результат удаления, как изображено на рисунке 43.

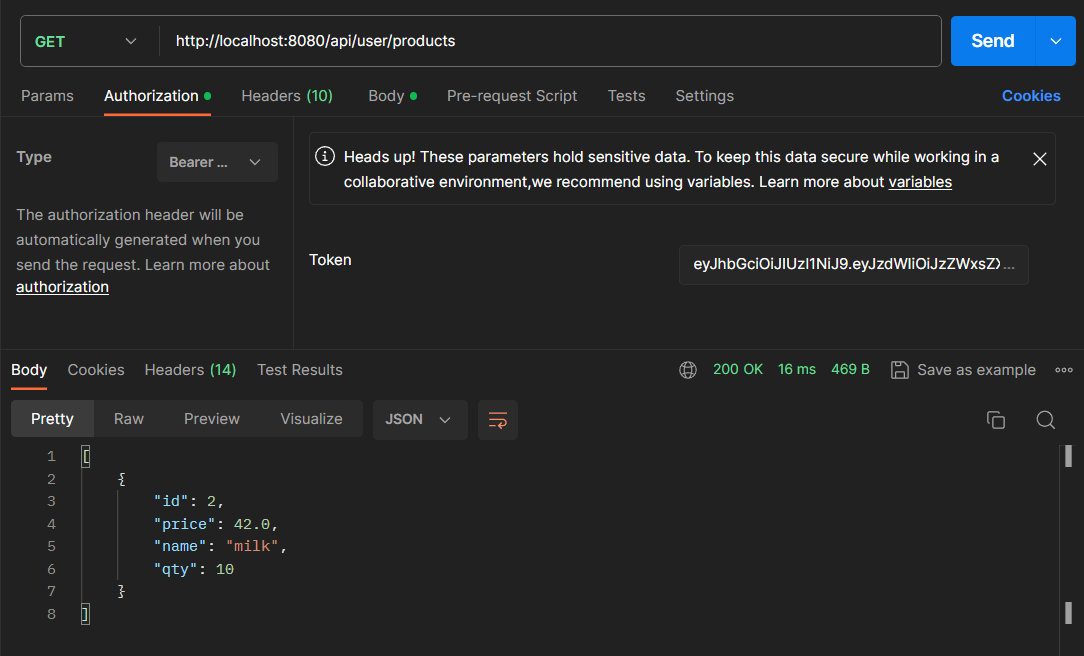


Рисунок 43 – Просмотр всех продуктов

Продукт действительно удалился, в результате чего мы убедились в работе всех функций приложения.

# Вывод

Убедившись в работе всех методов, нам удалось реализовать работу JWT токенов в нашем Java Spring веб-приложении. Была протестирована REST-составляющая приложения. А главное, что все работает через docker-контейнеры и успешно общается между собой реализуя аутентификацию пользователей. Репозиторий доступен по ссылке: <https://github.com/BadHard101/JWT_Token_Auth_via_Redis>

# ОТВЕТЫ НА ВОПРОСЫ

* 1. Spring Security.

Spring Security - это мощный и гибкий фреймворк для обеспечения безопасности в приложениях на основе Spring. Он предоставляет аутентификацию, авторизацию, защиту от атак и другие функции безопасности.

2. Что такое JWT Token и как он работает.

JWT (JSON Web Token) - это открытый стандарт (RFC 7519) для передачи утверждений между сторонами в формате JSON. Он состоит из заголовка, полезной нагрузки и подписи. JWT используется для передачи информации между клиентом и сервером, и его подпись обеспечивает проверку подлинности данных.

3. Микросервисная архитектура. Плюсы и минусы перед монолитной системой.

Плюсы: Легкость масштабируемости, независимость сервисов, легкость развертывания и обновлений.

Минусы: Сложность управления распределенным состоянием, необходимость обеспечения согласованности данных, дополнительная сложность в обеспечении безопасности.

4. Варианты масштабируемости.

Вертикальная масштабируемость: Увеличение мощности сервера путем добавления ресурсов (CPU, RAM).

Горизонтальная масштабируемость: Увеличение мощности системы путем добавления дополнительных серверов.

5. Что такое вертикальная масштабируемость.

Увеличение производительности системы путем добавления ресурсов к существующему серверу.

6. Что такое горизонтальная масштабируемость.

Увеличение производительности системы путем добавления новых серверов, обычно в сети.

7. Объяснить принцип сбора микросервисной архитектуры в Docker.

Docker упрощает упаковку, доставку и запуск приложений в контейнерах. Каждый микросервис может быть упакован в свой собственный контейнер, включая все зависимости, что обеспечивает надежную и переносимую среду выполнения.

8. NoSQL DB. Почему в сервисе авторизации удобнее использовать Redis.

Redis хорошо подходит для кеширования и быстрого доступа к данным, что важно для сервиса авторизации. Он обеспечивает быстрый доступ к данным по ключу, что улучшает производительность системы.

9. Многопотчность — что это такое. Как оно работает в Java.

Многопоточность - это выполнение нескольких потоков кода параллельно. В Java это реализуется через классы Thread и Runnable. Многопоточные приложения могут улучшить производительность, но требуют особой осторожности из-за потенциальных проблем с синхронизацией данных.

10. Класс Thread, расскажите про его методы.

Thread - это основной класс для работы с многопоточностью в Java.

Некоторые методы:

start(): Запускает новый поток.

run(): Определяет код, который будет выполняться в потоке.

sleep(long millis): Останавливает выполнение потока на указанное количество миллисекунд.

join(): Ожидает завершения выполнения потока.

interrupt(): Прерывает выполнение потока.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Сервер при помощи OAuth 2.0 JWT: <https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html> (дата обращения: 25.11.2023). – Текст: электронный.

2. Статья на Хабр / EST API с использованием Spring Security и JWT: <https://habr.com/ru/articles/545610/> (дата обращения: 25.11.2023). – Текст: электронный.

3. Документация Redis: <https://redis.io/docs/> (дата обращения: 25.11.2023). – Текст: электронный.

4. Документация Java: <https://docs.oracle.com/en/java/> (дата обращения: 25.11.2023). – Текст: электронный.

5. Документация Spring: <https://spring.io/quickstart> (дата обращения: 25.11.2023).

6. Методические указания по выполнению практической работы: <https://online-edu.mirea.ru/mod/resource/view.php?id=517436> (дата обращения: 25.11.2023).

7. Официальная документация докера: <https://docs.docker.com/> (дата обращения: 25.11.2023).

8. Более сложная и подробная статья про докер: <https://habr.com/ru/post/277699/> (дата обращения: 25.11.2023).