Service marketplace

Архитектурный документ

Авторы документа:

Нестеров Кирилл Валерьевич БПИ218

Шаламкова Алиса Станиславовна БПИ218

Преподаватель группы:

Пашигорев Кирилл Игоревич

0. Раздел регистрации изменений

Версия документа	Дата изменения	Описание изменения	Автор изменения
1.0.0	26.03.2024	Initial	Нестеров К. В.
1.0.1	03.06.2024	Описание доработок проекта (помечены цветом)	Шаламкова А.С.

1. Введение

1.1. Название проекта

Service Marketplace

1.2. Задействованные архитектурные представления

Для описания проекта будут использованы следующие представления: Диаграмма прецедентов, User-story диаграммы, пакетная и классовая диаграммы, а также диаграмма компонентов. В дополнительных материалах приложены sequence диаграммы.

1.3. Контекст задачи и среда функционирования системы

Разрабатываемая система представляет собой маркетплейс юридических услуг, созданный для облегчения взаимодействия между заказчиками и исполнителями юридических услуг. Она решает задачу обеспечения доступа к актуальным юридическим услугам и обеспечивает возможность для пользователей заказывать услуги и находить подходящих исполнителей.

1.4. Рамки и цели проекта

Project Scope:

Включено в проект:

- Разработка и внедрение веб-приложения маркетплейса юридических услуг, включая клиентскую и серверную части.
- Авторизация пользователей через JWT токены.
- Возможность просмотра объявлений и поиска по ключевым словам и фильтрам.
- Создание объявлений о поиске исполнителей и услуг.
- Система чатов для общения между заказчиками и исполнителями.
- Возможность оценивать работу и оставлять отзывы на заказанные услуги.
- Публичное отображение рейтинга исполнителей и заказчиков.
- Оплата заказа пользователя.
- Раздел с шаблонами юридических документов.

Цель проекта:

Основная цель проекта заключается в создании онлайн-платформы, которая связывает пользователей, нуждающихся в юридических услугах, с профессиональными исполнителями данных услуг. Система предоставляет удобный и эффективный механизм для заказа и предоставления юридических услуг, обеспечивая прозрачность и безопасность взаимодействия между сторонами.

Стек проекта:

Backend:

- Язык программирования: Java
- Фреймворк: Spring (для быстрой разработки и управления веб-приложением).
- СУБД: В качестве базы данных используется PostgreSQL, так как он обладает хорошей производительностью и достаточной надежностью для веб-приложений.
- IDE: IntelliJ IDEA для разработки и отладки Java-кода.
- ORM: Hibernate для удобной работы с базой данных из Java-приложения.

Frontend:

- Язык программирования: JavaScript
- Фреймворк: React.js для построения пользовательского интерфейса и управления состоянием приложения.

- Инструменты для разработки: Visual Studio Code, npm для управления зависимостями и сборки проекта.

Инструменты для деплоя:

- Хостинг: Для размещения веб-приложения будет использоваться облачная платформа AWS(Amazon Web Services)
- Контейнеризация: Docker используется для упаковки приложения и его зависимостей в контейнеры, обеспечивая единообразное развертывание в различных окружениях.
- Оркестратор контейнеров: Kubernetes используется для управления и автоматизации развертывания приложения в облачной среде.
- Инструменты CI/CD: Jenkins используется для непрерывной интеграции и доставки приложения, автоматизирующие процессы сборки, тестирования и развертывания.

Инструменты для тестирования:

- Библиотека языка Java Mockito и фреймворк Junit
- Библиотека Jacoco для покрытия тестами

2. Архитектурные факторы

2.1. Ключевые заинтересованные лица

Действующее лицо	**Заинтересованность в системе**
Разработчики	Разработчики заинтересованы в создании и поддержке функционального, надежного и безопасного веб-приложения, соответствующего требованиям и ожиданиям пользователей.
Тестировщики	Тестировщики заинтересованы в обеспечении качества программного продукта путем выявления и устранения ошибок, проверке корректности работы функций и соответствия требованиям к системе.
Владельцы продукта (вижн холдеры)	Владельцы продукта заинтересованы в создании успешного и конкурентоспособного продукта, который удовлетворяет потребности и ожидания пользователей, приносит прибыль и улучшает репутацию компании.
Пользователи	Пользователи заинтересованы в доступе к широкому выбору юридических услуг, удобном и быстром поиске исполнителей, простом и безопасном процессе заказа услуг, а также в получении качественных и своевременных услуг со стороны исполнителей.
Модераторы	Модераторы заинтересованы в поддержании порядка и безопасности на платформе, проверке объявлений и профилей пользователей, разрешении споров и решении возникающих проблем для обеспечения позитивного опыта всех участников системы.

2.2. Ключевые требования к системе

Ключевые требования к системе:

1. Скорость
2. Надежность
3. Расширяемость
4. Безопасность

2.3. Ключевые ограничения

Ключевые ограничения для этого проекта включают ограниченные ресурсы на разработку и поддержку системы, необходимость соблюдения законодательства о защите данных и конфиденциальности, а также требования к безопасности и защите от несанкционированного доступа. Также важно учитывать ограничения в доступности и качестве интернет-соединения для пользователей, особенно если система предполагает использование в различных регионах с разной инфраструктурой.

3. Общее архитектурное решение

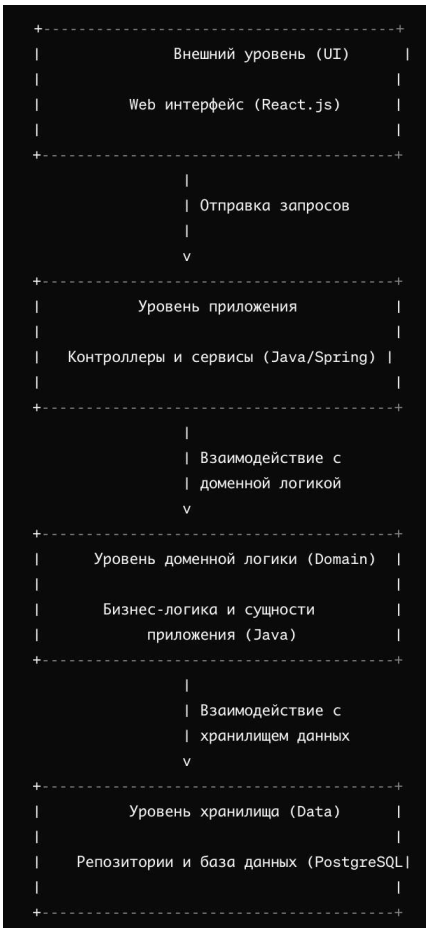
Для реализации данного продукта была выбрана “Чистая” архитектура.

Основные принципы "чистой архитектуры":

- Разделение системы на уровни, каждый из которых отвечает за определенные аспекты бизнес-логики и отделяется от других уровней.
- Независимость внутренних компонентов системы от внешних фреймворков и библиотек.

Структура системы:

- Внешний уровень (UI): Веб-интерфейс для взаимодействия пользователей с системой. Отвечает за представление данных и обработку пользовательских действий.
- Уровень приложения (Application): Контроллеры и сервисы, обеспечивающие бизнес-логику и координацию взаимодействия между различными компонентами.
- Уровень доменной логики (Domain): Содержит основные сущности бизнес-логики, правила и логику работы с данными, не зависящие от конкретных технических решений.
- Уровень хранилища (Data): Взаимодействие с базой данных и другими внешними источниками данных.



Обоснование выбора архитектуры:

- "Чистая архитектура" обеспечивает высокую гибкость и расширяемость системы, упрощает тестирование и поддержку кода.
- Разделение на уровни позволяет легко внедрять изменения в каждой части системы независимо друг от друга, а также обеспечивает четкое разделение ответственности между компонентами.
- Независимость от фреймворков обеспечивает возможность легкого изменения и обновления технологий в будущем, а также уменьшает риск возникновения зависимостей от конкретных решений.

3.1. Принципы проектирования

Основные принципы, используемые командой на этапе проектирования:

Принцип модульности:

- Разделение системы на независимые модули, каждый из которых отвечает за определенную функциональность. Это позволяет легко масштабировать, тестировать и поддерживать систему.

Принцип гибкости:

- Создание архитектуры, которая легко адаптируется к изменяющимся потребностям и требованиям рынка. Это включает в себя использование модульной архитектуры, управление зависимостями и принципы SOLID.

Принцип удобства использования (User-Centered Design):

- Основное внимание уделяется потребностям и удобству пользователей. Продукт должен быть интуитивно понятным, легким в использовании и отвечать потребностям целевой аудитории.

Принцип безопасности:

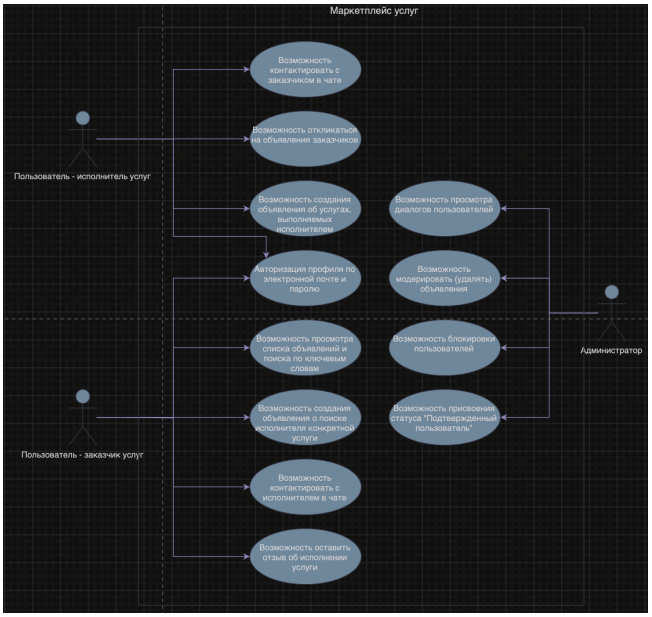
- Защита данных пользователей, конфиденциальность информации и обеспечение безопасности транзакций. Это включает в себя использование современных методов шифрования, защиту от атак и уязвимостей.

Принцип масштабируемости:

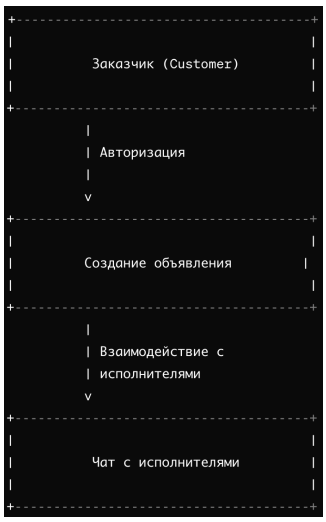
- Создание системы, которая легко масштабируется в зависимости от роста бизнеса и количества пользователей. Это включает в себя использование распределенной архитектуры, облачных технологий и горизонтальное масштабирование.

4. Архитектурные представления

4.1. Представление прецедентов

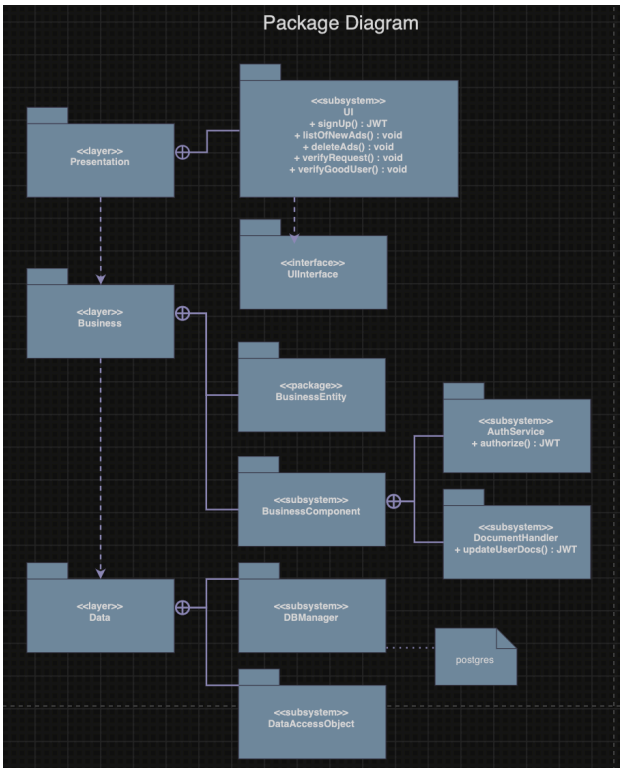


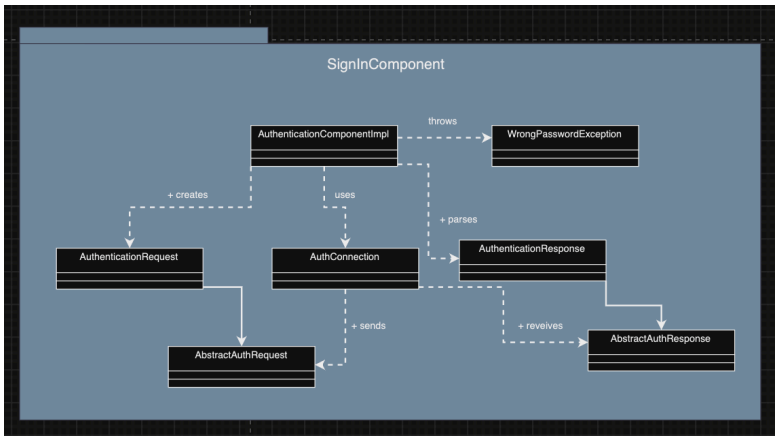
User-story для Пользователя-заказчика:

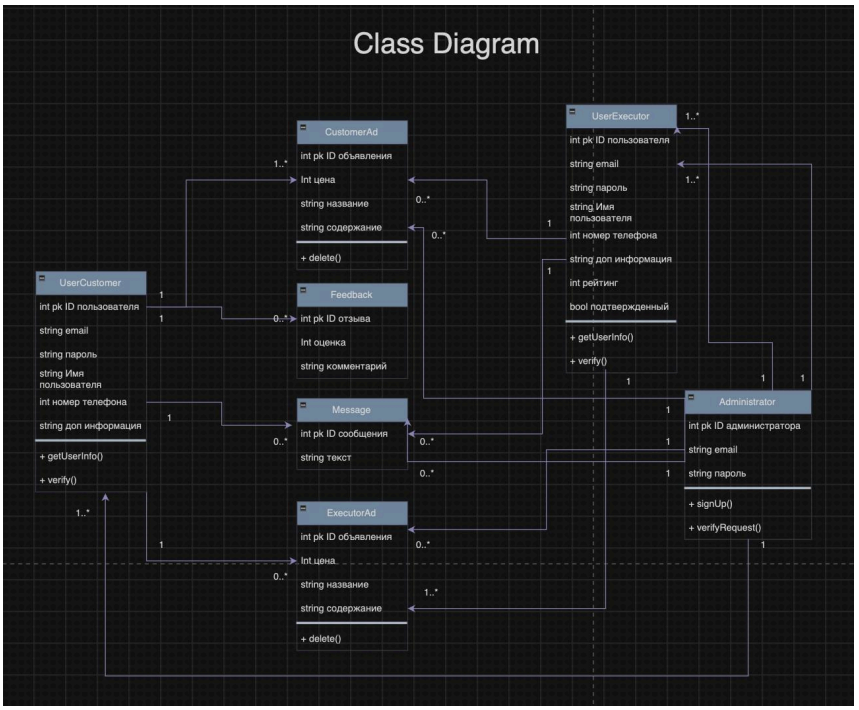


User-story для пользователя-исполнителя:

4.2. Логическое представление







4.3. Представление архитектуры процессов

В системе будут исполняться процессы создания объявлений заказов, откликов на эти объявления, взаимодействие через чат между заказчиками и исполнителями, а также оставление отзывов. Задачи системы, связанные с созданием и обработкой заказов, могут быть распараллелены для повышения эффективности и отзывчивости системы. Для предотвращения конфликтов при одновременном использовании услуг асинхронный код будет проработан и реализован для обеспечения корректной работы системы.

4.4. Физическое представление архитектуры

Система маркетплейса юридических услуг будет работать на серверах, предоставляемых облачным провайдером. Обычно такие серверы оснащены мощными процессорами, большим объемом оперативной памяти и быстрым доступом к хранилищу данных. Для хранения данных будет использоваться реляционная база данных, такая как PostgreSQL, работающая на отдельном сервере или в облачном хранилище. Клиентские устройства пользователей будут взаимодействовать с серверами через интернет, используя веб-браузеры для доступа к веб-приложению. Кроме того, для обеспечения безопасного обмена данными между клиентскими устройствами и серверами может быть использован протокол HTTPS.

4.5. Представление развертывания

Система будет разворачиваться в облачной среде, такой как AWS, Google Cloud Platform или Microsoft Azure.

Отдельно устанавливаемыми компонентами и модулями будут:

Frontend приложение: Веб-приложение на основе React.js, которое будет развернуто на веб-сервере, таком как Nginx или Apache.

Backend приложение: Серверная часть на основе Java с использованием фреймворка Spring Boot. Оно будет развернуто на сервере приложений, таком как Tomcat или Jetty.

База данных: Реляционная база данных, такая как PostgreSQL, будет установлена на отдельном сервере или будет использоваться управляемая база данных в облачной среде.

API Gateway: Для обеспечения безопасного и эффективного взаимодействия между клиентской и серверной частями системы можно использовать API Gateway, например, Amazon API Gateway.

Артефакты: Для заказчика будут поставляться исполняемые файлы (jar для Java-приложения, статические файлы для веб-приложения), конфигурационные файлы и инструкции по развертыванию.

Обновление системы будет осуществляться путем выкладывания новых версий приложений на серверы, а также обновлением конфигураций и ресурсов в облачной среде. Для обеспечения непрерывной работы системы во время обновления можно использовать методы blue-green deployment или канареечное развертывание.

4.6. Представление архитектуры данных

Система маркетплейса юридических услуг будет работать с различными типами данных, включая информацию о пользователях (заказчиках и исполнителях), объявлениях о услугах, чатах между пользователями, отзывах и оценках, а также описания юридических услуг и шаблоны документов. Данные будут интегрироваться в базу данных через серверное приложение, которое будет обеспечивать CRUD операции.

Репликация данных может быть реализована для обеспечения отказоустойчивости и масштабируемости системы.

Потоки данных в системе будут осуществляться через API, который будет предоставлять доступ к данным клиентским приложениям и обеспечивать взаимодействие между различными компонентами системы. Управление транзакциями в приложении поможет избежать гонку данных.

4.7. Представление архитектуры безопасности

Для обеспечения безопасности и сохранности данных в системе маркетплейса юридических услуг будут применены следующие меры:
Шифрование данных: Критические данные, такие как пароли пользователей и конфиденциальная информация, будут храниться в зашифрованном виде. Для обмена данными между клиентскими приложениями и серверами будет использоваться протокол HTTPS.
Механизмы аутентификации и авторизации: Пользователи будут проходить аутентификацию посредством введения уникальных учетных данных (например, электронной почты и пароля). для безопасного и эффективного обмена данными между клиентом и сервером будет использован JWT (JSON Web Tokens).
Ограничение доступа: Каждый пользователь будет иметь определенные права доступа к данным и функциональности системы в зависимости от его роли (например, заказчик или исполнитель).
Резервное копирование данных: Регулярные резервные копии данных будут создаваться и храниться в безопасном месте для обеспечения сохранности данных и возможности восстановления в случае сбоев или потери данных.

4.8. Представление реализации и разработки

IDEA и VSCode для разработки, GitHub для контроля версий, Junit для модульного тестирования и Jenkins для CI/CD. Jira для трека задач)

4.9. Представление производительности

Команда разработчик не приводит информацию об используемых алгоритмах.

4.10. Атрибуты качества системы

Безопасность, надежность, производительность, масштабируемость, модульность, согласованность, отказоустойчивость

4.10.1. Объем данных и производительность системы

Объем данных: Система должна быть способна обрабатывать большие объемы данных, включая профили пользователей, объявления о услугах, чаты между пользователями, отзывы и оценки. При этом количество пользователей и объем информации могут постоянно расти. Таким образом, система должна быть масштабируемой и готовой к увеличению объема данных.
Производительность: Система должна быть высокопроизводительной и отзывчивой, обеспечивая быстрый доступ к данным и оперативное выполнение запросов пользователей. Например, время загрузки страниц и выполнения запросов к базе данных должно быть минимальным, не превышающим несколько секунд. Также важно обеспечить высокую скорость обработки чатов и операций по созданию и обновлению объявлений. Допускается недоступность системы на уровне 0.01% или не более 4 минут в месяц.

4.10.2. Гарантии качества работы системы

Тестирование: модульное, нагрузочное, интеграционное. Сбор метрик и мониторинг (Prometheus для сбора, Grafana для отрисовки дешбордов). Журналирование/логирование, автоматизация отслеживания сбоев (нестандартного поведения системы, заметного на графиках)

Для обеспечения высокого качества кода и надежности системы, в проект были добавлены юнит-тесты. Эти тесты помогают выявлять ошибки на ранних стадиях разработки и обеспечивают корректную работу всех функциональных компонентов. Юнит-тесты были реализованы с использованием фреймворка Junit.

Процесс разработки тестов:

1. В проекте используется идеология TDD, однако тестов в проекте нет. Были дописаны примеры unit-тестов для контроллера DialogController. Аналогично стоит прописать тесты для остальных модулей проекта.

2. Запуск тестов для проверки корректности работы новой функциональности.

3. Регулярное обновление и дополнение тестов по мере развития проекта.

Пример тестов (ссылка на коммит: https://github.com/arisumerray/service-marketplace/tree/develop/src/test/java/org/example/controller):

```
package org.example.controller;


import jakarta.persistence.EntityExistsException;
import org.example.dto.DialogDto;
import org.example.dto.FullDialogDto;
import org.example.dto.UserForDialogDto;
import org.example.entity.Dialog;
import org.example.entity.Message;
import org.example.entity.User;
import org.example.repository.DialogRepository;
import org.example.repository.UserRepository;
import org.example.service.DialogService;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mapstruct.control.MappingControl;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.test.web.servlet.MockMvc;


import javax.management.remote.JMXPrincipal;
import java.security.Principal;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Optional;


import static org.junit.jupiter.api.Assertions.*;
```

```
import static org.mockito.Mockito.when;


class DialogControllerTest {

   @InjectMocks

   private DialogController dialogController;


   @Mock

   private DialogService dialogService;


   @Mock

   private UserRepository userRepository;




   @BeforeEach

   public void setUp() {

       MockitoAnnotations.openMocks(this);

   }


   @Test

   public void testGetAllDialogs() throws Exception {

       ArrayList<Message> messageList = new ArrayList<Message>(Arrays.asList(new Message(1,"a",null, null, null)));

       ArrayList<User> userList = new ArrayList<User>(Arrays.asList(new User(1, "a", "", "", "","", "",null, null,

               null, null, null, null, null)));

       Dialog dialog3 = new Dialog(1, messageList, userList);

       List<UserForDialogDto> userDtoList = new ArrayList<UserForDialogDto>(Arrays.asList(new UserForDialogDto(1, "","")));

       ArrayList<DialogDto> dtos = new ArrayList<>(Arrays.asList(new DialogDto(1, userDtoList)));

       when(dialogService.getAllDialogs(1)).thenReturn(dtos);

       User u = new User(1, "a", "", "", "","", "",null, null,

               null, null, null, null, null);

       when(userRepository.findByEmail("")).thenReturn(Optional.of(u));


       ResponseEntity<List<DialogDto>> response = dialogController.getAllDialogs(new JMXPrincipal(""));


       assertEquals(200, response.getStatusCodeValue());

       assertEquals(1, response.getBody().size());

   }


   @Test

   void getDialog() {

       ArrayList<Message> messageList = new ArrayList<Message>(Arrays.asList(new Message(1,"a",null, null, null)));

       ArrayList<User> userList = new ArrayList<User>(Arrays.asList(new User(1, "a", "", "", "","", "",null, null,

               null, null, null, null, null)));

       Dialog dialog = new Dialog(1, messageList, userList);

       ArrayList<UserForDialogDto> userForDtoList = new ArrayList<>(Arrays.asList(new UserForDialogDto(1,"a","b")));

       ArrayList<Message> messageList2 = new ArrayList<Message>(Arrays.asList(new Message(1,"a",null, null, null)));

       FullDialogDto dDto = new FullDialogDto(1, userForDtoList, messageList2);

       when(dialogService.getDialog(1,"")).thenReturn(dDto);

       when(dialogService.getDialog(null,"")).thenReturn(dDto);


       ResponseEntity<?> response = dialogController.getDialog(1, new JMXPrincipal(""));

       ResponseEntity<?> response2 = dialogController.getDialog(null, new JMXPrincipal(" "));


       assertEquals(200, response.getStatusCodeValue());
```

```
        assertEquals(200, response2.getStatusCodeValue());
        assertNotNull(response.getBody());
        assertNull(response2.getBody());
    }

    @Test
    void createDialog() {
        ArrayList<Message> messageList = new ArrayList<Message>(Arrays.asList(new Message(1,"a",null, null, null)));
        ArrayList<User> userList = new ArrayList<User>(Arrays.asList(new User(1, "a", "", "", "","", "",null, null,
                null, null, null, null, null)));
        Dialog dialog = new Dialog(1, messageList, userList);
        when(dialogService.createDialog(1,2)).thenReturn(dialog);
        when(dialogService.createDialog(1,3)).thenThrow(new EntityExistsException ("bebebe"));
        User u = new User(2, "", "", "", "","", "",null, null,
                null, null, null, null, null);
        User u2 = new User(3, " ", "", "", "","", "",null, null,
                null, null, null, null, null);
        when(userRepository.findByEmail("")).thenReturn(Optional.of(u));
        when(userRepository.findByEmail(" ")).thenReturn(Optional.of(u2));

        ResponseEntity<?> response = dialogController.createDialog(1, new JMXPrincipal(""));
        ResponseEntity<?> response2 = dialogController.createDialog(1, new JMXPrincipal(" "));
        assertEquals(200, response.getStatusCodeValue());
        assertNotNull(response.getBody());
        assertEquals(409, response2.getStatusCodeValue());
        assertNotNull(response2.getBody());
    }
}
```

5. Технические описания отдельных ключевых архитектурных решений

5.1. Техническое описание решения: идеология Test Driven Development

Не реализована в проекте.

5.1.1. Проблема

Как обеспечить высокое качество кода и функциональность системы?

5.1.2. Идея решения

Использовать методологию Test-Driven Development (TDD), при которой сначала пишутся тесты, затем функциональность для их прохождения.

5.1.3. Факторы

- TDD позволяет выявить и устранить ошибки на ранних стадиях разработки, что способствует созданию более надежного и стабильного кода.
- Этот подход упрощает процесс разработки, так как разработчики сосредотачиваются на необходимой функциональности и ее корректности с самого начала.

5.1.4. Решение

Разработчики будут следовать подходу TDD, который включает следующие этапы:

- Написание теста для новой функциональности или исправления.
- Запуск теста, который должен завершиться неудачно, так как соответствующая функциональность еще не реализована.
- Написание минимального кода, необходимого для прохождения теста.
- Запуск теста, который должен завершиться успешно.
- Рефакторинг кода, если необходимо, для улучшения его структуры и читаемости.

5.1.5. Мотивировка

Выбор TDD обеспечивает высокое качество кода и функциональности системы за счет того, что каждая часть функциональности тщательно тестируется перед ее реализацией. Это также способствует более быстрой разработке, так как обнаруженные ошибки исправляются на ранних этапах.

5.1.6. Неразрешенные вопросы

Отсутствуют

5.1.7. Альтернативы

Вместо использования TDD можно было рассмотреть другие методологии разработки, такие как Behavior-Driven Development (BDD) или traditional development. Однако, TDD выбрано из-за своей эффективности в обеспечении высокого качества кода и быстрой разработки.

5.2. Техническое описание решения: Использование RESTful API

5.2.1. Проблема

Как обеспечить эффективное взаимодействие между клиентской и серверной частями системы?

5.2.2. Идея решения

Использовать архитектурный стиль RESTful API для организации коммуникации между клиентом и сервером.

5.2.3. Факторы

- RESTful API предоставляет простой и гибкий способ взаимодействия между различными компонентами системы.
- Это позволяет обеспечить масштабируемость и гибкость в разработке и поддержке системы.
- Требования к производительности и надежности могут быть эффективно удовлетворены с помощью RESTful API.

5.2.4. Решение

- Для реализации RESTful API будет использован фреймворк Spring Boot для backend и фреймворк React.js для frontend. API будет построено в соответствии с принципами REST, включая использование HTTP методов (GET, POST, PUT, DELETE) для выполнения операций над ресурсами, а также формат данных JSON для обмена информацией между клиентом и сервером.

5.2.5. Мотивировка

Выбор RESTful API обеспечивает простоту, гибкость и масштабируемость взаимодействия между компонентами системы. Это позволит легко добавлять новые функции, модифицировать существующие и обеспечить стабильную работу системы при изменении требований.

5.2.6. Неразрешенные вопросы

Отсутствуют

5.2.7. Альтернативы

В качестве альтернативы использованию RESTful API можно было рассмотреть технологии, такие как GraphQL или SOAP. Однако, RESTful API выбрано из-за своей простоты, гибкости и широкой поддержки в индустрии.

5.3. Техническое описание решения: Реализация оплаты покупки

5.3.1. Проблема

До внесения изменений процесс оплаты был захардкожен - оплачивать заказы было нельзя, стояла заглушка.

5.2.2. Идея решения

Добавить в PaymentService зависимость ServiceOfferService для получения данных заказа по id заказа и создания платежа.

5.2.3. Решение

1. Добавить новый метод в ServiceOfferRepository получения данных заказа из БД по id заказа.
2. Добавление нового метода в интерфейс ServiceOfferService для получения сущности ServiceOffer с информацией из БД.
3. Реализация описанного в п.2 метода в классе ServiceOfferServiceImpl
4. Внедрение в PaymentService зависимости ServiceOfferService

5.2.4. Преимущества решения

Возможность оплатить любой заказ по id.

5.2.6. Технологический стек

Для реализации решения использовались следующие технологии:

- **Java**: Основной язык программирования.
- **Spring Framework**: Для управления зависимостями.
- **Maven/Gradle**: Для управления проектом и его зависимостями.
- **YooKassa SDK:** https://github.com/dynomake/yookassa-java-sdk

5.2.7. Альтернативы

Можно было избежать использование библиотеки по обработке платежей YooKassa SDK, вместо этого можно было напрямую работать с API YooKassa.

5.2.8. Заключение

Реализованное решение обеспечивает гибкую, надежную и безопасную обработку платежей, что улучшает пользовательский опыт и упрощает техническую поддержку системы. Ссылка: https://github.com/kirnes04/service-marketplace/compare/develop...arisumerray:service-marketplace:develop

PaymentController.java | ServiceOfferServiceImpl.java | ServiceOfferService.java × | ServiceOfferRepository.j

```java
package org.example.service;

import ...

public interface ServiceOfferService {  4 usages  1 implementation  kirnes04 *  1 related problem
    ServiceOffer createServiceOffer(CreateServiceOfferDto createServiceOfferDto, Integer receiverId, String r

    List<ServiceOffer> getAllServiceOffersOutgoingById(String name);  1 usage  1 implementation  kirnes04

    List<ServiceOffer> getAllServiceOffersIncomingById(String name);  1 usage  1 implementation  kirnes04

    ServiceOffer getOfferById(String name);  no usages  1 implementation  new *

    ServiceOffer markAsExecuted(Integer offerId, String name);  1 usage  1 implementation  kirnes04
}
```

ymentServiceImpl.java | pom.xml (service-marketplace) | PaymentController.java | ServiceOfferServiceImpl.java ×

```java
public class ServiceOfferServiceImpl implements ServiceOfferService {
    public ServiceOffer createServiceOffer(CreateServiceOfferDto createServiceOfferDto,  1 usage  kirnes04
        return serviceOfferRepository.save(serviceOffer);
    }

    public List<ServiceOffer> getAllServiceOffersOutgoingById(String name) {  1 usage  kirnes04
        return serviceOfferRepository.findAllOutgoingById(userRepository.findByEmail(name).get().getId());
    }

    public List<ServiceOffer> getAllServiceOffersIncomingById(String name) {  1 usage  kirnes04
        return serviceOfferRepository.findAllIncomingById(userRepository.findByEmail(name).get().getId());
    }

    public ServiceOffer getOfferById(Integer id) {  1 usage  new *
        List<ServiceOffer> res = serviceOfferRepository.getOfferById(id);
        if (!res.isEmpty()) {
            return res.get(0);
        }
        return null;
    }

    public ServiceOffer markAsExecuted(Integer offerId, String name) {  1 usage  kirnes04
        var serviceOffer = serviceOfferRepository.findById(offerId).get();
        if (serviceOffer.getIsExecuted()) {
            throw new EntityExistsException("Service offe
        }
        if (!Objects.equals(serviceOffer.getReceiverId().
            throw new AccessDeniedException("You can exec
        }
        serviceOffer.setIsExecuted(true);
        return serviceOfferRepository.save(serviceOffer);
    }
}
```

org.springframework.data.repository.CrudReposi

```
public abstract java.util.Optional<T> findById(
    ID id
)
```

Maven: org.springframework.data:spring-data-com...ons-3

PaymentServiceImpl.java | ServiceOfferServiceImpl.java | PaymentController.java | ServiceOfferService.java | PostService.java | ServiceOfferRepository.java

```java
import me.dynomake.yookassa.Yookassa;
import me.dynomake.yookassa.exception.BadRequestException;
import me.dynomake.yookassa.exception.UnspecifiedShopInformation;
import me.dynomake.yookassa.model.Amount;
import me.dynomake.yookassa.model.Payment;
import org.example.entity.ServiceOffer;
import org.example.service.PaymentService;
import org.example.service.ServiceOfferService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.io.IOException;
import java.io.OutputStream;
import java.math.BigDecimal;
import java.net.Authenticator;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;
import java.nio.charset.StandardCharsets;


@Service  no usages  kirnes04 *
public class PaymentServiceImpl implements PaymentService {

    @Autowired  1 usage
    ServiceOfferService serviceOfferService;

    public String payForOffer(Integer offerId, String name) {  1 usage  kirnes04 *
        try {
            ServiceOffer offer = serviceOfferService.getOfferById(offerId);
            Yookassa yookassa = Yookassa.initialize( shopIdentifier: 356950,  shopToken: "test_BPXHmGzpHRmRz4d1yP7QnEdcmroALAerh44khS6MwPI");
            Payment payment = yookassa.createPayment(new Amount(new BigDecimal(offer.getPrice()),  currency: "RUB"),  s: "google.com", offer.getTitle());
            return payment.confirmation.confirmation_url;
        } catch (UnspecifiedShopInformation | IOException | BadRequestException exception) {
            throw new RuntimeException(exception);
        }
    }
}
```

PaymentController.java | ServiceOfferServiceImpl.java | ServiceOfferRepository.java | ServiceOffer.java

```java
package org.example.repository;

import ...

@Repository  4 usages  kirnes04 *
public interface ServiceOfferRepository extends JpaRepository<ServiceOffer, Integer> {

    @Query(value = "SELECT * FROM public.service_offers WHERE sender_id = ?1", nativeQuery = true)  1 usage
    List<ServiceOffer> findAllOutgoingById(Integer id);

    @Query(value = "SELECT * FROM public.service_offers WHERE receiver_id = ?1", nativeQuery = true)  1 usage
    List<ServiceOffer> findAllIncomingById(Integer id);

    @Query(value = "SELECT * FROM public.service_offers WHERE id = ?1", nativeQuery = true)  no usages  new *
    List<ServiceOffer> getOfferById(Integer id);
}
```

6. Приложения

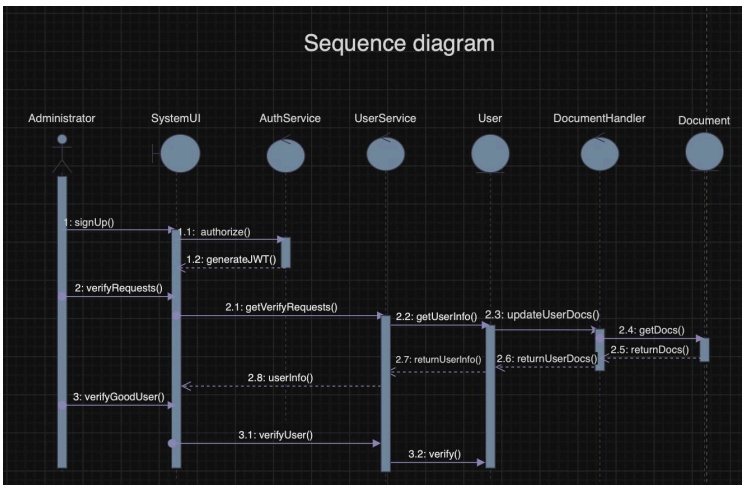
6.1. Словарь терминов

TDD	Test Driven development - методология разработки, в которой сначала пишутся тесты, а только после начинается сама разработка приложения
Maven	Apache Maven — фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах POM (англ. Project Object Model), на языке XML
Junit	JUnit – это фреймворк для модульного (юнит) тестирования на Java. Используется для написания и выполнения автотестов.

6.2. Ссылки на используемые документы

https://drive.google.com/file/d/15e1A50enL9y-OZdkiGqNaw4c3faeUsdp/view?usp=sharing - техническое задание

6.3. Sequence diagram



7. Оценка результативности доработок.

В ходе доработок были добавлены примеры тестов проекта, т.к. предыдущим разработчиком была предложена идея Test Driven Development, они были необходимы. Также реализован ключевой функционал проекта - всё-таки маркетплейс без рабочей оплаты бесполезен. Считаю доработки результативными и ключевыми.