

МЕТОДИЧЕСКОЕ ПОСОБИЕ ПО РАЗРАБОТКЕ КЛИЕНТ-СЕРВЕРНОГО ПРИЛОЖЕНИЯ С ИСПОЛЬЗОВАНИЕМ SPRING BOOT, REACT, POSTGRESQL, REDIS, KAFKA, DOCKER

СОДЕРЖАНИЕ:

1. Введение и описание проекта
2. Технологический стек и назначение компонентов
3. Архитектура приложения
4. Установка необходимого программного обеспечения
5. Создание проекта с нуля
6. Структура проекта
7. Настройка Backend (Spring Boot)
8. Настройка Frontend (React)
9. Настройка Docker и Docker Compose
10. Объяснение Redis (кэширование)
11. Объяснение Kafka (асинхронная обработка событий)
12. Структура базы данных postgresql
13. Основные компоненты кода с примерами
14. Запуск проекта
15. Тестирование приложения
16. Заключение

1. ВВЕДЕНИЕ И ОПИСАНИЕ ПРОЕКТА

Данное приложение представляет собой полнофункциональную платформу для заказа услуг на дом. Пользователи могут регистрироваться как клиенты или исполнители, просматривать доступные услуги, создавать заказы, отслеживать их статус и получать уведомления в реальном времени.

Основные возможности:

- Регистрация и аутентификация пользователей (JWT)
- Управление услугами и категориями
- Создание и управление заказами
- Система ролей (Клиент, Исполнитель, Администратор)
- Кэширование данных через Redis
- Асинхронная обработка событий через Kafka
- Уведомления в реальном времени через WebSocket
- Административная панель

2. ТЕХНОЛОГИЧЕСКИЙ СТЕК И НАЗНАЧЕНИЕ КОМПОНЕНТОВ

BACKEND (Серверная часть):

- Spring Boot 3.2.0 - основной фреймворк для создания REST API
- Spring Security - безопасность и аутентификация пользователей
- JWT (JSON Web Tokens) - токены для безопасной аутентификации
- Spring Data JPA - работа с базой данных через ORM
- PostgreSQL - реляционная база данных для хранения данных
- Redis - кэширование данных для повышения производительности
- Apache Kafka - обработка событий и асинхронная коммуникация
- WebSocket - уведомления в реальном времени
- Swagger/OpenAPI - автоматическая документация API

FRONTEND (Клиентская часть):

- React 18 - библиотека для создания пользовательского интерфейса
- TypeScript - типизированный JavaScript для надежности кода
- Vite - быстрый сборщик и dev-сервер
- React Router - маршрутизация между страницами
- TanStack Query - управление состоянием сервера и кэширование запросов
- Zustand - управление глобальным состоянием приложения
- Tailwind CSS - утилитарный CSS фреймворк для стилизации
- Axios - HTTP клиент для запросов к API

ИНФРАСТРУКТУРА:

- Docker - контейнеризация приложений
- Docker Compose - оркестрация множественных контейнеров
- Nginx - веб-сервер и reverse proxy для frontend

ЗАЧЕМ НУЖЕН DOCKER?

Docker позволяет упаковать приложение и все его зависимости в контейнер, который может работать на любой системе. Это обеспечивает:

- Единообразие окружения (одинаковая работа на всех машинах)
- Простоту развертывания (один раз настроили - работает везде)
- Изоляцию (каждое приложение в своем контейнере)
- Масштабируемость (легко запустить несколько экземпляров)

ЗАЧЕМ НУЖЕН REDIS?

Redis - это in-memory хранилище данных (хранит данные в оперативной памяти).

Используется для:

- Кэширования часто запрашиваемых данных (категории, услуги)
- Снижения нагрузки на базу данных
- Ускорения работы приложения

- Хранения сессий пользователей

Пример: Вместо того чтобы каждый раз обращаться к PostgreSQL за списком категорий, мы сохраняем их в Redis. При следующем запросе данные берутся из Redis (быстрее в 100 раз).

ЗАЧЕМ НУЖЕН KAFKA?

Apache Kafka - это распределенная платформа для обработки потоков событий.

Используется для:

- Асинхронной обработки событий (создание заказа, изменение статуса)
- Разделения ответственности между компонентами системы
- Обеспечения надежности (события не теряются)
- Масштабируемости (можно обрабатывать миллионы событий)

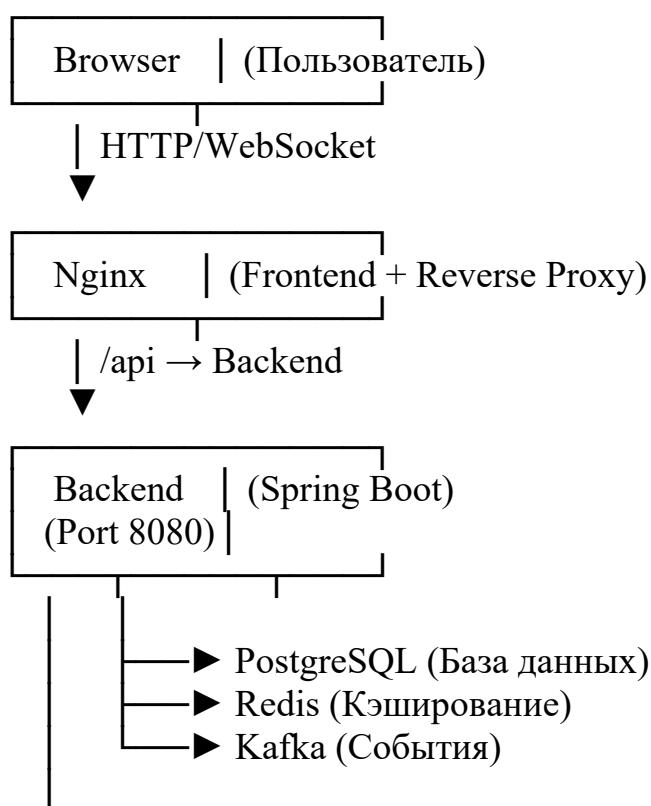
Пример: Когда клиент создает заказ, это событие отправляется в Kafka.

Различные сервисы могут подписаться на это событие и выполнить свои действия:

- Сервис уведомлений отправляет email
- Сервис аналитики обновляет статистику
- Сервис биллинга обрабатывает платеж

3. АРХИТЕКТУРА ПРИЛОЖЕНИЯ

Структура взаимодействия компонентов:





4. УСТАНОВКА НЕОБХОДИМОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Для работы с проектом необходимо установить:

1. Java Development Kit (JDK) 17 или выше

Скачать: <https://adoptium.net/>

Проверка: `java -version`

2. Maven 3.9+ (или использовать встроенный mvnw)

Скачать: <https://maven.apache.org/download.cgi>

Проверка: `mvn -version`

3. Node.js 18+ и npm

Скачать: <https://nodejs.org/>

Проверка: `node -v` и `npm -v`

4. Docker Desktop

Скачать: <https://www.docker.com/products/docker-desktop>

Проверка: `docker --version` и `docker-compose --version`

5. Git (опционально, для версионирования)

Скачать: <https://git-scm.com/>

6. IDE (рекомендуется):

- IntelliJ IDEA (для Backend)

- Visual Studio Code (для Frontend)

5. СОЗДАНИЕ ПРОЕКТА С НУЛЯ

ШАГ 1: СОЗДАНИЕ СТРУКТУРЫ ПРОЕКТА

Создайте корневую папку проекта:

`mkdir Проект_СТСР`

`cd Проект_СТСР`

Создайте структуру папок:

`mkdir backend`

`mkdir frontend`

`mkdir scripts`

ШАГ 2: СОЗДАНИЕ BACKEND ПРОЕКТА

Вариант А: Через Spring Initializr (<https://start.spring.io/>)

- Выберите Maven Project
- Java 17
- Spring Boot 3.2.0
- Добавьте зависимости:
 - * Spring Web
 - * Spring Data JPA
 - * Spring Security
 - * PostgreSQL Driver
 - * Spring Data Redis
 - * Spring for Apache Kafka
 - * Spring WebSocket
 - * Spring Boot Actuator
 - * Lombok
 - * Validation

Вариант В: Вручную

Создайте файл backend/pom.xml (см. раздел 7)

ШАГ 3: СОЗДАНИЕ FRONTEND ПРОЕКТА

Перейдите в папку frontend:

```
cd frontend
```

Инициализируйте React проект с Vite:

```
npm create vite@latest . -- --template react-ts
```

Установите зависимости:

```
npm install
```

Дополнительные зависимости:

```
npm install react-router-dom axios @tanstack/react-query zustand
```

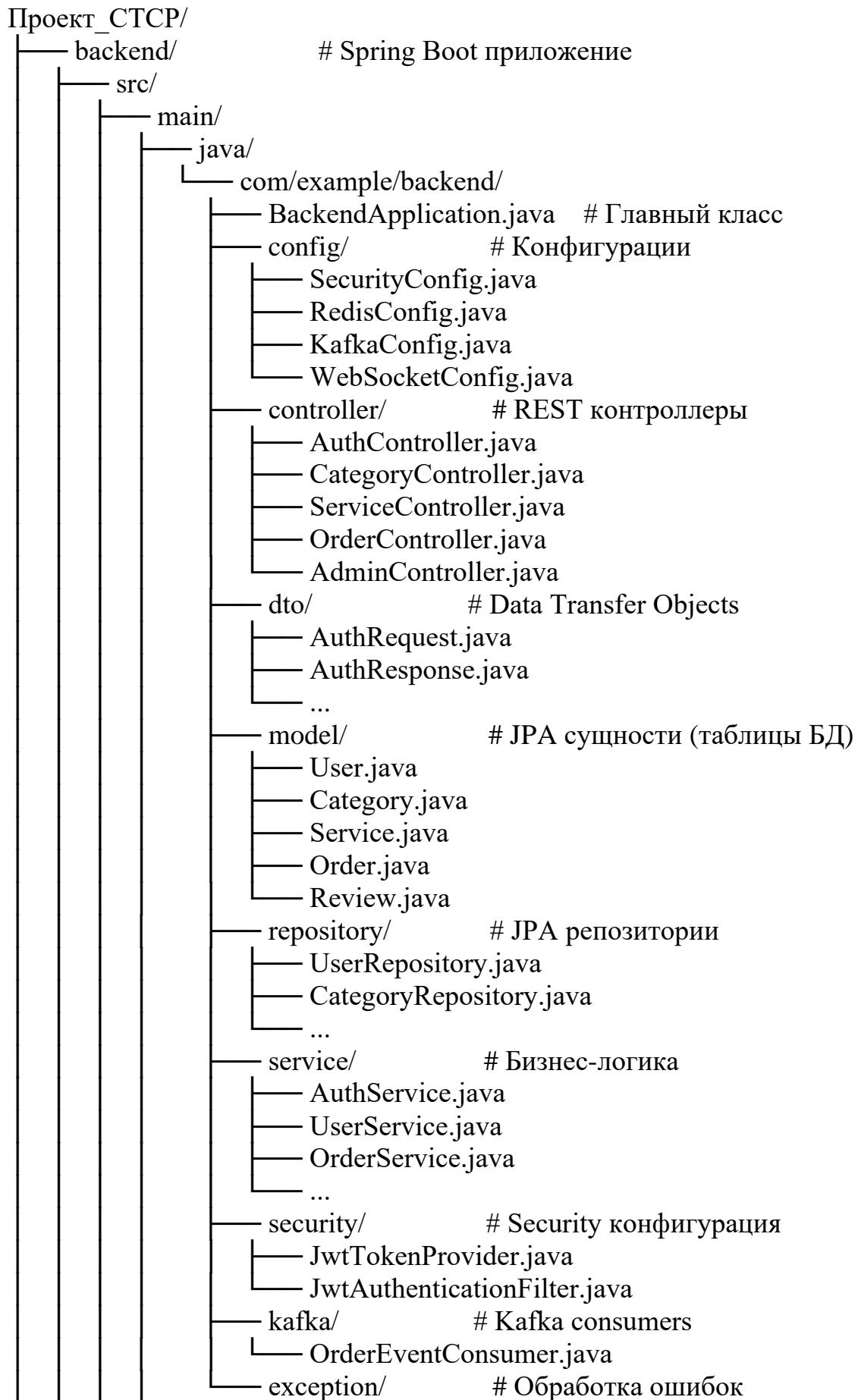
```
npm install react-hook-form zod @hookform/resolvers
```

```
npm install sockjs-client @stomp/stompjs react-hot-toast
```

```
npm install lucide-react clsx tailwind-merge
```

```
npm install -D tailwindcss postcss autoprefixer
```

6. СТРУКТУРА ПРОЕКТА



```

    └── GlobalExceptionHandler.java
    └── resources/
        └── application.properties      # Конфигурация
    └── test/
    └── Dockerfile
    └── pom.xml

    └── frontend                  # React приложение
        └── src/
            └── api/                # API клиенты
                ├── client.ts       # Axios конфигурация
                └── services.ts     # API методы
            └── components/        # React компоненты
                ├── Layout.tsx
                └── ProtectedRoute.tsx
            └── pages/              # Страницы
                ├── HomePage.tsx
                ├── LoginPage.tsx
                ├── ServicesPage.tsx
                ├── OrdersPage.tsx
                ├── AdminPage.tsx
                └── ProviderPage.tsx
            └── store/              # Zustand stores
                └── authStore.ts
            └── App.tsx             # Главный компонент
            └── main.tsx            # Точка входа
        └── Dockerfile
        └── nginx.conf
        └── package.json
        └── vite.config.ts

    └── docker-compose.yml      # Docker Compose конфигурация
    └── README.md

```

7. НАСТРОЙКА BACKEND (SPRING BOOT)

7.1. ФАЙЛ pom.xml (Maven зависимости)

Создайте файл backend/pom.xml:

```

<?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>3.2.0</version>
<relativePath/>
</parent>
<groupId>com.example</groupId>
<artifactId>backend</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>backend</name>
<properties>
    <java.version>17</java.version>
    <lombok.version>1.18.30</lombok.version>
</properties>
<dependencies>
    <!-- Spring Boot Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Boot Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <!-- Spring Boot Security -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>

    <!-- Spring Boot Validation -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>

    <!-- Spring Boot Redis -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>

    <!-- Spring Kafka -->
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>

    <!-- JWT -->
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-api</artifactId>
        <version>0.12.3</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-impl</artifactId>
        <version>0.12.3</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
```

```

<artifactId>jjwt-jackson</artifactId>
<version>0.12.3</version>
<scope>runtime</scope>
</dependency>

<!-- PostgreSQL -->
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>

<!-- Lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>

<!-- Jackson for Redis -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>

<!-- Spring Boot WebSocket -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>

<!-- Spring Boot Actuator -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<!-- Swagger/OpenAPI -->
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.3.0</version>
</dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
            <configuration>
                <source>17</source>
                <target>17</target>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <version>${lombok.version}</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
        <plugin>

```

```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<configuration>
    <excludes>
        <exclude>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
        </exclude>
    </excludes>

<mainClass>com.example.backend.BackendApplication</mainClass>
    </configuration>
</plugin>
</plugins>
</build>
</project>

```

ОБЪЯСНЕНИЕ ЗАВИСИМОСТЕЙ:

- spring-boot-starter-web: создание REST API
- spring-boot-starter-data-jpa: работа с базой данных
- spring-boot-starter-security: безопасность
- spring-boot-starter-data-redis: кэширование
- spring-kafka: обработка событий
- jjwt: JWT токены для аутентификации
- postgresql: драйвер для PostgreSQL
- lombok: автоматическая генерация геттеров/сеттеров
- springdoc-openapi: документация API

7.2. ГЛАВНЫЙ КЛАСС ПРИЛОЖЕНИЯ

Создайте **backend/src/main/java/com/example/backend/BackendApplication.java:** файл

```

package com.example.backend;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class BackendApplication {
    public static void main(String[] args) {
        SpringApplication.run(BackendApplication.class, args);
    }
}

```

ОБЪЯСНЕНИЕ:

`@SpringBootApplication` - это составная аннотация, которая включает:

- `@Configuration` - класс является конфигурацией Spring
- `@EnableAutoConfiguration` - автоматическая настройка Spring Boot
- `@ComponentScan` - сканирование компонентов в пакете

7.3. КОНФИГУРАЦИЯ (application.properties)

Создайте файл backend/src/main/resources/application.properties:

```
# Название приложения
spring.application.name=home-services-backend

# Порт сервера и контекстный путь
server.port=8080
server.servlet.context-path=/api

# Настройки базы данных PostgreSQL
spring.datasource.url=jdbc:postgresql://localhost:5432/home_services_db
spring.datasource.username=postgres
spring.datasource.password=postgrespassword
spring.datasource.driver-class-name=org.postgresql.Driver

# Настройки JPA/Hibernate
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=false
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

# Настройки Redis
spring.data.redis.host=localhost
spring.data.redis.port=6379
spring.cache.type=redis
spring.cache.redis.time-to-live=3600000

# Настройки Kafka
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.producer.key-
serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-
serializer=org.springframework.kafka.support.serializer.JsonSerializer
spring.kafka.consumer.group-id=home-services-group
spring.kafka.consumer.key-
deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-
deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.auto-offset-reset=earliest
spring.kafka.consumer.properties.spring.json.trusted.packages=*

# JWT настройки
jwt.secret=mySecretKey123456789012345678901234567890123456789012345
67890
jwt.expiration=86400000

# Actuator (мониторинг)
management.endpoints.web.exposure.include=health,info
management.endpoints.web.base-path=/actuator

# Swagger документация
springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/swagger-ui.html

# CORS (разрешенные источники)
app.cors.allowed-origins=http://localhost:3000,http://localhost:5173
```

ОБЪЯСНЕНИЕ КЛЮЧЕВЫХ ПАРАМЕТРОВ:

- server.servlet.context-path=/api: все API endpoints будут иметь префикс /api
- spring.jpa.hibernate.ddl-auto=update: автоматическое создание/обновление таблиц

- spring.cache.type=redis: использование Redis для кэширования
- spring.kafka.bootstrap-servers: адрес Kafka сервера
- jwt.secret: секретный ключ для подписи JWT токенов
- jwt.expiration: время жизни токена (24 часа)

7.4. СОЗДАНИЕ МОДЕЛИ (ENTITY) - ПРИМЕР User

Создайте файл backend/src/main/java/com/example/backend/model/User.java:

```
package com.example.backend.model;

import jakarta.persistence.*;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.time.LocalDateTime;
import java.util.HashSet;
import java.util.Set;

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

    @NotBlank
    @Column(unique = true, nullable = false)
    private String username;

    @NotBlank
    @Email
    @Column(unique = true, nullable = false)
    private String email;

    @NotBlank
    @Column(nullable = false)
    private String password;

    @NotBlank
    private String firstName;

    @NotBlank
    private String lastName;

    private String phone;
    private String address;

    @Enumerated(EnumType.STRING)
    private Role role = Role.CUSTOMER;

    @Column(nullable = false)
    private LocalDateTime createdAt = LocalDateTime.now();
```

```

    @Column(nullable = false)
    private Boolean active = true;

    @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
    private Set<Order> orders = new HashSet<>();

    public enum Role {
        CUSTOMER, PROVIDER, ADMIN
    }
}

```

ОБЪЯСНЕНИЕ АННОТАЦИЙ:

- `@Entity`: класс является JPA сущностью (таблицей в БД)
- `@Table(name = "users")`: имя таблицы в БД
- `@Id`: первичный ключ
- `@GeneratedValue`: автоматическая генерация ID
- `@Column`: настройки колонки (unique, nullable)
- `@NotBlank`: валидация (поле не может быть пустым)
- `@Email`: валидация email
- `@Enumerated`: сохранение enum как строки
- `@OneToMany`: связь один-ко-многим (один пользователь - много заказов)
- `@Data`: Lombok генерирует геттеры, сеттеры, `toString`, `equals`, `hashCode`
- `@NoArgsConstructor`: конструктор без параметров
- `@AllArgsConstructor`: конструктор со всеми параметрами

7.5. СОЗДАНИЕ РЕПОЗИТОРИЯ

Создайте файл
`backend/src/main/java/com/example/backend/repository/UserRepository.java`:

```

package com.example.backend.repository;

import com.example.backend.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByUsername(String username);
    Optional<User> findByEmail(String email);
    boolean existsByUsername(String username);
    boolean existsByEmail(String email);
}

```

ОБЪЯСНЕНИЕ:

- `JpaRepository<User, Long>`: базовый интерфейс Spring Data JPA
 - * `User` - тип сущности
 - * `Long` - тип первичного ключа

- Методы findByUsername, findByEmail: Spring Data JPA автоматически создает SQL запросы на основе имени метода
- Optional: обертка для безопасной работы с null

7.6. СОЗДАНИЕ СЕРВИСА

Создайте файл
backend/src/main/java/com/example/backend/service/UserService.java:

```
package com.example.backend.service;

import com.example.backend.model.User;
import com.example.backend.repository.UserRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@RequiredArgsConstructor
public class UserService {
    private final UserRepository userRepository;

    @Cacheable(value = "users")
    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    @Cacheable(value = "users", key = "#id")
    public User getUserById(Long id) {
        return userRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("User not found"));
    }

    @CacheEvict(value = "users", allEntries = true)
    public User createUser(User user) {
        return userRepository.save(user);
    }
}
```

ОБЪЯСНЕНИЕ:

- `@Service`: класс является сервисом (бизнес-логика)
- `@RequiredArgsConstructor`: Lombok создает конструктор для final полей
- `@Cacheable`: результат метода кэшируется в Redis
- `@CacheEvict`: очистка кэша при изменении данных
- `userRepository.save()`: сохранение в БД (INSERT или UPDATE)

7.7. СОЗДАНИЕ КОНТРОЛЛЕРА

Создайте файл
backend/src/main/java/com/example/backend/controller/AuthController.java:

```
package com.example.backend.controller;

import com.example.backend.dto.AuthRequest;
import com.example.backend.dto.AuthResponse;
import com.example.backend.dto.RegisterRequest;
import com.example.backend.service.AuthService;
import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/auth")
@RequiredArgsConstructor
public class AuthController {
    private final AuthService authService;

    @PostMapping("/register")
    public ResponseEntity<AuthResponse> register(
        @Valid @RequestBody RegisterRequest request) {
        return ResponseEntity.ok(authService.register(request));
    }

    @PostMapping("/login")
    public ResponseEntity<AuthResponse> login(
        @Valid @RequestBody AuthRequest request) {
        return ResponseEntity.ok(authService.login(request));
    }
}
```

ОБЪЯСНЕНИЕ:

- `@RestController`: класс обрабатывает HTTP запросы, возвращает JSON
 - `@RequestMapping("/auth")`: базовый путь для всех методов
 - `@PostMapping`: обработка POST запросов
 - `@RequestBody`: данные из тела запроса преобразуются в объект
 - `@Valid`: валидация данных из запроса
 - `ResponseEntity`: обертка для HTTP ответа (статус код + тело)

7.8. СОЗДАНИЕ DTO (Data Transfer Object)

Создайте файл
backend/src/main/java/com/example/backend/dto/AuthRequest.java:

```
package com.example.backend.dto;

import jakarta.validation.constraints.NotBlank;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class AuthRequest {
    @NotBlank(message = "Username is required")
    private String username;
```

```
    @NotBlank(message = "Password is required")
    private String password;
}
```

ОБЪЯСНЕНИЕ:

DTO используется для передачи данных между клиентом и сервером.
Не содержит бизнес-логики, только данные.

8. НАСТРОЙКА FRONTEND (REACT)

8.1. ФАЙЛ package.json

Создайте файл frontend/package.json:

```
{
  "name": "home-services-frontend",
  "private": true,
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.20.0",
    "axios": "^1.6.2",
    "@tanstack/react-query": "^5.12.2",
    "zustand": "^4.4.7",
    "react-hook-form": "^7.48.2",
    "zod": "^3.22.4",
    "@hookform/resolvers": "^3.3.2",
    "sockjs-client": "^1.6.1",
    "@stomp/stompjs": "^7.0.0",
    "react-hot-toast": "^2.4.1",
    "lucide-react": "^0.294.0",
    "clsx": "^2.0.0",
    "tailwind-merge": "^2.1.0"
  },
  "devDependencies": {
    "@types/react": "^18.2.43",
    "@types/react-dom": "^18.2.17",
    "@typescript-eslint/eslint-plugin": "^6.14.0",
    "@typescript-eslint/parser": "^6.14.0",
    "@vitejs/plugin-react": "^4.2.1",
    "autoprefixer": "^10.4.16",
    "eslint": "^8.55.0",
    "postcss": "^8.4.32",
    "tailwindcss": "^3.3.6",
    "typescript": "^5.2.2",
    "vite": "^5.0.8"
  }
}
```

8.2. API КЛИЕНТ (Axios)

Создайте файл frontend/src/api/client.ts:

```
import axios from 'axios';

const apiClient = axios.create({
  baseURL: '/api',
  headers: {
    'Content-Type': 'application/json',
  },
});

// Интерцептор для добавления JWT токена
apiClient.interceptors.request.use((config) => {
  const token = localStorage.getItem('token');
  if (token) {
    config.headers.Authorization = `Bearer ${token}`;
  }
  return config;
});

// Интерцептор для обработки ошибок
apiClient.interceptors.response.use(
  (response) => response,
  (error) => {
    if (error.response?.status === 401) {
      localStorage.removeItem('token');
      window.location.href = '/login';
    }
    return Promise.reject(error);
  }
);

export default apiClient;
```

ОБЪЯСНЕНИЕ:

- axios.create(): создание экземпляра Axios с базовыми настройками
- baseURL: базовый URL для всех запросов
- interceptors.request: добавление токена к каждому запросу
- interceptors.response: обработка ошибок (401 = неавторизован)

8.3. API СЕРВИСЫ

Создайте файл frontend/src/api/services.ts:

```
import apiClient from './client';

export interface User {
  id: number;
  username: string;
  email: string;
  firstName: string;
  lastName: string;
  role: 'CUSTOMER' | 'PROVIDER' | 'ADMIN';
}

export interface AuthRequest {
  username: string;
```

```

        password: string;
    }

export interface AuthResponse {
    token: string;
    user: User;
}

export const authApi = {
    register: (data: RegisterRequest) =>
        apiClient.post<AuthResponse>('/auth/register', data),
    login: (data: AuthRequest) =>
        apiClient.post<AuthResponse>('/auth/login', data),
};

```

ОБЪЯСНЕНИЕ:

- `apiClient.post<T>()`: POST запрос, возвращает данные типа Т
- Типизация TypeScript обеспечивает безопасность типов

8.4. ГЛАВНЫЙ КОМПОНЕНТ

Создайте файл `frontend/src/App.tsx`:

```

import { BrowserRouter, Routes, Route } from 'react-router-dom';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import Layout from './components/Layout';
import HomePage from './pages/HomePage';
import LoginPage from './pages/LoginPage';
import ServicesPage from './pages/ServicesPage';
import ProtectedRoute from './components/ProtectedRoute';

const queryClient = new QueryClient();

function App() {
    return (
        <QueryClientProvider client={queryClient}>
            <BrowserRouter>
                <Routes>
                    <Route path="/login" element={<LoginPage />} />
                    <Route path="/" element={<Layout />}>
                        <Route index element={<HomePage />} />
                        <Route path="services" element={<ServicesPage />} />
                        <Route
                            path="orders"
                            element={
                                <ProtectedRoute>
                                    <OrdersPage />
                                </ProtectedRoute>
                            }
                        />
                    </Route>
                </Routes>
            </BrowserRouter>
        </QueryClientProvider>
    );
}

export default App;

```

ОБЪЯСНЕНИЕ:

- BrowserRouter: роутинг в браузере
- QueryClientProvider: провайдер для React Query (кэширование запросов)
- Routes/Route: определение маршрутов
- ProtectedRoute: защита маршрутов (только для авторизованных)

8.5. СТРАНИЦА ЛОГИНА

Создайте файл frontend/src/pages/LoginPage.tsx:

```
import { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import { useMutation } from '@tanstack/react-query';
import { authApi } from '../api/services';

export default function LoginPage() {
  const navigate = useNavigate();
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');

  const loginMutation = useMutation({
    mutationFn: (data: { username: string; password: string }) =>
      authApi.login(data),
    onSuccess: (response) => {
      localStorage.setItem('token', response.data.token);
      navigate('/');
    },
  });

  const handleSubmit = (e: React.FormEvent) => {
    e.preventDefault();
    loginMutation.mutate({ username, password });
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
        placeholder="Username"
      />
      <input
        type="password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
        placeholder="Password"
      />
      <button type="submit">Login</button>
    </form>
  );
}
```

ОБЪЯСНЕНИЕ:

- useMutation: хук для мутаций (POST, PUT, DELETE)
- onSuccess: выполняется при успешном запросе
- localStorage: сохранение токена в браузере

- navigate: переход на другую страницу

9. НАСТРОЙКА DOCKER И DOCKER COMPOSE

9.1. DOCKERFILE ДЛЯ BACKEND

Создайте файл backend/Dockerfile:

```
# Этап сборки
FROM maven:3.9-eclipse-temurin-17 AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests

# Этап выполнения
FROM eclipse-temurin:17-jre
WORKDIR /app

# Установка curl для healthcheck
RUN apt-get update && apt-get install -y curl && rm -rf /var/lib/apt/lists/*

# Копирование JAR файла
COPY --from=build /app/target/backend-*.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "app.jar"]
```

ОБЪЯСНЕНИЕ:

- FROM: базовый образ
- WORKDIR: рабочая директория
- COPY: копирование файлов
- RUN: выполнение команд
- EXPOSE: открытие порта
- ENTRYPOINT: команда запуска

9.2. DOCKERFILE ДЛЯ FRONTEND

Создайте файл frontend/Dockerfile:

```
# Этап сборки
FROM node:18-alpine AS build
WORKDIR /app
COPY package*.json .
RUN npm ci
COPY .
RUN npm run build

# Этап выполнения
FROM nginx:alpine
RUN apk add --no-cache curl
COPY --from=build /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
```

```
CMD ["nginx", "-g", "daemon off;"]
```

9.3. NGINX КОНФИГУРАЦИЯ

Создайте файл frontend/nginx.conf:

```
server {  
    listen 80;  
    server_name localhost;  
    root /usr/share/nginx/html;  
    index index.html;  
  
    # SPA routing (для React Router)  
    location / {  
        try_files $uri $uri/ /index.html;  
    }  
  
    # API proxy (перенаправление на backend)  
    location /api {  
        proxy_pass http://backend:8080;  
        proxy_http_version 1.1;  
        proxy_set_header Host $host;  
        proxy_set_header X-Real-IP $remote_addr;  
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  
        proxy_set_header X-Forwarded-Proto $scheme;  
    }  
  
    # WebSocket proxy  
    location /api/ws {  
        proxy_pass http://backend:8080;  
        proxy_http_version 1.1;  
        proxy_set_header Upgrade $http_upgrade;  
        proxy_set_header Connection "upgrade";  
        proxy_set_header Host $host;  
    }  
}
```

ОБЪЯСНЕНИЕ:

- try_files: для SPA (Single Page Application) все запросы идут на index.html
- proxy_pass: перенаправление запросов /api на backend
- WebSocket: специальная настройка для WebSocket соединений

9.4. DOCKER COMPOSE КОНФИГУРАЦИЯ

Создайте файл docker-compose.yml в корне проекта:

```
services:  
  postgres:  
    image: postgres:16-alpine  
    container_name: home-services-postgres  
    environment:  
      POSTGRES_DB: home_services_db  
      POSTGRES_USER: postgres  
      POSTGRES_PASSWORD: postgrespassword  
    ports:  
      - "5432:5432"  
    volumes:
```

```

    - postgres_data:/var/lib/postgresql/data
networks:
    - home-services-network
healthcheck:
    test: ["CMD-SHELL", "pg_isready -U postgres"]
    interval: 10s
    timeout: 5s
    retries: 5

redis:
    image: redis:7-alpine
    container_name: home-services-redis
    ports:
        - "6379:6379"
    volumes:
        - redis_data:/data
    networks:
        - home-services-network
    healthcheck:
        test: ["CMD", "redis-cli", "ping"]
        interval: 10s
        timeout: 5s
        retries: 5

zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    container_name: home-services-zookeeper
    environment:
        ZOOKEEPER_CLIENT_PORT: 2181
        ZOOKEEPER_TICK_TIME: 2000
    ports:
        - "2181:2181"
    networks:
        - home-services-network
    healthcheck:
        test: ["CMD", "nc", "-z", "localhost", "2181"]
        interval: 10s
        timeout: 5s
        retries: 5

kafka:
    image: confluentinc/cp-kafka:7.5.0
    container_name: home-services-kafka
    depends_on:
        zookeeper:
            condition: service_healthy
    ports:
        - "9092:9092"
        - "9093:9093"
    environment:
        KAFKA_BROKER_ID: 1
        KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
        KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:29092,PLAINTEXT_HOST://0.0.0.0:9092
        KAFKA_ADVERTISED_LISTENERS:
            PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
        KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
            PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
        KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
        KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
        KAFKA_AUTO_CREATE_TOPICS_ENABLE: "true"
    volumes:
        - kafka_data:/var/lib/kafka/data
    networks:
        - home-services-network

```

```

healthcheck:
  test: ["CMD-SHELL", "kafka-broker-api-versions --bootstrap-server localhost:29092 || exit 1"]
    interval: 10s
    timeout: 10s
    retries: 30
    start_period: 60s

backend:
  build:
    context: ./backend
    dockerfile: Dockerfile
  container_name: home-services-backend
  depends_on:
    postgres:
      condition: service_healthy
    redis:
      condition: service_healthy
    kafka:
      condition: service_healthy
  ports:
    - "8080:8080"
  environment:
    SPRING_DATASOURCE_URL: jdbc:postgresql://postgres:5432/home_services_db
    SPRING_DATASOURCE_USERNAME: postgres
    SPRING_DATASOURCE_PASSWORD: postgrespassword
    SPRING_DATA_REDIS_HOST: redis
    SPRING_DATA_REDIS_PORT: 6379
    SPRING_KAFKA_BOOTSTRAP_SERVERS: kafka:29092
  networks:
    - home-services-network
  restart: unless-stopped
  healthcheck:
    test: ["CMD-SHELL", "curl -f http://localhost:8080/api/actuator/health || exit 1"]
      interval: 30s
      timeout: 10s
      retries: 5
      start_period: 120s

frontend:
  build:
    context: ./frontend
    dockerfile: Dockerfile
  container_name: home-services-frontend
  depends_on:
    backend:
      condition: service_started
  ports:
    - "3000:80"
  networks:
    - home-services-network
  restart: unless-stopped

volumes:
  postgres_data:
  redis_data:
  kafka_data:

networks:
  home-services-network:
    driver: bridge

```

ОБЪЯСНЕНИЕ:

- services: список сервисов (контейнеров)
- image: Docker образ для использования
- build: сборка образа из Dockerfile
- environment: переменные окружения
- ports: проброс портов (host:container)
- volumes: постоянное хранилище данных
- networks: сеть для связи контейнеров
- depends_on: зависимости между сервисами
- healthcheck: проверка здоровья контейнера

10. ОБЪЯСНЕНИЕ REDIS (КЭШИРОВАНИЕ)

10.1. ЧТО ТАКОЕ REDIS?

Redis (Remote Dictionary Server) - это in-memory хранилище данных типа ключ-значение. Данные хранятся в оперативной памяти, что обеспечивает очень быстрый доступ.

10.2. ЗАЧЕМ НУЖЕН REDIS В ПРОЕКТЕ?

1. КЭШИРОВАНИЕ ДАННЫХ:

- Часто запрашиваемые данные (категории, услуги) сохраняются в Redis
- При следующем запросе данные берутся из Redis (быстрее в 100 раз)
- Снижается нагрузка на базу данных PostgreSQL

2. ПОВЫШЕНИЕ ПРОИЗВОДИТЕЛЬНОСТИ:

- Чтение из памяти: ~0.1 мс
- Чтение с диска (PostgreSQL): ~10 мс
- Ускорение в 100 раз!

10.3. КАК ИСПОЛЬЗУЕТСЯ В ПРОЕКТЕ?

Пример использования в сервисе:

```
@Service
public class CategoryService {

    @Cacheable(value = "categories")
    public List<Category> getAllCategories() {
        // Этот метод выполнится только при первом вызове
        // Результат сохранится в Redis
        // При следующих вызовах данные возьмутся из Redis
        return categoryRepository.findAll();
    }

    @CacheEvict(value = "categories", allEntries = true)
    public Category createCategory(Category category) {
        // При создании новой категории кэш очищается
    }
}
```

```

        // При следующем запросе getAllCategories() данные обновятся
        return categoryRepository.save(category);
    }
}

```

ОБЪЯСНЕНИЕ АННОТАЦИЙ:

- **@Cacheable**: результат метода кэшируется
- **@CacheEvict**: очистка кэша при изменении данных

10.4. КОНФИГУРАЦИЯ REDIS

Создайте **backend/src/main/java/com/example/backend/config/RedisConfig.java**: файл

```

package com.example.backend.config;

import org.springframework.cache.CacheManager;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.cache.RedisCacheConfiguration;
import org.springframework.data.redis.cache.RedisCacheManager;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import
org.springframework.data.redis.serializer.GenericJackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.RedisSerializationContext;
import org.springframework.data.redis.serializer.StringRedisSerializer;

import java.time.Duration;

@Configuration
@EnableCaching
public class RedisConfig {

    @Bean
    public CacheManager cacheManager(RedisConnectionFactory connectionFactory)
    {
        RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheConfig()
                .entryTtl(Duration.ofHours(1)) // Время жизни кэша: 1 час

        .serializeKeysWith(RedisSerializationContext.SerializationPair
                .fromSerializer(new StringRedisSerializer()))

        .serializeValuesWith(RedisSerializationContext.SerializationPair
                .fromSerializer(new
GenericJackson2JsonRedisSerializer()));

        return RedisCacheManager.builder(connectionFactory)
                .cacheDefaults(config)
                .build();
    }
}

```

ОБЪЯСНЕНИЕ:

- **@EnableCaching**: включение кэширования в Spring
- **entryTtl**: время жизни записей в кэше (1 час)
- **serializeKeysWith**: сериализация ключей (String)

- serializeValuesWith: сериализация значений (JSON)

11. ОБЪЯСНЕНИЕ KAFKA (АСИНХРОННАЯ ОБРАБОТКА СОБЫТИЙ)

11.1. ЧТО ТАКОЕ KAFKA?

Apache Kafka - это распределенная платформа для обработки потоков событий в реальном времени. Позволяет публиковать и подписываться на потоки данных.

11.2. ЗАЧЕМ НУЖЕН KAFKA В ПРОЕКТЕ?

1. АСИНХРОННАЯ ОБРАБОТКА:

- Создание заказа не блокирует основной поток
- События обрабатываются независимо

2. РАЗДЕЛЕНИЕ ОТВЕТСТВЕННОСТИ:

- Сервис заказов создает заказ
- Сервис уведомлений отправляет email
- Сервис аналитики обновляет статистику
- Все независимо друг от друга

3. НАДЕЖНОСТЬ:

- События не теряются
- Можно обработать позже при сбое

11.3. КАК ИСПОЛЬЗУЕТСЯ В ПРОЕКТЕ?

Пример: Создание заказа

1. КЛИЕНТ СОЗДАЕТ ЗАКАЗ:

```
POST /api/orders
{
  "serviceId": 1,
  "scheduledDateTime": "2024-01-15T10:00"
}
```

2. OrderService СОЗДАЕТ ЗАКАЗ И ОТПРАВЛЯЕТ СОБЫТИЕ В KAFKA:

```
@Service
public class OrderService {
    private final KafkaTemplate<String, OrderEvent> kafkaTemplate;

    public Order createOrder(CreateOrderRequest request) {
        // Создание заказа в БД
        Order order = orderRepository.save(new Order(...));
    }
}
```

```

    // Отправка события в Kafka
    OrderEvent event = new OrderEvent();
    event.setOrderId(order.getId());
    event.setEventType("ORDER_CREATED");
    event.setTimestamp(LocalDateTime.now());

    kafkaTemplate.send("order-events", event);

    return order;
}
}

```

3. OrderEventConsumer ОБРАБАТЫВАЕТ СОБЫТИЕ:

```

@Component
public class OrderEventConsumer {

    @KafkaListener(topics = "order-events")
    public void handleOrderEvent(OrderEvent event) {
        if ("ORDER_CREATED".equals(event.getEventType())) {
            // Отправка уведомления клиенту
            notificationService.sendNotification(
                event.getCustomerId(),
                "Заказ создан",
                "Ваш заказ успешно создан"
            );

            // Отправка уведомления исполнителю
            notificationService.sendNotification(
                event.getProviderId(),
                "Новый заказ",
                "У вас новый заказ"
            );
        }
    }
}

```

11.4. КОНФИГУРАЦИЯ KAFKA

Создайте файл
[backend/src/main/java/com/example/backend/config/KafkaConfig.java](#):

```

package com.example.backend.config;

import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.support.serializer.JsonSerializer;

import java.util.HashMap;
import java.util.Map;

@Configuration
public class KafkaConfig {

```

```

    @Value("${spring.kafka.bootstrap-servers}")
    private String bootstrapServers;

    @Bean
    public ProducerFactory<String, Object> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
        bootstrapServers);
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        JsonSerializer.class);
        return new DefaultKafkaProducerFactory<>(configProps);
    }

    @Bean
    public KafkaTemplate<String, Object> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}

```

ОБЪЯСНЕНИЕ:

- ProducerFactory: фабрика для создания Kafka Producer
- KafkaTemplate: шаблон для отправки сообщений в Kafka
- StringSerializer: сериализация ключей (String)
- JsonSerializer: сериализация значений (JSON)

11.5. СОЗДАНИЕ СОБЫТИЯ

Создайте файл backend/src/main/java/com/example/backend/dto/OrderEvent.java:

```

package com.example.backend.dto;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.time.LocalDateTime;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class OrderEvent {
    private Long orderId;
    private Long customerId;
    private Long providerId;
    private String eventType; // ORDER_CREATED, ORDER_UPDATED, ORDER_CANCELLED
    private LocalDateTime timestamp;
    private Object data; // Дополнительные данные
}

```

12. СТРУКТУРА БАЗЫ ДАННЫХ POSTGRESQL

12.1. ОБЗОР БАЗЫ ДАННЫХ

Проект использует PostgreSQL - мощную реляционную базу данных. Схема базы данных создается автоматически при первом запуске приложения через Hibernate (ddl-auto=update).

12.2. ТАБЛИЦЫ БАЗЫ ДАННЫХ

12.2.1. Таблица users (Пользователи)

Хранит информацию о всех пользователях системы.

Поле	Тип	Описание
id	BIGSERIAL PRIMARY KEY	Уникальный идентификатор
username	VARCHAR UNIQUE NOT NULL	Имя пользователя (的独特的)
email	VARCHAR UNIQUE NOT NULL	Email адрес (unique)
password	VARCHAR NOT NULL	Хешированный пароль
first_name	VARCHAR NOT NULL	Имя
last_name	VARCHAR NOT NULL	Фамилия
phone	VARCHAR	Телефон (optional)
address	VARCHAR	Адрес (optional)
role	VARCHAR NOT NULL	Роль: CUSTOMER, PROVIDER, ADMIN
active	BOOLEAN NOT NULL	Активен ли пользователь (default: true)
created_at	TIMESTAMP NOT NULL	Дата создания аккаунта

Связи:

- Один ко многим с orders (как customer - клиент заказа)
- Один ко многим с services (как provider - исполнитель услуги)

Пример SQL запроса:

```
SELECT id, username, email, role, active FROM users;
```

12.2.2. Таблица categories (Категории услуг)

Хранит категории услуг (Уборка, Красота, Ремонт и т.д.).

Поле	Тип	Описание
id	BIGSERIAL PRIMARY KEY	Уникальный идентификатор
name	VARCHAR UNIQUE NOT NULL	Название категории (unique)
description	VARCHAR	Описание категории
icon	VARCHAR	Иконка (эмодзи)

Связи:

- Один ко многим с services (одна категория - много услуг)

Пример SQL запроса:

```
SELECT * FROM categories ORDER BY name;
```

12.2.3. Таблица services (Услуги)

Хранит информацию об услугах, которые предоставляют исполнители.

Поле	Тип	Описание
id	BIGSERIAL PRIMARY KEY	Уникальный идентификатор
name	VARCHAR NOT NULL	Название услуги
description	VARCHAR	Описание услуги
price	DECIMAL NOT NULL	Цена услуги
duration_minutes	INTEGER	Длительность в минутах
image_url	VARCHAR	URL изображения услуги
category_id	BIGINT NOT NULL	FK к categories (категория)
provider_id	BIGINT	FK к users (исполнитель)
active	BOOLEAN NOT NULL	Активна ли услуга (default: true)

Связи:

- Многие к одному с categories (каждая услуга принадлежит категории)
- Многие к одному с users (каждая услуга принадлежит исполнителю)
- Один ко многим с orders (одна услуга - много заказов)

Пример SQL запроса:

```
SELECT s.id, s.name, s.price, c.name as category, u.username as provider
FROM services s
JOIN categories c ON s.category_id = c.id
LEFT JOIN users u ON s.provider_id = u.id
WHERE s.active = true;
```

12.2.4. Таблица orders (Заказы)

Хранит информацию о заказах клиентов.

Поле	Тип	Описание
id	BIGSERIAL PRIMARY KEY	Уникальный идентификатор
customer_id	BIGINT NOT NULL	FK к users (клиент)
service_id	BIGINT NOT NULL	FK к services (услуга)

provider_id	BIGINT	FK к users (исполнитель)	
scheduled_date_time	TIMESTAMP NOT NULL	Запланированное время выполнения	
address	VARCHAR	Адрес выполнения заказа	
notes	VARCHAR	Примечания к заказу	
status	VARCHAR NOT NULL	Статус: PENDING, CONFIRMED, IN_PROGRESS, COMPLETED, CANCELLED	
total_price	DECIMAL NOT NULL	Общая стоимость заказа	
created_at	TIMESTAMP NOT NULL	Дата создания заказа	
completed_at	TIMESTAMP	Дата завершения заказа	

Связи:

- Многие к одному с users (customer - клиент заказа)
- Многие к одному с services (заказанная услуга)
- Многие к одному с users (provider - исполнитель заказа)
- Один к одному с reviews (один заказ - один отзыв)

Статусы заказа:

- PENDING - ожидает подтверждения
- CONFIRMED - подтвержден
- IN_PROGRESS - выполняется
- COMPLETED - завершен
- CANCELLED - отменен

Пример SQL запроса:

```
SELECT o.id,
       u1.username as customer,
       s.name as service,
       o.status,
       o.total_price,
       o.scheduled_date_time
  FROM orders o
 JOIN users u1 ON o.customer_id = u1.id
 JOIN services s ON o.service_id = s.id
 WHERE o.status = 'PENDING';
```

12.2.5. Таблица reviews (Отзывы)

Хранит отзывы клиентов о выполненных заказах.

Поле	Тип	Описание	
id	BIGSERIAL PRIMARY KEY	Уникальный идентификатор	
order_id	BIGINT NOT NULL	FK к orders (заказ)	
customer_id	BIGINT NOT NULL	FK к users (клиент, автор отзыва)	

provider_id BIGINT NOT NULL	FK к users (исполнитель)	
rating INTEGER NOT NULL	Оценка от 1 до 5	
comment VARCHAR	Текст отзыва	
created_at TIMESTAMP NOT NULL	Дата создания отзыва	

Связи:

- Многие к одному с orders (отзыв относится к заказу)
- Многие к одному с users (customer - автор отзыва)
- Многие к одному с users (provider - о ком отзыв)

Пример SQL запроса:

```
SELECT r.id,
       r.rating,
       r.comment,
       u1.username as customer,
       u2.username as provider,
       s.name as service
  FROM reviews r
 JOIN users u1 ON r.customer_id = u1.id
 JOIN users u2 ON r.provider_id = u2.id
 JOIN orders o ON r.order_id = o.id
 JOIN services s ON o.service_id = s.id
 ORDER BY r.created_at DESC;
```

12.3. СХЕМА СВЯЗЕЙ МЕЖДУ ТАБЛИЦАМИ

```
users (1) —————< (N) orders
users (1) —————< (N) services
users (1) —————< (N) reviews (как customer)
users (1) —————< (N) reviews (как provider)
categories (1) —————< (N) services
services (1) —————< (N) orders
orders (1) ————— (1) reviews
```

12.4. ТЕСТОВЫЕ ДАННЫЕ

При первом запуске приложения автоматически создаются тестовые данные (если база данных пуста).

12.4.1. Тестовые пользователи (5):

Username	Email	Password	Role
admin	admin@example.com	admin123	ADMIN
provider1	provider1@example.com	provider123	PROVIDER

provider2 provider2@example.com provider123 PROVIDER
customer1 customer1@example.com customer123 CUSTOMER
customer2 customer2@example.com customer123 CUSTOMER

12.4.2. Тестовые категории (6):

1. Уборка
2. Красота
3. Ремонт
4. Доставка
5. Обучение
6. Сад и огород

12.4.3. Тестовые услуги (15):

- 3 услуги по уборке
- 3 услуги красоты
- 3 ремонтные услуги
- 2 услуги доставки
- 2 услуги обучения
- 2 услуги по саду и огороду

12.5. ПОДКЛЮЧЕНИЕ К БАЗЕ ДАННЫХ

12.5.1. Параметры подключения:

- Host: localhost (или postgres в Docker сети)
- Port: 5432
- Database: home_services_db
- Username: postgres
- Password: postgrespassword

12.5.2. Подключение через Docker:

```
# Подключение к контейнеру PostgreSQL
docker exec -it home-services-postgres psql -U postgres -d home_services_db
```

```
# Полезные команды PostgreSQL:
\dt          # Показать все таблицы
\d users     # Описание таблицы users
\l          # Список всех баз данных
\q          # Выход
```

12.5.3. Подключение через SQL клиент:

Используйте любой SQL клиент (pgAdmin, DBeaver, DataGrip и т.д.):

- Host: localhost
- Port: 5432
- Database: home_services_db
- Username: postgres
- Password: postgrespassword

12.6. ПОЛЕЗНЫЕ SQL ЗАПРОСЫ

12.6.1. Просмотр всех пользователей с их ролями:

```
SELECT id, username, email, role, active, created_at
FROM users
ORDER BY created_at DESC;
```

12.6.2. Просмотр всех категорий с количеством услуг:

```
SELECT c.id, c.name, c.icon, COUNT(s.id) as services_count
FROM categories c
LEFT JOIN services s ON c.id = s.category_id AND s.active = true
GROUP BY c.id, c.name, c.icon
ORDER BY c.name;
```

12.6.3. Просмотр услуг с информацией о категории и исполнителе:

```
SELECT s.id,
       s.name,
       s.price,
       s.duration_minutes,
       c.name as category,
       u.username as provider,
       s.active
  FROM services s
 JOIN categories c ON s.category_id = c.id
 LEFT JOIN users u ON s.provider_id = u.id
 ORDER BY c.name, s.name;
```

12.6.4. Просмотр заказов с полной информацией:

```
SELECT o.id,
       u1.username as customer,
       u2.username as provider,
       s.name as service,
       o.status,
       o.total_price,
       o.scheduled_date_time,
       o.created_at
  FROM orders o
 JOIN users u1 ON o.customer_id = u1.id
 LEFT JOIN users u2 ON o.provider_id = u2.id
 JOIN services s ON o.service_id = s.id
 ORDER BY o.created_at DESC;
```

12.6.5. Статистика по заказам:

```
SELECT
    status,
    COUNT(*) as count,
    SUM(total_price) as total_revenue
FROM orders
GROUP BY status
ORDER BY count DESC;
```

12.6.6. Средний рейтинг исполнителей:

```
SELECT
    u.id,
    u.username,
    AVG(r.rating) as avg_rating,
    COUNT(r.id) as reviews_count
FROM users u
JOIN reviews r ON u.id = r.provider_id
WHERE u.role = 'PROVIDER'
GROUP BY u.id, u.username
HAVING COUNT(r.id) > 0
ORDER BY avg_rating DESC;
```

12.7. УПРАВЛЕНИЕ СХЕМОЙ БАЗЫ ДАННЫХ

12.7.1. Автоматическое создание схемы:

В application.properties настроено:
spring.jpa.hibernate.ddl-auto=update

Это означает, что Hibernate автоматически:

- Создает таблицы при первом запуске
- Обновляет схему при изменении моделей
- НЕ удаляет данные при изменении

12.7.2. Ручное управление (SQL):

```
# Создание таблицы вручную (если нужно)
CREATE TABLE users (
    id BIGSERIAL PRIMARY KEY,
    username VARCHAR(255) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    first_name VARCHAR(255) NOT NULL,
    last_name VARCHAR(255) NOT NULL,
    phone VARCHAR(255),
    address VARCHAR(255),
    role VARCHAR(50) NOT NULL,
    active BOOLEAN NOT NULL DEFAULT true,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

# Создание индексов для ускорения запросов
CREATE INDEX idx_users_username ON users(username);
```

```

CREATE INDEX idx_users_email ON users(email);
CREATE INDEX idx_orders_customer_id ON orders(customer_id);
CREATE INDEX idx_orders_status ON orders(status);
CREATE INDEX idx_services_category_id ON services(category_id);

```

13. ОСНОВНЫЕ КОМПОНЕНТЫ КОДА С ПРИМЕРАМИ

13.1. ПОЛНЫЙ ПРИМЕР: СОЗДАНИЕ ЗАКАЗА

BACKEND - OrderController.java:

```

@RestController
@RequestMapping("/orders")
@RequiredArgsConstructor
public class OrderController {
    private final OrderService orderService;

    @PostMapping
    @PreAuthorize("hasAnyRole('CUSTOMER', 'PROVIDER')")
    public ResponseEntity<OrderDto> createOrder(
        @Valid @RequestBody CreateOrderRequest request) {
        OrderDto order = orderService.createOrder(request);
        return ResponseEntity.ok(order);
    }
}

```

BACKEND - OrderService.java:

```

@Service
@RequiredArgsConstructor
public class OrderService {
    private final OrderRepository orderRepository;
    private final ServiceRepository serviceRepository;
    private final KafkaTemplate<String, OrderEvent> kafkaTemplate;

    public OrderDto createOrder(CreateOrderRequest request) {
        // 1. Получение услуги
        Service service = serviceRepository.findById(request.getServiceId())
            .orElseThrow(() -> new RuntimeException("Service not found"));

        // 2. Получение текущего пользователя
        User customer = SecurityUtil.getCurrentUser();

        // 3. Создание заказа
        Order order = new Order();
        order.setCustomer(customer);
        order.setService(service);
        order.setScheduledDateTime(request.getScheduledDateTime());
        order.setStatus(Order.OrderStatus.PENDING);
        order.setTotalPrice(service.getPrice());

        // 4. Сохранение в БД
        order = orderRepository.save(order);

        // 5. Отправка события в Kafka
        OrderEvent event = new OrderEvent();
        event.setOrderId(order.getId());
        event.setCustomerId(customer.getId());
        event.setProviderId(service.getProvider().getId());
    }
}

```

```

        event.setEventType("ORDER_CREATED");
        event.setTimestamp(LocalDateTime.now());

        kafkaTemplate.send("order-events", event);

        // 6. Возврат DTO
        return convertToDto(order);
    }
}

```

FRONTEND - OrdersPage.tsx:

```

import { useMutation, useQuery } from '@tanstack/react-query';
import { orderApi } from '../api/services';

export default function OrdersPage() {
    // Получение списка заказов
    const { data: orders, isLoading } = useQuery({
        queryKey: ['orders'],
        queryFn: () => orderApi.getMyOrders(),
    });

    // Создание нового заказа
    const createOrderMutation = useMutation({
        mutationFn: (data: CreateOrderRequest) => orderApi.create(data),
        onSuccess: () => {
            // Обновление списка заказов
            queryClient.invalidateQueries({ queryKey: ['orders'] });
        },
    });

    const handleCreateOrder = (serviceId: number) => {
        createOrderMutation.mutate({
            serviceId,
            scheduledDateTime: new Date().toISOString(),
        });
    };

    if (isLoading) return <div>Loading...</div>

    return (
        <div>
            {orders?.map(order => (
                <div key={order.id}>
                    <h3>{order.service.name}</h3>
                    <p>Status: {order.status}</p>
                </div>
            )));
        </div>
    );
}

```

13.2. ПРИМЕР: АУТЕНТИФИКАЦИЯ С JWT

BACKEND - JwtTokenProvider.java:

```

@Component
@RequiredArgsConstructor
public class JwtTokenProvider {
    @Value("${jwt.secret}")

```

```

private String secret;

@Value("${jwt.expiration}")
private long expiration;

public String generateToken(User user) {
    Date now = new Date();
    Date expiryDate = new Date(now.getTime() + expiration);

    return Jwts.builder()
        .setSubject(user.getUsername())
        .claim("userId", user.getId())
        .claim("role", user.getRole().name())
        .setIssuedAt(now)
        .setExpiration(expiryDate)
        .signWith(SignatureAlgorithm.HS512, secret)
        .compact();
}

public boolean validateToken(String token) {
    try {
        Jwts.parser().setSigningKey(secret).parseClaimsJws(token);
        return true;
    } catch (JwtException | IllegalArgumentException e) {
        return false;
    }
}
}

```

FRONTEND - authStore.ts (Zustand):

```

import { create } from 'zustand';

interface AuthState {
    token: string | null;
    user: User | null;
    setAuth: (token: string, user: User) => void;
    logout: () => void;
}

export const useAuthStore = create<AuthState>((set) => ({
    token: localStorage.getItem('token'),
    user: null,
    setAuth: (token, user) => {
        localStorage.setItem('token', token);
        set({ token, user });
    },
    logout: () => {
        localStorage.removeItem('token');
        set({ token: null, user: null });
    },
}));
```

14. ЗАПУСК ПРОЕКТА

14.1. ПОДГОТОВКА

Убедитесь, что установлены:
- Docker Desktop запущен

- Порты 3000, 8080, 5432, 6379, 9092 свободны

14.2. КОМАНДЫ ДЛЯ ЗАПУСКА

```
# Перейдите в корневую папку проекта
cd Проект_СТСР

# Остановите и удалите старые контейнеры (если есть)
docker-compose down -v

# Соберите и запустите все сервисы
docker-compose up -d --build

# Проверьте статус контейнеров
docker-compose ps

# Просмотрите логи
docker-compose logs -f

# Просмотрите логи конкретного сервиса
docker-compose logs -f backend
docker-compose logs -f frontend
```

14.3. ПРОВЕРКА РАБОТЫ

1. Frontend: <http://localhost:3000>
2. Backend API: <http://localhost:8080/api>
3. Swagger UI: <http://localhost:8080/api/swagger-ui.html>
4. Health Check: <http://localhost:8080/api/actuator/health>

14.4. ОСТАНОВКА ПРОЕКТА

```
# Остановка контейнеров
docker-compose stop

# Остановка и удаление контейнеров
docker-compose down

# Остановка, удаление контейнеров и volumes (удалит данные БД!)
docker-compose down -v
```

14.5. ПЕРЕСБОРКА ПРОЕКТА

```
# Пересборка без кэша
docker-compose build --no-cache

# Пересборка и запуск
docker-compose up -d --build
```

15. ТЕСТИРОВАНИЕ ПРИЛОЖЕНИЯ

15.1. ТЕСТОВЫЕ ПОЛЬЗОВАТЕЛИ

При первом запуске создаются тестовые пользователи:

Роль	Username	Password
Admin	admin	admin123
Provider	provider1	provider123
Customer	customer1	customer123

15.2. ТЕСТИРОВАНИЕ API ЧЕРЕЗ SWAGGER

1. Откройте <http://localhost:8080/api/swagger-ui.html>
2. Нажмите "Authorize" и введите токен
3. Протестируйте endpoints

15.3. ТЕСТИРОВАНИЕ ЧЕРЕЗ CURL

```
# Регистрация
curl -X POST http://localhost:8080/api/auth/register \
-H "Content-Type: application/json" \
-d '{
    "username": "testuser",
    "email": "test@example.com",
    "password": "password123",
    "firstName": "Test",
    "lastName": "User"
}'

# Вход
curl -X POST http://localhost:8080/api/auth/login \
-H "Content-Type: application/json" \
-d '{
    "username": "admin",
    "password": "admin123"
}'

# Получение категорий (с токеном)
curl -X GET http://localhost:8080/api/categories \
-H "Authorization: Bearer YOUR_TOKEN"
```

15.4. ТЕСТИРОВАНИЕ FRONTEND

1. Откройте <http://localhost:3000>
2. Зарегистрируйтесь или войдите
3. Протестируйте функциональность:
 - Просмотр услуг
 - Создание заказа
 - Просмотр заказов
 - Административная панель (для admin)

16. ЗАКЛЮЧЕНИЕ

Данное методическое пособие содержит всю необходимую информацию для создания полнофункционального клиент-серверного приложения с использованием современных технологий:

- Spring Boot для backend
- React для frontend
- PostgreSQL для хранения данных
- Redis для кэширования
- Kafka для асинхронной обработки событий
- Docker для контейнеризации

После выполнения всех шагов у вас будет рабочее приложение, которое можно развивать и масштабировать.

ВАЖНЫЕ МОМЕНТЫ:

1. Всегда используйте Docker Compose для запуска проекта
2. Проверяйте логи при возникновении проблем
3. Используйте Swagger для тестирования API
4. Следите за версиями зависимостей
5. Регулярно делайте бэкапы базы данных