**МИНОБРНАУКИ РОССИИ**

**Федеральное государственное автономное образовательное учреждение высшего образования «Южный федеральный университет»**

**Институт высоких технологий и пьезотехники**

**Кафедра прикладной информатики и инноватики**

**Луценко Константин Сергеевич**

**Мобильное приложение для управления клиентской базой специалистов индустрии красоты**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ**

**РАБОТА БАКАЛАВРА**

**по направлению 09.03.03 – Прикладная информатика**

**Научный руководитель –**

**профессор кафедры информационных и измерительных технологий**

**Яценко Д.В.**

**Ростов-на-Дону – 2025**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ**

**ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ**

**ИНСТИТУТ ВЫСОКИХ ТЕХНОЛОГИЙ И ПЬЕЗОТЕХНИКИ**

**Кафедра прикладной информатики и инноватики**

**З А Д А Н И Е**

**на выпускную квалификационную работу**

**Студент** гр. 4–7 Луценко К.С.

**Тема:** Мобильное приложение для управления клиентской базой специалистов индустрии красоты

**Срок сдачи законченной работы** \_.\_.2025

**Исходные данные:**

Государственные стандарты

1. ГОСТ 7.1–2003 «Библиографическая запись. Библиографическое описание. Общие требования и правила составления»;
2. ГОСТ 7.82–2001 «Библиографическая запись. Библиографическое описание электронных ресурсов»;
3. ГОСТ 19.001–77. Общие положения;
4. ГОСТ 19.781–90. Термины и определения;
5. ГОСТ 19.701–90 (ИСО 5807–85). Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения;
6. ГОСТ 19.103–77. Обозначения программ и программных документов;
7. ГОСТ 19.101–77. Виды программ и программных документов;
8. ГОСТ 19.106–78. Требования к программным документам, выполненным печатным способом;
9. ГОСТ 19.104–78. Основные надписи;
10. ГОСТ 19.301–79. Программа и методика испытаний. Требования к содержанию и оформлению;
11. ГОСТ 19.402–78. Описание программы;
12. ГОСТ 19.401–78. Текст программы. Требования к содержанию и оформлению;
13. ГОСТ 19.504–79. Руководство программиста. Требования к содержанию и оформлению;
14. ГОСТ 19.505–79. Руководство оператора. Требования к содержанию и оформлению;
15. ГОСТ 19.202–78. Спецификация. Требования к содержанию и оформлению.
16. Основное требование – разработать мобильное приложение, выполняющее основные требуемые функции.

***5. Перечень вопросов, подлежащих разработке:***

1. *Анализ предметной области.*
2. *Обоснование технических решений.*
3. *Проектирование элементов системы.*
4. *Реализация компонентов системы.*
5. *Разработка методики тестирования.*

***6. Перечень графического материала:***

*Подготовка графических материалов для презентации работы*

***7. Консультанты по работе:***

***8. Дата выдачи задания: \_\_.\_\_.2025***

***9. Руководитель Яценко Д.В.***

*Подпись ФИО*

***Задание принято к исполнению***

*Дата Подпись студента*

АННОТАЦИЯ

СОДЕРЖАНИЕ

[ГЛАВА 1. АНАЛИЗ, ОБЗОР РЕШЕНИЙ, ПОСТАНОВКА ЗАДАЧИ 9](#_Toc192960688)

[1.1 Первичная постановка задачи 9](#_Toc192960689)

[1.2 Декомпозиция задачи 9](#_Toc192960690)

[1.3. Обзор существующих аналогов 10](#_Toc192960691)

[1.4 Требования к системе 14](#_Toc192960692)

[1.5 Обзор решений 14](#_Toc192960693)

[1.5.1 Выбор средств для мобильной разработки 14](#_Toc192960694)

[1.5.2 Выбор решений серверной части 18](#_Toc192960695)

[1.5.3 Выбор сред разработки 21](#_Toc192960696)

[1.5.4 Выбор системы управления базами данных 22](#_Toc192960697)

[1.6. Выводы по главе 23](#_Toc192960698)

[ГЛАВА 2. ПРОЕКТИРОВАНИЕ СИСТЕМЫ 25](#_Toc192960699)

[2.1 UML моделирование 25](#_Toc192960700)

[2.1.1. Диаграмма прецедентов 26](#_Toc192960701)

[2.1.2 Диаграмма классов 29](#_Toc192960702)

[2.1.3 Диаграмма компонентов 31](#_Toc192960703)

[2.1.4 Диаграмма развертывания 34](#_Toc192960704)

[2.2 Выводы по главе 35](#_Toc192960705)

[ГЛАВА 3. РЕАЛИЗАЦИЯ 37](#_Toc192960706)

[3.1 Создание клиентской части 37](#_Toc192960707)

[3.1.1 Реализация экранов регистрации и авторизации 37](#_Toc192960708)

[3.1.2 Основные вкладки – главная, записи, чаты, личный кабинет 39](#_Toc192960709)

[3.1.2.1 Домашняя страница и ее логика 41](#_Toc192960710)

[3.1.2.2 Экран записей и его логика 48](#_Toc192960711)

[3.1.2.3 Экран чатов и его логика 51](#_Toc192960712)

[3.1.2.4 Экран профиля и его логика 53](#_Toc192960713)

[3.1.2.5 Уведомления 54](#_Toc192960714)

[3.2.1 Презентационный слой 57](#_Toc192960715)

[3.2.1.1 Контроллер аутентификации 57](#_Toc192960716)

[3.2.1.2 Контроллер пользователей 58](#_Toc192960717)

[3.2.1.3 Контроллер услуг 59](#_Toc192960718)

[3.2.1.4 Контроллер записей 60](#_Toc192960719)

[3.2.1.5 Контроллер уведомлений 60](#_Toc192960720)

[3.2.2 Бизнес-слой 61](#_Toc192960721)

[3.2.2.1 Сервис услуг 62](#_Toc192960722)

[3.2.2.2 Сервис записей 64](#_Toc192960723)

[3.2.2.3 Сервис сообщений 64](#_Toc192960724)

[3.2.3 Слой взаимодействия с базами данных 66](#_Toc192960725)

[3.2.4 Логирование 69](#_Toc192960726)

[3.2.5 Документирование 70](#_Toc192960727)

[3.3 Вывод по главе 71](#_Toc192960728)

[ГЛАВА 4. ТЕСТИРОВАНИЕ 72](#_Toc192960729)

[4.1 Тестирование серверной части 72](#_Toc192960730)

[4.2 Разработка unit-тестов 72](#_Toc192960731)

[4.2.1 Тестирование контроллеров 73](#_Toc192960732)

[4.2.2 Тестирование сервисов 75](#_Toc192960733)

[4.3 Ручное тестирование 76](#_Toc192960734)

[4.4 Выводы по главе 77](#_Toc192960735)

[ГЛАВА 5. БЕЗОПАСНОСТЬ ЖИЗНЕДЕЯТЕЛЬНОСТИ 78](#_Toc192960736)

[5.1 Понятие безопасности жизнедеятельности 78](#_Toc192960737)

[5.2 Основные правила безопасности жизнедеятельности при обращении с компьютером 78](#_Toc192960738)

[5.3 Выводы по главе 79](#_Toc192960739)

[ГЛАВА 6. ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ 80](#_Toc192960740)

[6.1 Расчет затраченных средств на разработку 80](#_Toc192960741)

[6.2 Монетизация приложения 80](#_Toc192960742)

[6.3 Выводы по главе 81](#_Toc192960743)

[ПРИЛОЖЕНИЕ А 87](#_Toc192960744)

ВВЕДЕНИЕ

На сегодняшний день информационные технологии плотно проникли практически во все сферы человеческой деятельности, что привело к постоянно растущей потребности в разработке все более новых, уникальных и сложных информационных систем.

Если рассматривать процедуру работы предпринимателей, осуществляющих взаимодействие с клиентской базой, то можно заметить, что раньше достаточное количество времени затрачивалось на ведение различного рода документации, одной из которых, например, является журнал записи. Ранее требовалось найти способ установить связь с человеком, при условии того, что не все используют соцсети и мессенджеры, а затем проследить за правильностью записи и ее очередностью. Сегодня же, процесс стал значительно проще, так как появились средства автоматизации, которые позволяют снять огромную «ручную» работу с мастера и передать ее вычислительной системе [1].

С переносом подобных рутинных процессов в информационные системы решается множество проблем ведения бумажной записи: человеческие ошибки, результатом которых является наложение записей двух разных клиентов друг на друга или подобные; трудозатраты на осуществление и выбор удобного варианта записи; необходимость напоминаний многим из клиентов о том, когда и на какую процедуру они записаны.

Актуальность темы дипломной работы обусловлена активным ростом применения программных средств взаимодействия с клиентской базой. Такое мобильное приложение стало бы значимой альтернативой платным сервисам, которые трудны и недоступны самозанятым и клиентам, позволив первым упростить работу, а вторым ускорить и облегчить нахождение нужного мастера и запись к нему.

ГЛАВА 1. АНАЛИЗ, ОБЗОР РЕШЕНИЙ, ПОСТАНОВКА ЗАДАЧИ

1.1 Первичная постановка задачи

Задачей выпускной квалификационной работы является разработка мобильного приложения для работы с клиентской базой мастеров индустрии красоты. Итогом работы должно стать приложение под IOS/Android, призванное облегчить взаимодействие между мастером и клиентом, а также снять немалую нагрузку с мастера и автоматизировать рутинные задачи, позволив предпринимателю увеличить охват клиентов и оказать как можно больше услуг.

Приложение должно позволять:

* Клиентам: найти требуемую услугу, подходящего мастера и записаться к нему, предварительно оценив местоположение и отзывы об услуге. Получить уведомление об услуге заранее. Также должна быть возможность связаться с мастером непосредственно внутри приложения, а также инструменты управления записью.
* Мастерам: контролировать записывающихся клиентов, взаимодействовать с ними. Собирать статистику и отчетность об оказанных услугах. Должны существовать возможности продвижения для повышения заинтересованности клиентов.

Основная идея – создать простое, понятное и доступное для каждого кроссплатформенное приложение под любой смартфон. Необходимо обеспечить простоту интерфейса и достаточное число инструментов для удобства как клиентов, так и мастеров.

1.2 Декомпозиция задачи

Выделенную ранее задачу можно разделить на следующие этапы:

1. Поиск аналогов, исследование их достоинств и недостатков, взятие на заметку некоторых идей по реализации преимуществ и нейтрализации неудобств.
2. Анализ основных требований к приложению.
3. Проектирование мобильного приложения, разработка требований к нему.
4. Проектирование серверной архитектуры, разработка требований к API.
5. Разработка программного кода двух принципиальных частей системы.
6. Тестирование мобильной и серверной части, а также их взаимодействия. Устранение возникших проблем.
7. Демонстрация результатов, проверка на выполнение всех требований.
8. Развертывание серверной части, тестирование на целевой аудитории.

1.3. Обзор существующих аналогов

Первоочередным действием при начале разработки информационной системы является обзор и анализ конкурентов. Проведение данного этапа позволяет учесть все недостатки похожих приложений, а наиболее удачные реализации функций улучшить и внедрить и в свой продукт. Таким образом, достигается реализация максимально удобного и ориентированного на пользователя приложения, обладающего уникальными инструментами для работы.

Для оценки конкурентов будут рассмотрены следующие сервисы: YClients, SimplyBook, DIKIDI, Masters, Мой Профи. Исследованы все преимущества и недостатки для построения конкурентоспособного приложения для своей целевой аудитории.

YClients – российская платформа онлайн-записи для различного вида бизнеса [2].

Данная система позволяет не только автоматизировать рутинные процессы, но и вести бизнес в целом – рассчитывать зарплату, подключать онлайн-кассу и другое. Платформа завоевывает обширную долю рынка Российской Федерации и активно конкурирует с DIKIDI, который будет рассмотрен ниже.

Преимущества:

* Платформа поможет решить подавляющее множество проблем при ведении бизнеса. Осуществлять складской учет, создавать программы лояльности, организовывать интеграции с другими сервисами.
* Заключила партнерство с крупными соцсетями, банками и другими площадками, что позволяет увеличить вовлеченность клиентов к услугам и бизнесу [3].
* Подходит практически любой сфере бизнеса и предлагает инструменты для удобного менеджмента.

Недостатки:

* Достаточно дорогостоящая для начинающих специалистов.
* Высокий порог входа: из-за расширения возможностей интерфейс стал нагруженным и не всегда интуитивно понятен. Для решения этой проблемы созданы курсы, однако они очень дороги.
* Разработчики не прислушиваются к массовым требованиям о внедрении необходимых функций, допускают множество ошибок в реализации.

SimplyBook – приложение онлайн-записи от SimplyBook.me с системой бронирования для сайтов и онлайн-календарем [4].

Преимущества:

* Достаточно прост для использования со стороны клиента.
* Даёт возможность создания собственного шаблонного сайта для записи на базе домена SimplyBook.
* Интеграция с различными API, возможность уведомлений клиентов по SMS.

Недостатки:

* Зарубежный, поэтому в любое время может стать недоступен для использования.
* Изначально создан для англоязычной аудитории и плохо переведен на русский язык. Незнание английского языка может вызывать проблемы при использовании и настройке приложения.
* Множество проблем с работоспособностью приложения, виджетов. Неудобство интерфейса.
* Отсутствие качественной и быстрой технической поддержки.
* Достаточно дорог для малого бизнеса, оплату из России осуществить сложнее из-за политики зарубежных платежных систем.

DIKIDI – бесплатный международный сервис онлайн-записи, автоматизации, управления и продвижения бизнеса в сфере услуг. [5]

Преимущества:

* Прост в освоении и настройке, не требует долгого обучения.
* Интеграции с картами, 1С, SMS-уведомлениями и другими сервисами.
* Есть бесплатный тариф, предоставляющий базовый функционал.

Недостатки:

* Отсутствие качественной технической поддержки.
* Приложение часто работает с ошибками и сбоями.
* Некоторые функции доступны только в web-версии, а реализация в мобильной версии затянута.

Rient – бесплатный сервис онлайн-записи + CRM для сферы услуг [6].

Преимущества:

* Бесплатный базовый функционал.
* Удобная подробная статистика.
* Инструменты CRM [7], обеспечивающие управление не только взаимоотношениями с клиентами, но и бизнесом.

Недостатки:

* Отсутствие мобильного приложения.

Мой профи – мобильное приложение для планирования сеансов и продвижения мастеров красоты [8].

Преимущества:

* Ориентировано на частных мастеров, как следствие предлагает конкурентно низкую стоимость подписки.
* Простое в использовании, не требует обширной предварительной настройки. Для любого пользователя, имеющего в наличии мобильный телефон не составит труда записаться, либо организовать онлайн-запись на свои услуги.
* Кроссплатформенность: разработчики создали и сайт, и мобильное приложение, что позволяет записаться с любого устройства, будь то персональный компьютер, планшет или смартфон.

Недостатки:

* Устаревший интерфейс, не привлекающий к элементам управления.
* Множество отзывов об ухудшении работоспособности приложения при его обновлении, ошибки. Разработчики не всегда вовремя исправляют проблемы, что приводит к снижению числа пользователей.
* Проблемы с получением уведомлений, периодически возникают сбои и напоминания как для мастеров, так и для клиентов, перестают работать.

1.4 Требования к системе

Итогом разработки должна стать информационная система, соответствующая следующим требованиям:

1. Система должна быть доступна на мобильных устройствах с любой операционной системой;
2. Приложение должно обладать разным функционалом для мастеров и клиентов;
3. Позволяет клиенту подобрать необходимую услугу, записаться на неё и, при необходимости, отменить;
4. Позволяет мастеру осуществлять контроль за записями клиентов, регулировать их;
5. Реализовывает возможность связи посредством встроенного чата;
6. Собирает удобную статистику для мастера;

1.5 Обзор решений

Были исследованы решения для мобильной и серверной части, чтобы достичь максимального удобства и при этом достаточной производительности приложения, обеспечив при этом хороший пользовательский опыт. Далее представлен обзор решений для каждой части системы.

1.5.1 Выбор средств для мобильной разработки

Можно выделить две основные категории языков программирования для разработки приложений на смартфоны – нативные и кроссплатформенные. Нативные языки программирования – платформы, созданные специально под определенную операционную систему. Для IOS это Swift, для Android – Java, Kotlin. В то время как нативные языки позволяют более тонко работать с ресурсами той или иной системы, позволяя достичь максимальной производительности и практически полного контроля, кроссплатформенные языки позволяют не тратить время на разработку двух приложений для каждой операционной системы, а направить весь вектор разработки на особенности самого функционала приложения.

В нашем случае, основная целевая аудитория – девушки, которые активно пользуются смартфонами. Согласно исследованию сервиса Яндекс.Радар по состоянию на 9 ноября 2024 года доля устройств на основе операционной системы Android чуть менее 80%, в то время как устройства под управлением IOS составили более 20% [9]. На графике можно увидеть, как менялись предпочтения в операционных системах смартфонов за 10 лет (Рисунок 1).

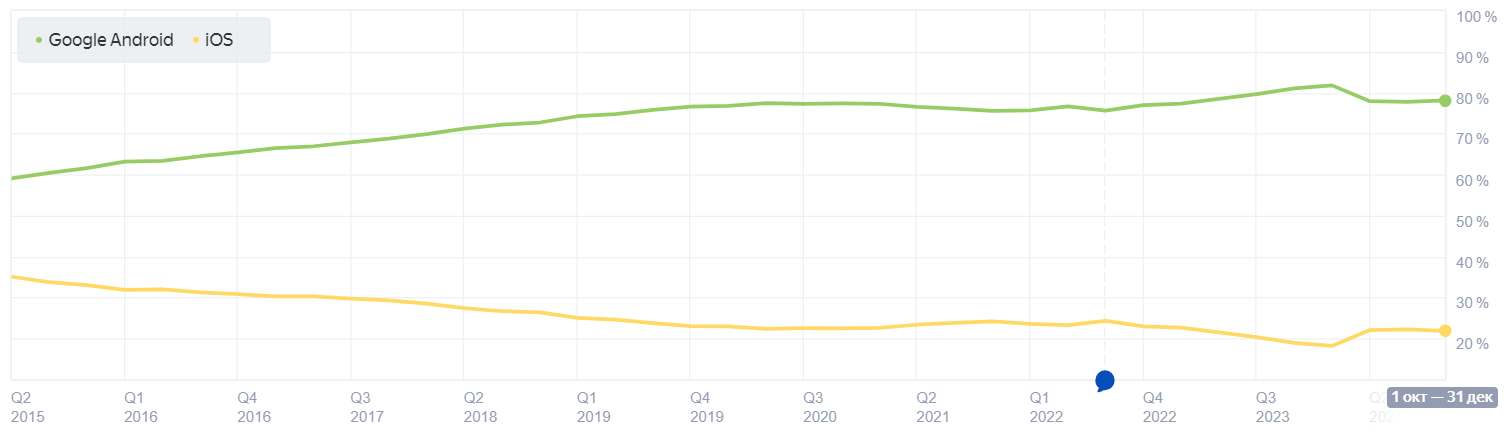


Рисунок 1 – Использование операционных систем на смартфонах в России в 2015-2024 годах.

Поэтому основное внимание нужно уделить этим двум операционным системам, что влечет за собой важность использования именно кроссплатформенных инструментов для экономии трудозатрат при разработке.

Рассмотрим самые популярные кроссплатформенные языки программирования React Native и Flutter.

React Native – платформа, созданная Facebook для создания мобильных приложения, используя в основе JavaScript и React, позволяющая создавать одну кодовую базу для разных операционных систем.

Был выпущен в 2015 году с открытым исходным кодом и в первые пару лет завоевал огромное доверие разработчиков. На нем были написаны такие мобильные приложения, как Skype, Discord, Facebook, Airbnb и другие.

Можно выделить основные причины популярности данного инструмента:

1. Экономия трудозатрат. Компания тратит средства и время на одну группу разработчиков, которые делают одно приложение для двух операционных систем, вместо двойных затрат во времени и средствах под каждую ОС отдельно;
2. В основе лежит React, который в свою очередь построен с использованием JavaScript, что обеспечивает низкий уровень входа в разработку, особенно, если есть предыдущий опыт в двух вышеупомянутых средствах;
3. Фреймворк позволил разработчикам веб-платформ без труда переквалифицироваться дополнительно в разработчиков и мобильных решений.

Flutter – инструментарий для создания скомпилированных приложений на основе виджетов, что даёт в результате визуально коренные элементы для каждой из операционных систем. Представлен компанией Google в конце 2018 года.

Поначалу фреймворк был воспринят с осторожностью, ведь на рынке был уже достаточно популярный и понятный React Native. Однако с течением лет ситуация изменилась и сейчас можно утверждать, что средства – прямые конкуренты друг другу. Можно сравнивать данные инструменты по следующим критериям:

1. Производительность. Разработчики Flutter предусмотрели его компиляцию в собственные библиотеки ARM или x86, что заметно влияет на производительность. React Native не компилируется в собственный код, и у него все еще есть слой JavaScript, что делает его менее производительным, чем Flutter. Однако Facebook знают о проблеме и занимаются ее решением, как результат, разработка движка Hermes, значительно повышающего производительность платформы на мобильных системах. Flutter в любом случае немного будет выигрывать в производительности, за счет того, что в качестве основы использует движок на C++, а React Native никак в ближайшее время не сможет избавиться от bridge архитектуры, устанавливая соединение с JavaScript посредством JSON. Однако разница в производительности несущественна, что позволяет выбирать любой из понравившихся инструментов.
2. Визуальная составляющая. Flutter использует виджеты, которые гарантированно будут выглядеть так же, как и элементы, построенные на нативных средствах разработки. React Native же работает напрямую с нативными элементами, что помимо правильного внешнего вида даёт возможность обновления в реальном времени.
3. IDE. Для React Native представлен более широкий спектр средств: практически все среды разработки, доступные для React поддерживают и React Native. В случае с Flutter не все так гладко, пока его поддерживают VS Code, Intellij и Android Studio.
4. API и пользовательский интерфейс. Для Flutter представлено больше встроенных средств, что даёт большую надежность и возможность выбора того, что необходимо. В случае с React Native встроенных библиотек не так много, лишь основные необходимые, однако, написано множество сторонних библиотек и даже фреймворков поверх React Native, которые реализуют тот же функционал, что и библиотеки Flutter, хоть местами и не так стабильны [10].

На основе вышеперечисленных рассуждений можно сделать вывод, что на данном этапе инструменты идут наравне, каждый имея свои плюсы и минусы. Поэтому выбор зависит только от предпочтений в языке разработки и ключевых особенностях разрабатываемого приложения. В нашем случае приложение визуально не нагруженное, поэтому React Native подойдет в качестве простого и понятного средства для разработки. Для облегчения сборки и запуска будем использовать фреймворк Expo, предоставляющий возможность видеть изменения в коде приложения в реальном времени благодаря развертыванию сервера и открытия его в приложении Expo Go. Так же Expo предоставляет собственные библиотеки и заготовки для разработки, что ускоряет создание приложения.

1.5.2 Выбор решений серверной части

Для серверной части буду использовать Java Spring Boot. Он написан на Java - одном из самых важных и широко используемых языков программирования в мире. В отличие от ряда других языков программирования, влияние которых с течением времени ослабевало, влияние Java с годами становилось только сильнее. Разработчики языка выделяют следующие особенности данного языка, которые и делают его настолько популярным:

* Простота – язык Java был спроектирован так, чтобы быть легким в изучении и эффективным в использовании профессиональным программистом. Если у вас есть определенный опыт программирования, то освоить Java не составит особого труда;
* надежность – Java является строго типизированным языком: ваш код проверяется на этапе компиляции. Тем не менее, код также проверяется и во время выполнения. Многие трудно обнаруживаемые ошибки, которые часто приводят к возникновению сложных для воспроизведения ситуаций во время выполнения, в Java попросту невозможны. Предсказуемость поведения написанного вами кода в несходных условиях - ключевая особенность Java;
* объектная ориентация – обильно заимствуя из многих продуктивных объектно-ориентированных сред, существующих на протяжении последних нескольких десятилетий, в Java удалось найти баланс между парадигмой "все является объектом" сторонников чистоты стиля и более прагматичной моделью "не путайтесь под ногами: объектная модель в Java проста и легко расширяема, в то время как элементарные типы, такие как целочисленные, были сохранены высоко производительными сущностями, которые не являются объектами;
* многопоточность – в Java поддерживается многопоточное программирование, которое дает возможность писать программы, выполняющие много действий одновременно. Исполняющая среда Java поддерживает элегантное, но вместе с тем сложное решение для синхронизации множества процессов, которое делает возможным построение бесперебойно работающих интерактивных систем;
* нейтральность к архитектуре – разработчики преследовали идеологию работоспособности написанного кода везде и всегда;
* интерпретируемость и высокая производительность – язык Java делает возможным создание межплатформенных программ за счет их компиляции в промежуточное представление, называемое байт-кодом Java. Байт-код может выполняться на любой системе, где внедрена машина JVM;
* распределенность – язык поддерживает TCP/IP и удаленный вызов методов;
* динамичность – программы на Java содержат существенный объем информации о типах времени выполнения, используемой для проверки и разрешения доступа к объектам во время выполнения, что делает возможным безопасное и надлежащее динамическое связывание кода. Это критически важно для надежности среды Java, в которой небольшие фрагменты байт-кода могут динамически обновляться в функционирующей системе [11].

Spring Boot – проект, упрощающий настройку и разработку приложений Spring. Построен на основе Spring Framework. Призван упростить разработку приложений с минимальной конфигурацией и ускорить процесс создания веб-приложений.

Преимущества Spring Boot:

1. На основе зависимостей, добавленных в проект, делает предположения о желаемом и производит настройку на основе этого, убирая ненужные части стандартной конфигурации Spring;
2. Имеет встроенный сервер Tomcat, Jetty или Undertow, позволяющий разрабатывать автономные приложения, собирающиеся в JAR-файл для легкого развертывания на любой машине с JVM.

Hibernate ORM (объектно-реляционное отображение) – фреймворк, упрощающий взаимодействие с базами данных для Java-приложений. Сопоставляет объектно-ориентированную модель предметной области с реляционной базой данных и дает возможность обращаться с данными, как с сущностями, которые разработчик сам реализует в виде желаемых таблиц в базе данных.

Преимущества Hibernate:

1. Заменяет прямой доступ к базе данных на функции обработки объектов высокого уровня, что, в свою очередь, облегчает понимание и работу разработчкикам;
2. Сопоставляет таблицы в базах данных с классами в Java и упрощает операции с базой;
3. Автоматически обеспечивает сохраняемость, разработчику больше не нужно заботиться о базовых операциях работы с базой данных, затрачивая время на написание явного кода взаимодействия с СУБД;
4. Предоставляет собственный мощный язык запросов HQL, в который заложены идеологии наследования, полиморфизма и ассоциации [12].

Таким образом, связка Java Spring Boot и Hibernate с необходимыми дополнительными библиотеками позволяют разрабатывать мощные веб-приложения, легко взаимодействующие с базой данных, а также реализующие все преимущества языка программирования Java.

1.5.3 Выбор сред разработки

Для разработки мобильной части выбрана Visual Studio Code.

Visual Studio Code, или VSCode, — текстовый редактор/редактор исходного кода, разработанный Microsoft с использованием Electron Framework для Windows, Linux и MacOS. Это один из самых популярных редакторов исходного кода на рынке. Вот некоторые особенности, которые отличают его от обычного текстового редактора:

* IntelliSense. VSCode выходит за рамки обычной подсветки синтаксиса и предлагает IntelliSense — интеллектуальные дополнения, адаптированные к коду и его тонкостям, которые дополнительно можно отрегулировать под свой вкус с помощью плагинов;
* Запуск и отладка. VSCode позволяет запускать ваш код (с отладкой или без неё с помощью веб-редактора) в самом приложении, а также обращаться к терминалу Windows напрямую из приложения для запуска некоторых средств, если это необходимо;
* Встроенный Git. В VSCode упрощена работа с Git, что позволяет контролировать коммиты, подготавливать файлы и выполнять другие действия с Git прямо из редактора. Реализовано это в виде удобного раздела, делающего работу прозрачной и быстрой, достаточно авторизоваться в Git и готово;
* Расширение: написано множество плагинов, расширяющих базовый функционал приложения и внедряющих множество полезных инструментов, что позволяет создать уникальную среду разработки под интересы каждого разработчика [13].

Для разработки серверной части выбрана IntelliJ IDEA.

IDEA разработана частной компанией JetBrains. Это интегрированная среда разработки (IDE), написанная на Java для разработки программного обеспечения и проектов. В ней существует бесплатная Community версия, дающая базовый функционал.

Возможности IntelliJ IDEA:

* Интеллектуальные подсказки к коду. В реальном времени во время написания IDE предугадывает, что вы хотите написать, давая подробную информацию о каждом классе или методе. Достаточно выбрать, что необходимо и не затрачивать время на самостоятельное написание. Так же данная функция помогает вспомнить забытые методы или просмотреть неизвестные с целью воспользоваться ими;
* Поиск повторяющегося кода. Все разработчики борются за повторное использование кода. IDE ищет повторяющийся код и помогает его заменить;
* Быстрые исправления. Если разработчик допускает ошибку в написании кода, IDE это отслеживает и предлагает варианты решения в реальном времени;
* Поддержка Git. Аналогично VSCode позволяет легко и быстро настроить взаимодействие с Git;
* Поддержка плагинов. Позволяет устанавливать плагины из магазина, что позволяет делать IDE уникальной под ваши вкусы и предпочтения [14].

1.5.4 Выбор системы управления базами данных

В качестве СУБД выбрана PostgreSQL – мощная объектно-реляционная система баз данных с открытым исходным кодом, которая использует и расширяет язык SQL в сочетании со многими функциями, позволяющими безопасно хранить и масштабировать работу с данными. История создания PostgreSQL началась в 1986 году в рамках проекта POSTGRES Калифорнийского университета в Беркли и насчитывает более 35 лет активной разработки на базовой платформе.

PostgreSQL заслужил прочную репутацию благодаря своей проверенной архитектуре, надежности, целостности данных, надежному набору функций, расширяемости и приверженности сообщества разработчиков программного обеспечения с открытым исходным кодом постоянному предоставлению производительных и инновационных решений. PostgreSQL работает на всех основных операционных системах, с 2001 года поддерживает ACID и имеет мощные дополнения, такие как популярный расширитель геопространственных баз данных PostGIS.

PostgreSQL обладает множеством функций, призванных помочь разработчикам создавать приложения, администраторам – защищать целостность данных и создавать отказоустойчивые среды, а также управлять данными независимо от того, насколько они масштабны [15].

1.6. Выводы по главе

В данной главе рассмотрена идеология разрабатываемого приложения, поставлена цель разработки и произведена декомпозиция задачи, что позволяет построить последовательный и четкий план разработки. Сформированы функциональные требования к системе.

Рассмотрены аналоги, что позволило понять, каким функционалом должно обладать приложение и какие недостатки конкурентов устранять, чтобы обладать популярностью у пользователей и выполнять необходимые функции.

Выбраны языки программирования, фреймворки, средства управления базами данных и среды разработки для мобильной и серверной частей приложения. Рассмотрены React Native, Expo, Java, Spring Boot, Hibernate, преимущества каждого из них и обоснован их выбор, чтобы обеспечить необходимые характеристики приложения, такие как визуальная составляющая, производительность, безопасность и другие. В случае с средами разработки рассмотрен функционал и возможности, влияющие на скорость и производительность разработки. Выбраны VSCode и IntelliJ IDEA, предоставляющие весь необходимый функционал для комфортной работы.

Изучена техника безопасности при разработке.

ГЛАВА 2. ПРОЕКТИРОВАНИЕ СИСТЕМЫ

2.1 UML моделирование

Проектирование – важнейший начальный этап разработки качественного приложения. Данная процедура позволяет четко определить архитектуру приложения, его базовые составляющие, которые потом предстоит реализовать. Особенно важно продумать максимально возможное количество деталей на раннем этапе, потому что чем позже будет выявлена ошибка или недоработка при проектировании, тем дороже и длительнее её исправление.

Существуют различные инструменты, позволяющие проектировать системы, однако наиболее популярным из них является унифицированный язык моделирования – UML (Unified Modeling Language) – язык графического описания, позволяющий моделировать информационные системы, бизнес-процессы, а также отображать организационные структуры [16].

Главная причина создания языка моделирования UML –сократить разрыв в понимании различными разработчиками устройства тех или иных систем, что, с свою очередь, способствует ускорению и упрощению коммуникации между сотрудниками и значительно сокращает время создания продукта.

UML предоставляет набор диаграмм и их нотаций, которые условно можно разделить на два вида:

1. Структурные диаграммы, иллюстрирующие сложную структуру систем и их компоновку
2. Диаграммы поведения, иллюстрирующие взаимодействие с системой, процесс её работы

Преимущества языка UML:

* Коммуникация. Облегчает коммуникацию между разработчиками и другими работниками, принимающими участие в проекте.
* Визуализация. Предоставляет комплекс диаграмм, упрощающих понимание устройства, работы и взаимодействия системы.
* Стандартизация. Гарантирует использование различными разработчиками общего и доступного инструмента.

Недостатки:

* Сложность и избыточность. Для ознакомления с языком может потребоваться длительное время, а многие компоненты, предоставленные инструментом, используются очень редко.
* Чрезмерность или недостаточность моделирования. Существует риск создания чрезмерно нагруженной и трудной для понимания диаграммы, а также недостаточно информативной для понимания, что может вызывать недопонимание.

2.1.1. Диаграмма прецедентов

Диаграмма прецедентов, или диаграмма вариантов использования (англ. use-case diagram) – диаграмма, отображающая какой функционал приложения доступен определенным группам пользователей.

Основные группы пользователей отображаются символом «человечка» и в терминологии UML называются актерами (англ. actor) – это сущности, которые пользуются системой.

Актеры пользуются функционалом системы, который на диаграмме принято обозначать эллипсом с названием функции внутри и называть вариантом использования (англ. use-case) – это совокупность действий, которые приложение предоставляет актерам.

Для связывания элементов используются соединительные линии, отображение которых имеет значение, в терминологии их называют отношениями. Выделяют 4 вида отношений в диаграмме прецедентов:

* Отношение ассоциации (англ. association relationship) – предназначено для соединения актеров и вариантов использования. Обозначается сплошной линией;
* Отношение обобщения (англ. generalization relationship) – реализует повторное использование кода из принципов объектно-ориентированного программирования и позволяет обобщать некоторые частные случаи актеров или вариантов использования до более общих. Например, в нашем приложении можно обобщить актеров «Мастер» и «Клиент» до сущности «Пользователь». Обозначается сплошной линией с полой треугольной стрелкой;
* Отношение включения (англ. include relationship) – позволяет более точно указать варианты использования, обязательно входящие в какой-то более общий, отобразить составные действия. Обозначается пунктирной линией с V-образной стрелкой на конце и надписью «include» над ней;
* Отношение расширения (англ. extend relationship) – похоже на отношение включения, однако входящий вариант не обязателен и может как быть использован, так и нет. Обозначается пунктирной линией аналогично отношению включения, отличается надписью «extend» над стрелкой [17].

Руководствуясь концепцией приложения и требованиями к нему, изложенными в первой главе, была разработана диаграмма вариантов использования (Рисунок 2). Актерами являются пользователи, разделенные по ролям («Мастер» и «Клиент»), а также «Система».

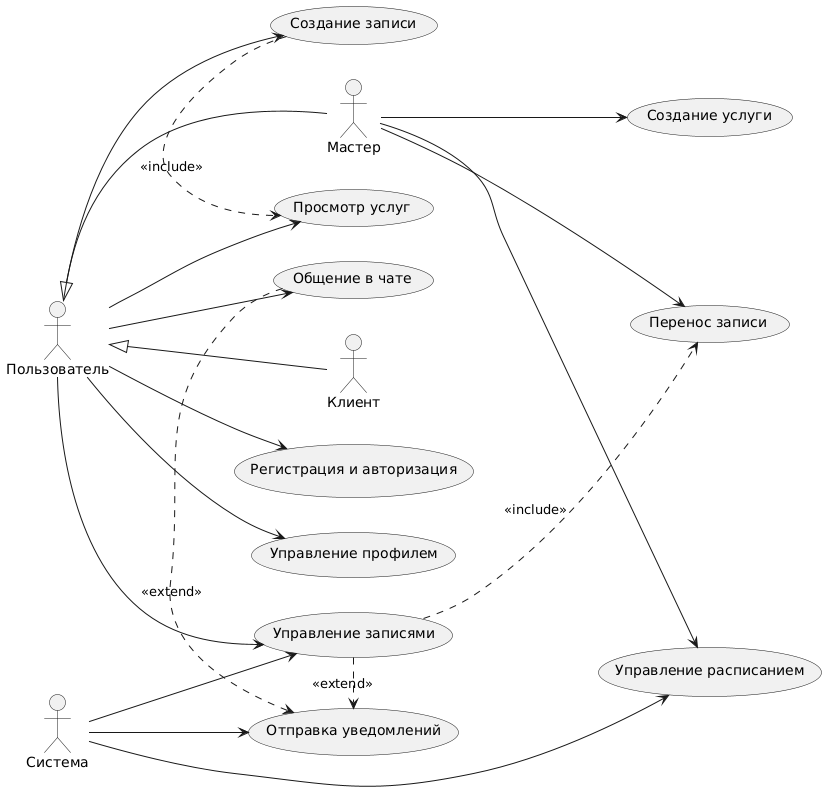


Рисунок 2 - Диаграмма вариантов использования

Из диаграммы видим, что доступны следующие варианты использования приложения:

* Регистрация и авторизация. Позволяет пользователям регистрироваться и авторизовываться в системе, получать доступ к ней.
* Просмотр услуг. Позволяет пользователям просматривать доступные услуги. Включается в вариант использования «Создание записи».
* Создание записи. Позволяет пользователям записываться на услуги. Включает вариант использования «Просмотр услуг».
* Управление записями. Позволяет пользователям переносить и отменять записи под контролем системы, через подтверждение. Включает вариант использования «Перенос записи». Расширяется до варианта использования «Отправка уведомлений».
* Перенос записи. Позволяет пользователями переносить записи. Включается в вариант использования «Управление записями».
* Общение в чате. Позволяет пользователям взаимодействовать в чате. Расширяется до варианта использования «Отправка уведомлений».
* Управление профилем. Позволяет пользователям управлять данными профиля.
* Создание услуги. Позволяет мастерам создавать услуги, доступные клиентам для записи.
* Управление расписанием. Позволяет мастерам управлять расписанием под контролем системы.
* Отправка уведомлений. Позволяет системе отправлять уведомления пользователям. Расширяется от вариантов использования «Управление записями» и «Общение в чате».

2.1.2 Диаграмма классов

Диаграмма классов – визуальный инструмент, отображающий структуру, классы, методы системы и взаимосвязи между ними. Позволяет участникам проекта понять устройство и организацию системы [18].

На диаграмму наносят два типа элементов: классы и отношения между ними. Класс имеет вид прямоугольника с названием в верхнем блоке, атрибутами в среднем блоке и методами в нижнем, а также их видимости (public, protected, private). Для каждого атрибута указывается тип данных, в котором он хранится, для каждого метода указываются параметры, их типы данных и возвращаемое значение. Отношения между классами можно выделить следующие:

* Отношение ассоциации – отображает связь между классами. Изображается в виде сплошной линии с заполненным треугольником на конце. Под связью пишут кратность связи (один к одному, один ко многим и так далее);
* Отношение зависимости – отображает зависимость одного класса от другого (зависимый класс является атрибутом другого). Изображается в виде пунктирной линии с заполненным треугольником на конце;
* Отношение обобщения (наследования) – отображает наследование производного класса от родительского. Изображается в виде сплошной линии с незаполненным треугольником на конце;
* Отношение агрегации – отображает, что один из классов включает в себя в качестве составной части другой класс. При этом части, из которых состоит класс, могут существовать абсолютно обособленно. Изображается в виде сплошной линии с незаполненным ромбом на конце. У линии также может указываться кратность;
* Отношение композиции – похоже на отношение агрегации, так же отображает, что класс состоит из частей, однако эти классы-части не могут существовать обособленно. Изображается сплошной линией с заполненным ромбом на конце.

На основе диаграммы прецедентов, описанной в прошлом пункте, были спроектированы основные сущности, необходимые для работы системы (Рисунок 3).

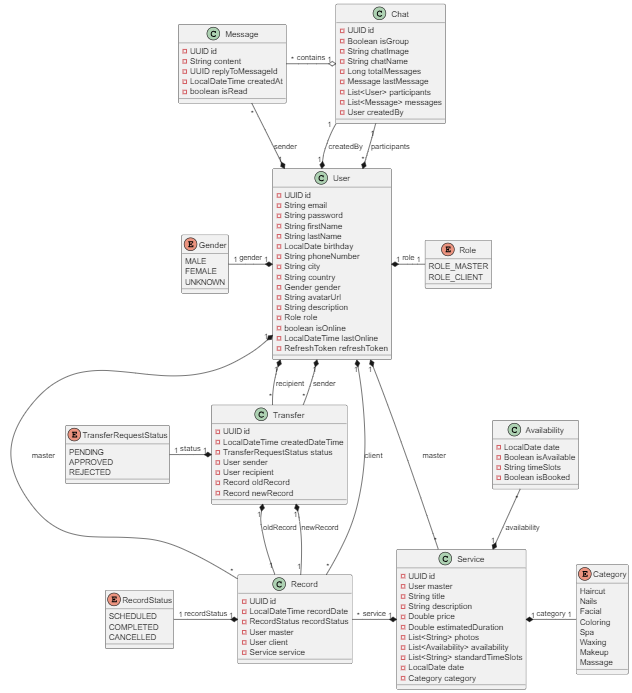


Рисунок 3 - Диаграмма классов

2.1.3 Диаграмма компонентов

Диаграмма компонентов – структурная диаграмма UML, предназначенная для визуального отображения архитектуры системы и взаимосвязей между её компонентами [19].

Основные элементы нотации диаграммы компонентов:

* Компонент – часть системы, представляющий определенный функционал. Это могут быть модули, классы, сервисы, устройства и так далее. Обозначается в виде прямоугольника с именем компонента.
* Интерфейс – возможности взаимодействия одного компонента с другими. Бывают предоставляемыми и требуемыми – это соответственно, тот функционал, который может быть предоставлен компонентом, или наоборот, требуемые функции. Предоставляемый интерфейс отображается сплошной линией с окружностью на конце, требуемый – сплошной линией с полуокружностью на конце. Соединяются между собой между двумя взаимодействующими компонентами.
* Порт – выход или вход компонента, через который происходит взаимодействие с другими. Отображается небольшим прямоугольником в одной из сторон компонента.
* Зависимость – представляет связь, означающую, что один компонент зависит от другого и изменение одного из них влечет изменения в другом. Обозначается пунктирной линией с треугольником на конце, исходящей из порта и входящей в него [20].

На основе предыдущих диаграмм (Рисунки 2 и 3), была создана диаграмма компонентов, отображающая общую архитектуру системы (Рисунок 4).

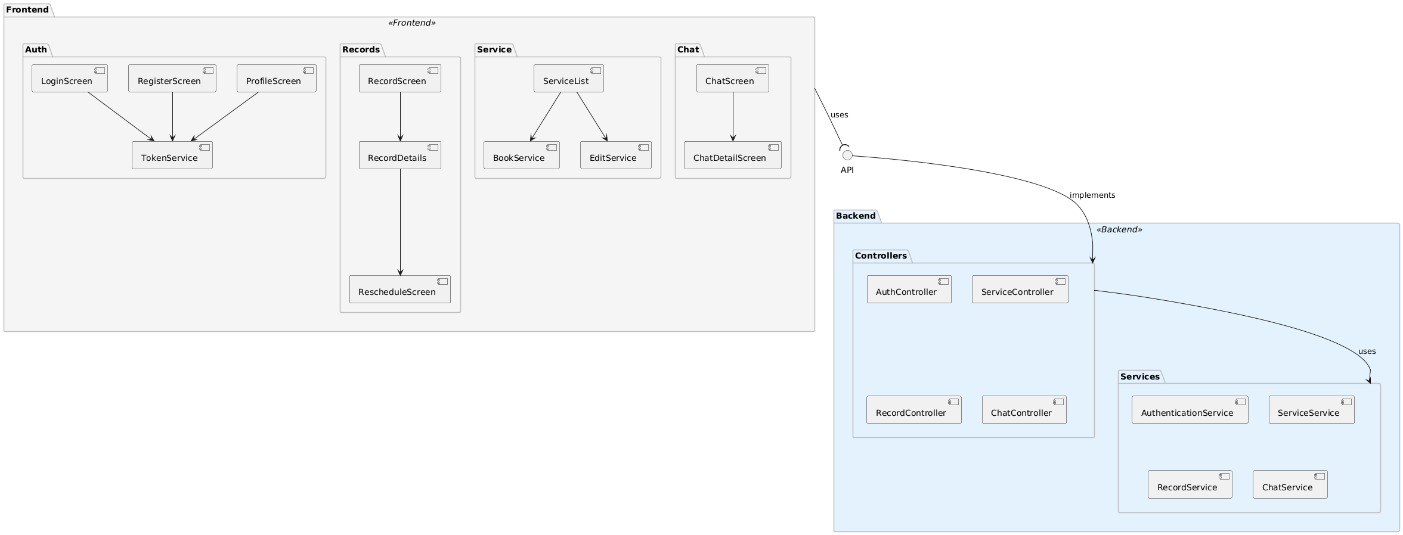


Рисунок 4 – Диаграмма компонентов

2.1.4 Диаграмма развертывания

Диаграмма развертывания – диаграмма UML, отображающая физическое развертывание программной системы [21].

Основные компоненты:

* Узлы – аппаратные устройства, которые используются для развертывания на них программных компонентов;
* Компоненты – программные модули, развертываемые на узлах;
* Артефакты – файлы, находящиеся на узлах и представляющие собой реализацию компонентов (файлы конфигурации, базы данных, скрипты);
* Зависимости – отображают связи между компонентами диаграммы.

Исходя из компонентов, их взаимодействия (Рисунок 4) составлена диаграмма развертывания (Рисунок 5).

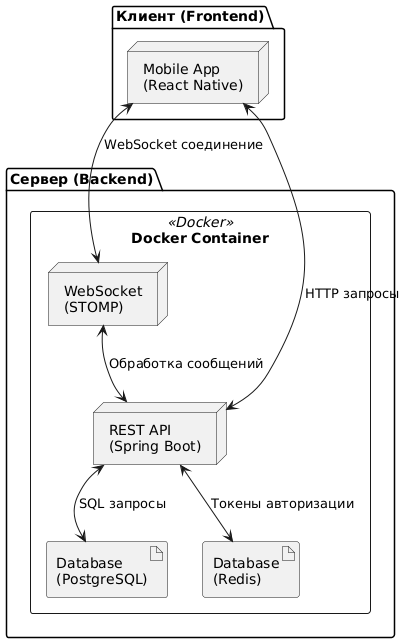


Рисунок 5 – Диаграмма развертывания

2.2 Выводы по главе

В данной главе с помощью диаграмм UML была спроектирована программная система. Диаграммы позволяют получить целостное представление о приложении и его ключевых особенностях.

На первом этапе, исходя из требований к приложению, изложенных в первой главе, была составлена диаграмма прецедентов, которая позволила установить ключевые функции системы, которые нужно реализовать.

Затем исходя из функционала, заложенного в предыдущей диаграмме, была спроектирована диаграмма классов, которая отобразила объектно-ориентированную модель приложения и взаимодействие внутри его основных сущностей.

На основе классов, определенных ранее была составлена диаграмма компонентов, позволяющая понять архитектуру системы в целом, чтобы определить какие сервисы и контроллеры реализовать для организации основной бизнес-логики приложения.

Наконец, реализована диаграмма развертывания, отображающая физическую составляющую системы – базы данных, контейнеры и базы данных.

Таким образом, была смоделирована внутренняя работа системы и её компоненты, благодаря визуальному пониманию которых в дальнейшем менее затратно реализовывать кодовую базу приложения.

ГЛАВА 3. РЕАЛИЗАЦИЯ

3.1 Создание клиентской части

После тщательного проектирования приложения можно приступать к его реализации. Для начала решено было реализовать основную визуальную составляющую клиентского приложения, а затем приступать к реализации серверной части.

3.1.1 Реализация экранов регистрации и авторизации

Наше приложение будет выглядеть по-разному для мастеров и клиентов, поэтому необходимо реализовать регистрацию и авторизацию. Более того, для идентификации клиентов и мастеров, необходимы их учетные записи – это еще одна причина не делать приложение доступным без регистрации и входа.

В пользовательском опыте каждого человека на сегодняшний день настолько много случаев регистрации/авторизации, что даже незнакомому с разработкой будет ясно, что и как должно выглядеть. На основе этого реализованы два первых экрана (Рисунок 6 и Рисунок 7).

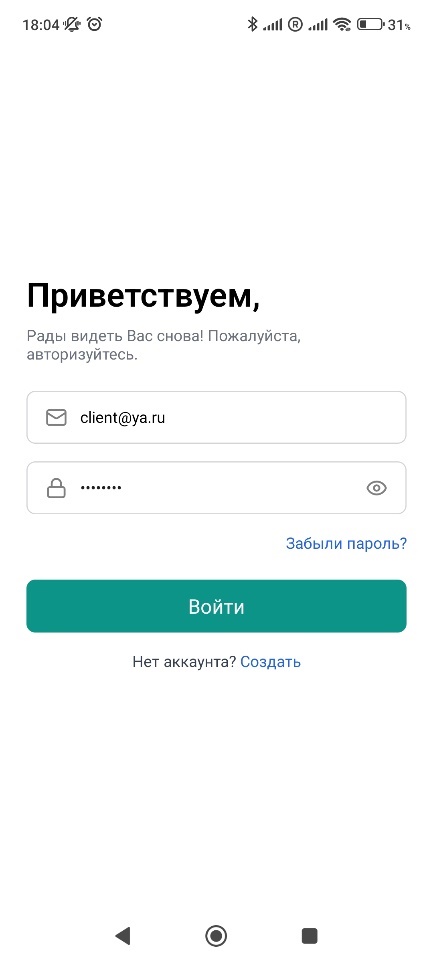


Рисунок 6 – Экран авторизации

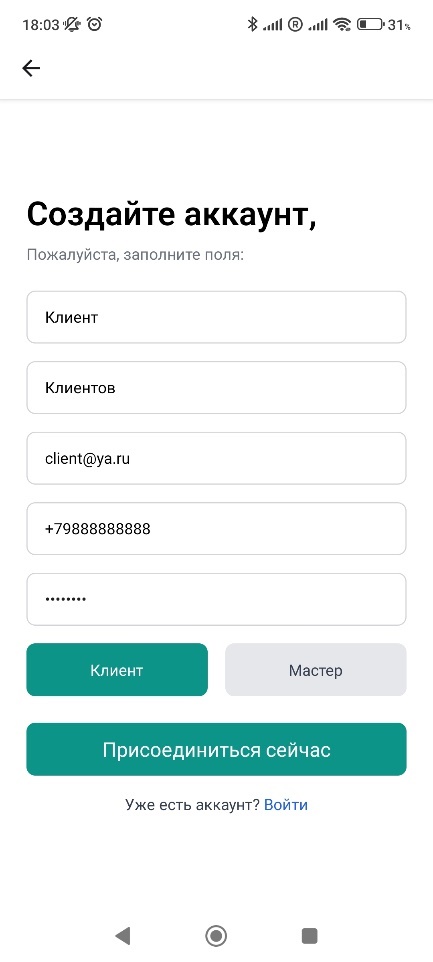


Рисунок 7 – Экран регистрации

На экране регистрации видим основные необходимые поля, необходимые для пользователя в нашей системе: имя, фамилия, электронная почта, номер телефона и пароль, а также роль (мастер или клиент). Для авторизации достаточно ввести почту и пароль и вход будет успешно осуществлен. В будущем, запросы на регистрацию и вход будут доступны всем пользователям, а работа внутри приложения с серверной частью уже после авторизации с получением токена.

3.1.2 Основные вкладки – главная, записи, чаты, личный кабинет

Далее необходимо реализовать основную функциональность приложения, обеспечивающую качественный пользовательский опыт. Для этого обратимся к инструменту Tabs модуля «react-router» фреймворка Expo [22]. Данный тэг позволяет организовать среди основных крупных экранов навигацию в нижней панели (Рисунок 8). На рисунке можем видеть, что выбрана вторая вкладка с значком календаря. Для каждого экрана можно добавлять название, но было принято решение убрать подписи.

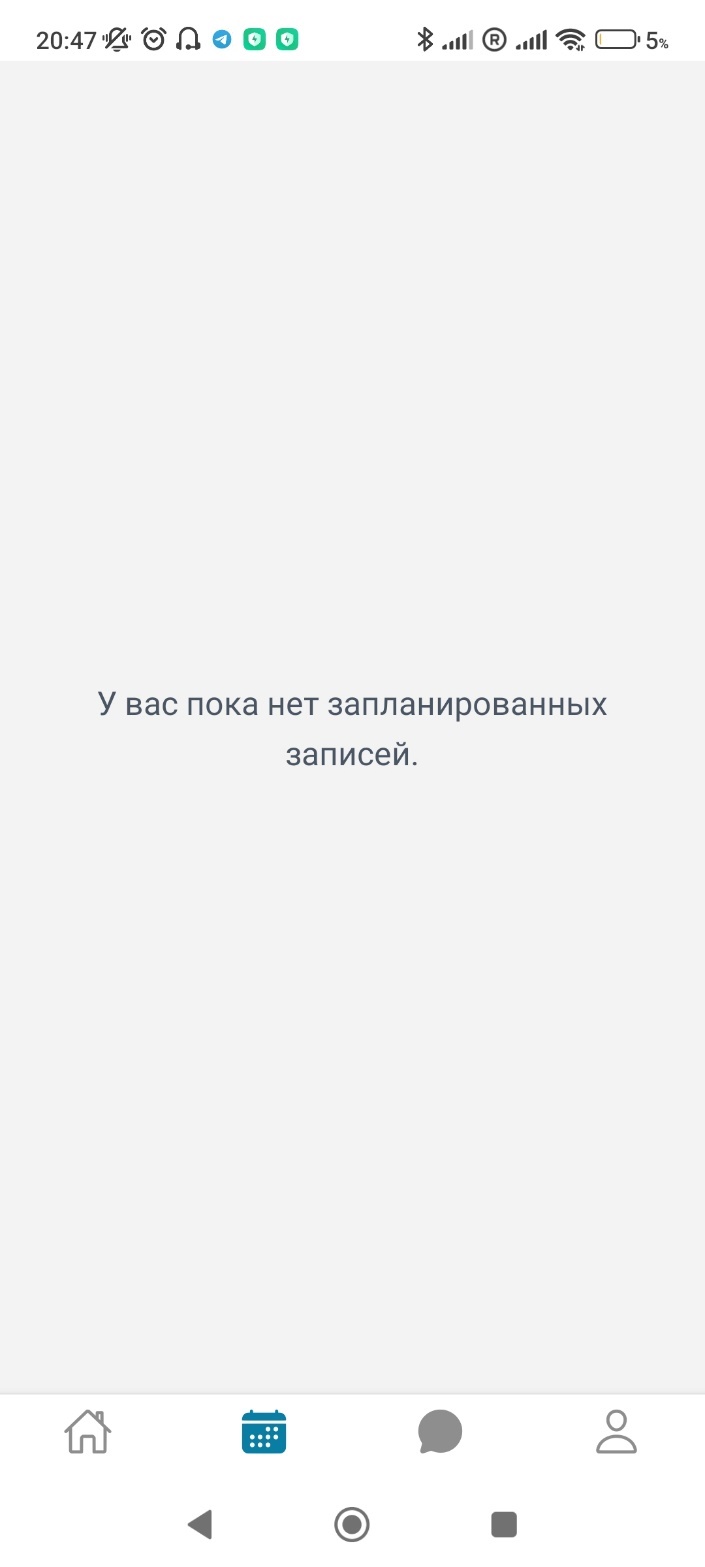


Рисунок 8 – Нижняя панель навигации

Навигация Expo позволяет объявлять основные экраны в области приложения и их параметры (например, верхнюю панель с названием экрана). Реализацию вкладки чатов с бейджем количества непрочитанных можно увидеть на листинге 1:

Листинг 1 – Вкладка экрана чатов нижней панели приложения

<Tabs

screenOptions={{

tabBarActiveTintColor: Colors[colorScheme ?? "light"].tint,

headerShown: false,

}}

>

<Tabs.Screen

name="ChatScreen"

options={{

title: "",

tabBarIcon: ({ color }) => (

<View style={styles.iconContainer}>

<TabBarIcon name="chatbubble" color={color} />

{unreadCount > 0 && (

<Badge size={18} style={styles.badge}>

{unreadCount}

</Badge>

)}

</View>

),

}}

/>

</Tabs>

В переменную unreadCount пишется количество непрочитанных сообщений, пришедших в ответе от сервера. У каждой вкладки можно настроить цвета иконки, само ее изображение, кастомные элементы в виде тэга View и так далее. Таким образом панель не только становится навигатором между основными экранами приложения, но и отображается на каждом из них. При реализации важно отделять экраны, в которых будет нижняя панель, от остальных. Основные экраны нашего приложения следующие: главная, записи, чаты, личный кабинет.

3.1.2.1 Домашняя страница и ее логика

Как упоминалось ранее, внешний вид экранов будет отличаться в зависимости от роли пользователя, например клиент не может создавать услуг, а только записываться на них. Различия экранов мастера и клиента можно увидеть на рисунках 9 и 10.

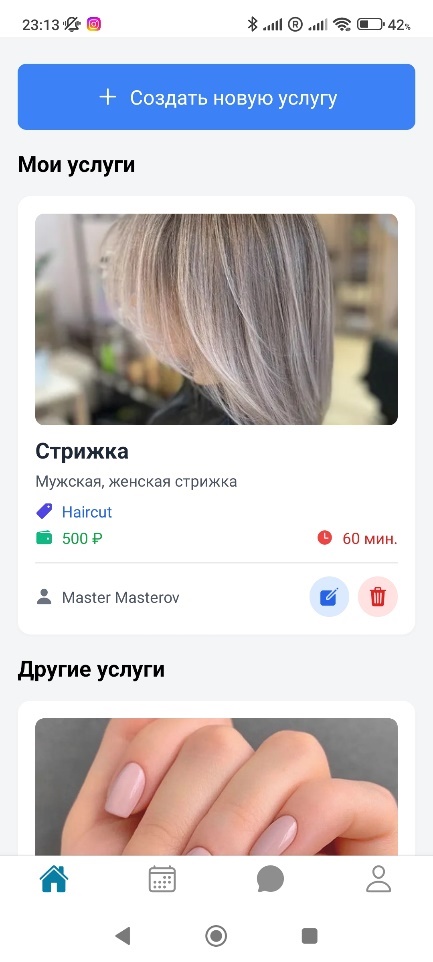


Рисунок 9 – Главный экран мастера

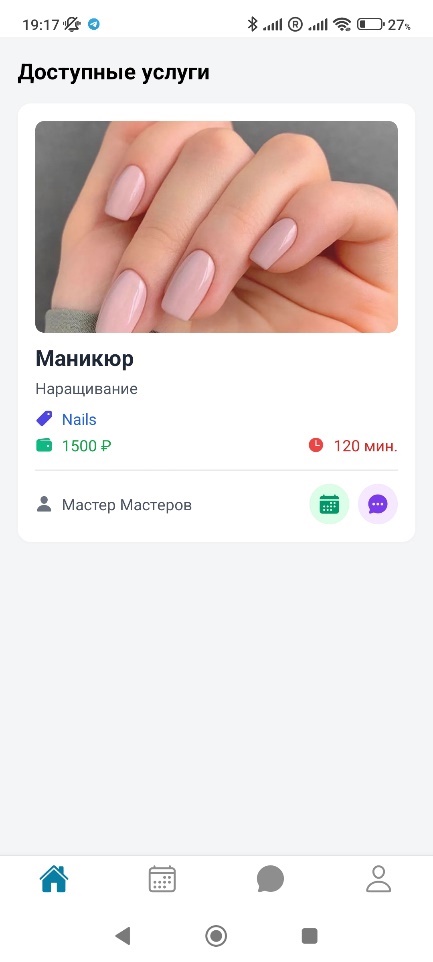


Рисунок 10 – Домашняя страница клиента

На каждой карточке отображена главная информация – фото услуги, название, описание, категория, стоимость, предположительное время и мастер.

Для клиента доступно две дополнительные возможности с главного экрана: записаться на услугу и написать мастеру. Рассмотрим их по порядку.

При нажатии на зеленую иконку с календарем в правом нижнем углу карточки (Рисунок 10) происходит переход на экран записи, на котором отображена основная информация об услуге, а также календарь с выделенными датами, доступными для бронирования (Рисунки 11 и 12).

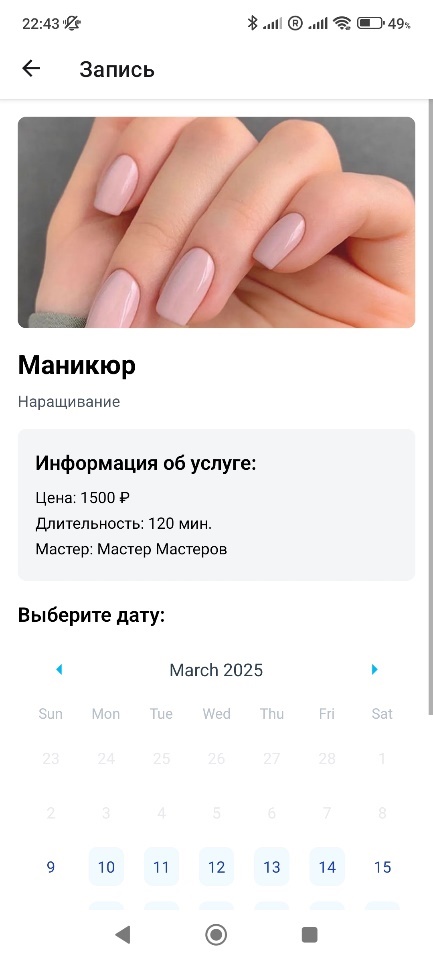


Рисунок 11 – Экран записи на услугу

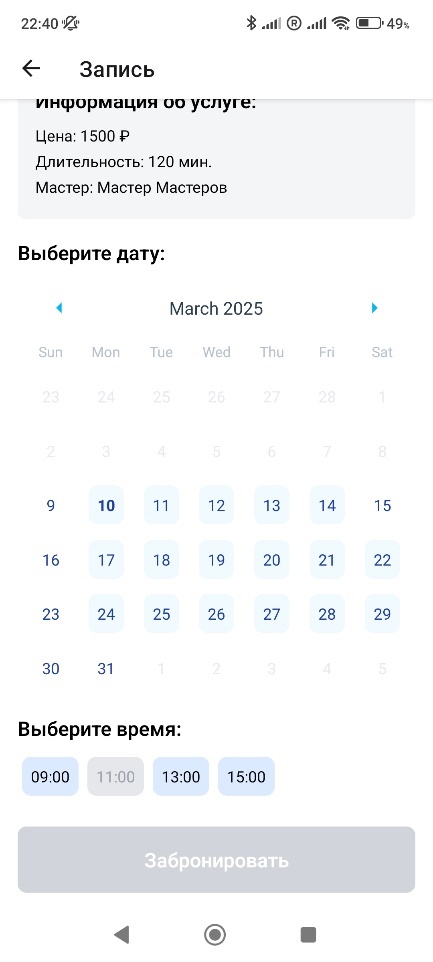


Рисунок 12 – Экран записи на услугу (продолжение)

При нажатии на определенный день календаря отображается время, доступное для записи, при этом время, на которое уже записан клиент, отображается серым цветом и недоступно для нажатия. При выборе времени и нажатии кнопки «Забронировать» (Рисунок 13) выводится информация об успешности (Рисунок 14) и переход на соответствующий экран, где отображается информация о записях.

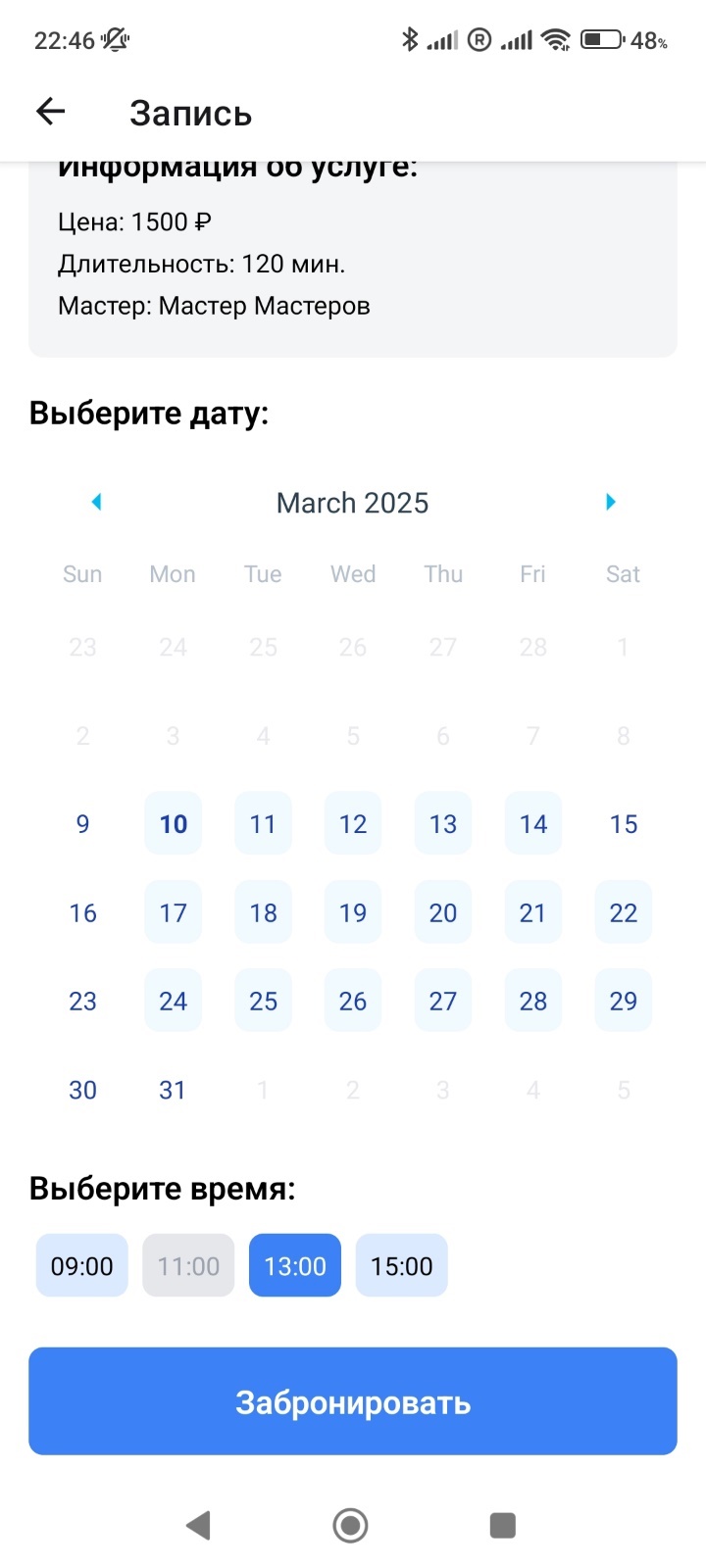


Рисунок 13 – Выбор времени записи и подтверждение

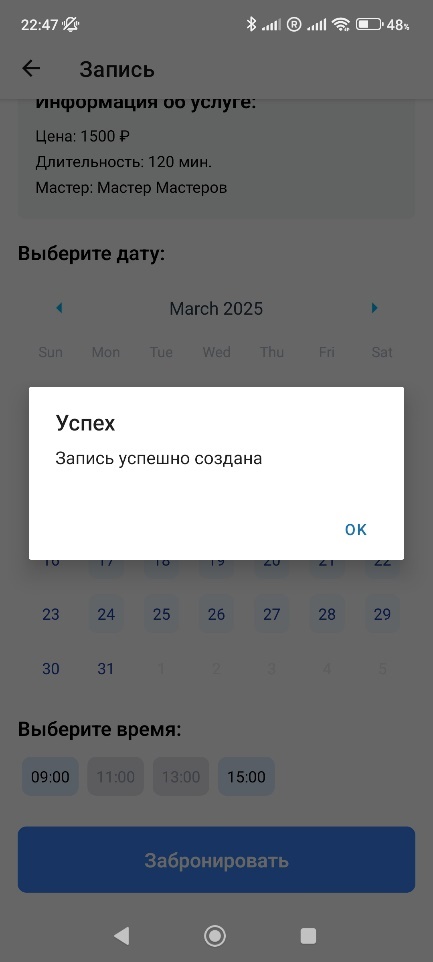


Рисунок 14 – Всплывающее окно успешности записи на услугу

Для мастера, в отличие от клиента, возможностей больше (Рисунок 10). Сверху расположена кнопка создания услуги, при нажатии на которую открывается соответствующий экран (Рисунок 15).

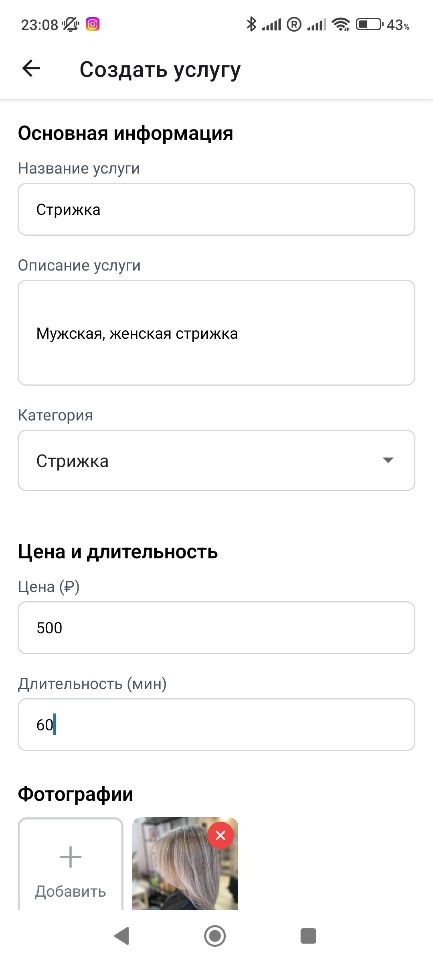


Рисунок 15 – Экран создания услуги

Данная часть экрана содержит в себе основные поля об услуге: название, описание, категорию, стоимость, длительность, фотографии. Основные категории услуг: стрижка, маникюр, лицо, окрашивание, spa, makeup, воск, массаж. Ниже расположен календарь, логику которого можно разделить на три этапа. Первый из них – выбор стандартных тайм-слотов. Предусмотрен он для того, чтобы ускорить выбор дат. Мастеру удобно выбрать стандартное время приема, например 9, 11, 13 и 15 часов (Рисунок 16), а затем выбирать дни, на которые автоматически это время распределится (Рисунок 17). После выбора дней и подтверждения кнопкой «Редактировать тайм-слоты» появляется возможность выбрать конкретный день и внести финальные изменения, например, убрать или добавить время (Рисунок 18).

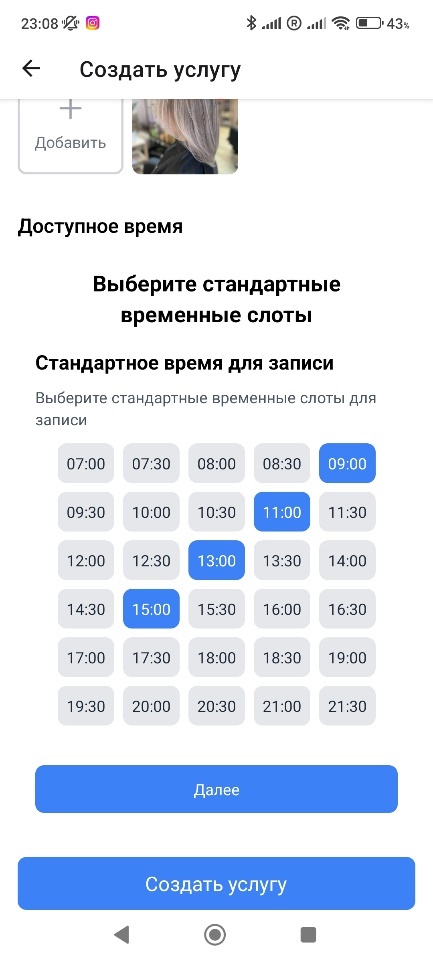


Рисунок 16 – Выбор стандартных тайм-слотов

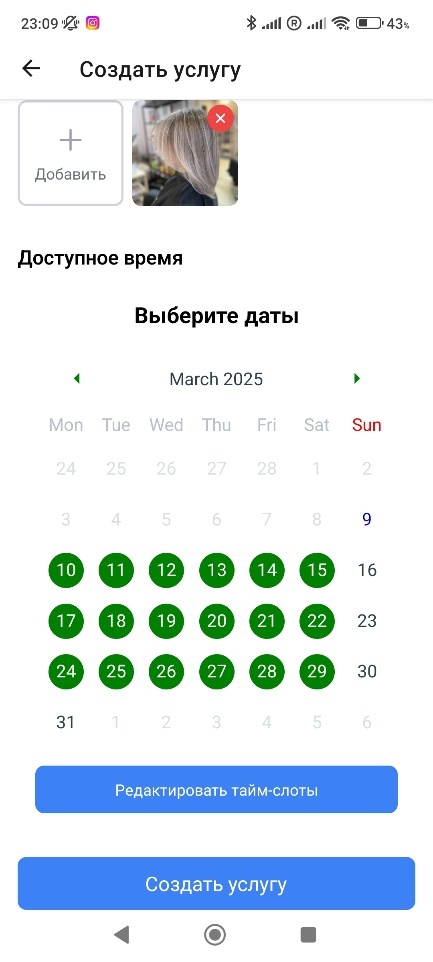


Рисунок 17 – Выбор дней оказания услуги

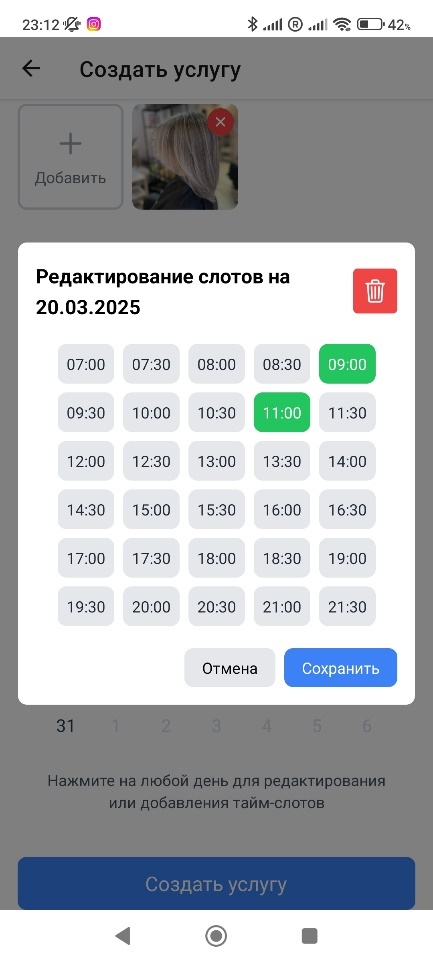


Рисунок 18 – Редактирование тайм-слотов конкретного дня

После нажатия на кнопку «Создать услугу» выводится всплывающее сообщение об успешности и переход на экран услуг.

Мастер также может отредактировать созданную услугу, логика данного экрана аналогична экрану создания. Календарь находится на третьем этапе, когда происходит редактирование тайм-слотов точечно, при нажатии на каждый день. Разумеется, реализовано и удаление услуги через подтверждение действия.

Секция других услуг интерактивна аналогично главному экрану клиента, то есть в данном случае мастер может записаться на услугу и связаться с другим мастером.

3.1.2.2 Экран записей и его логика

Страница записей позволяет клиенту увидеть информацию о предстоящих либо прошедших записях на услуги, а также осуществить перенос или отмену записи, связаться с мастером (Рисунок 19).

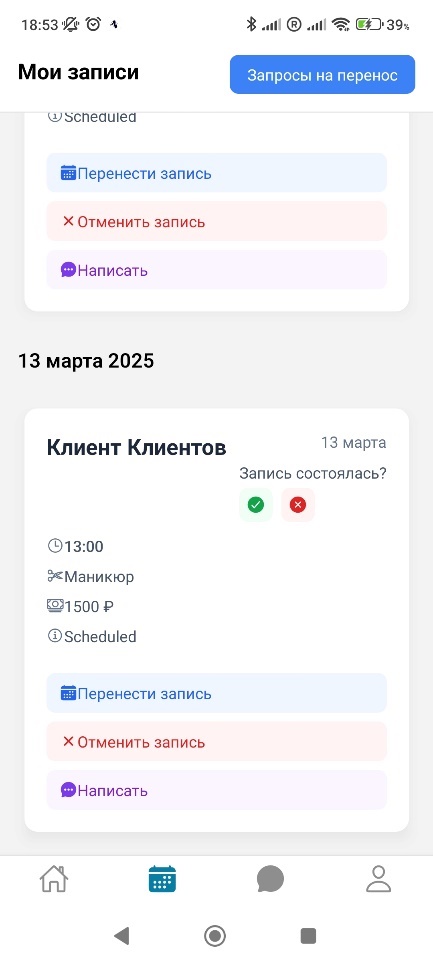


Рисунок 19 – Экран записей

На экране записей пользователю отображаются все его записи. При этом запись можно перенести, для этого существует кнопка «Перенести запись», которая осуществляет открытие календаря с выбором доступных дат и времени для записи (Рисунок 20).

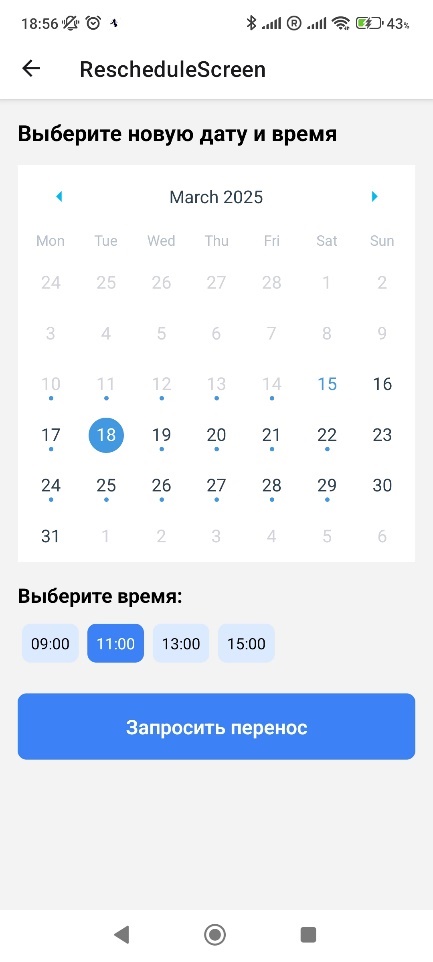


Рисунок 20 – Экран переноса записи

При выборе даты и времени и подтверждении переноса, осуществляется отправка уведомления второй стороне (мастеру или клиенту, соответственно) и ожидается подтверждение на экране запросов на перенос (Рисунок 21).

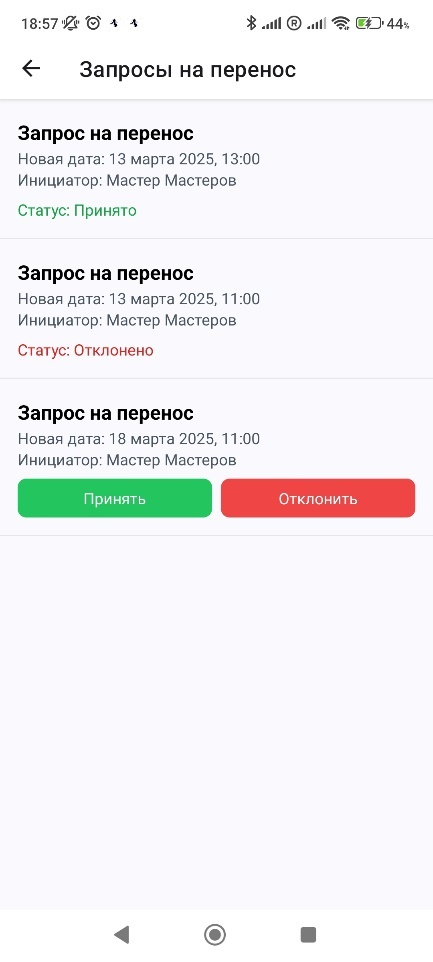


Рисунок 21 – Экран запросов на перенос записей

Уведомление при этом имеет вид, показанный на рисунке 22.

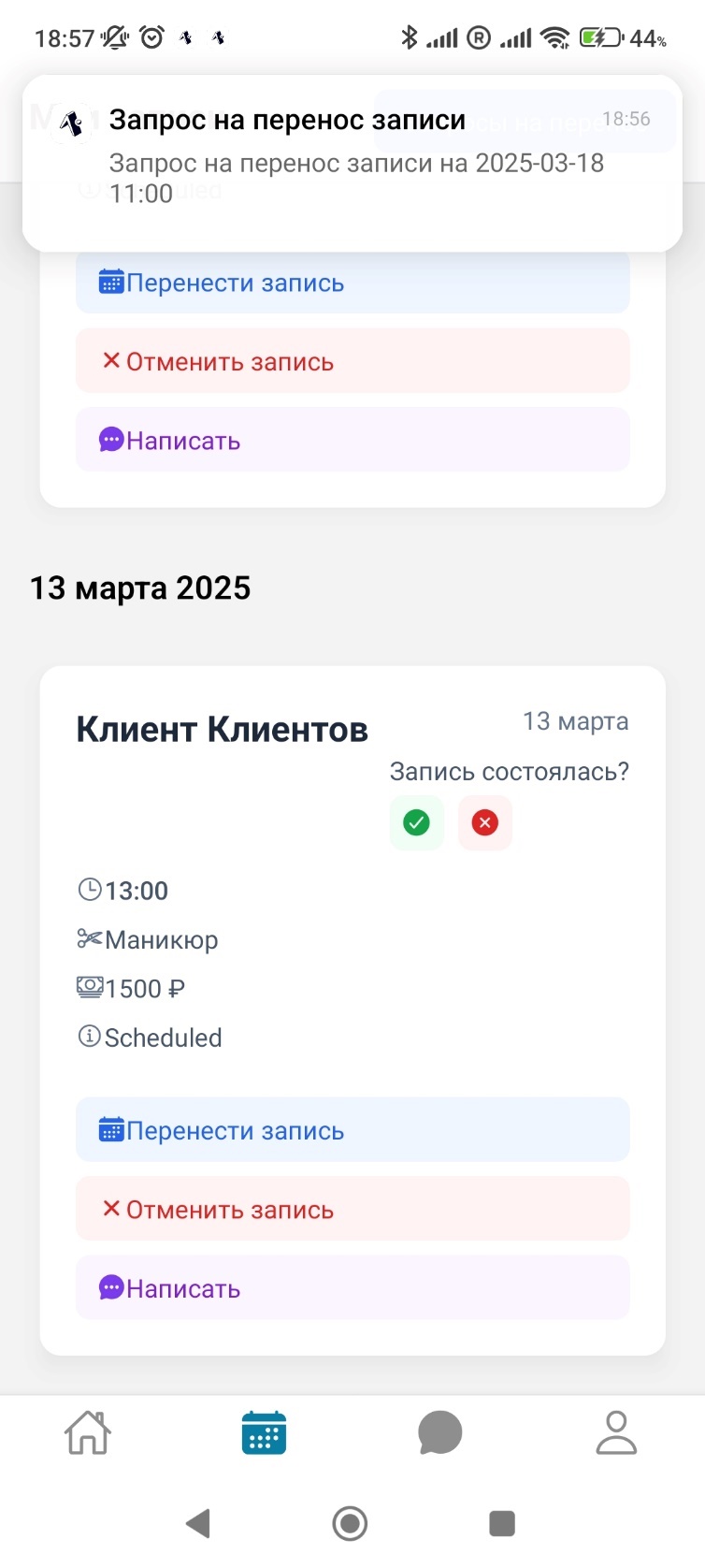


Рисунок 22 – Уведомление о запросе на перенос записи

При нажатии на кнопку «Отменить запись» запрашивается подтверждение операции и происходит удаление, после чего экран обновляется. Нажатие на кнопку «Написать» осуществляет переход к чату с пользователем (клиент или мастер в зависимости от текущей роли).

3.1.2.3 Экран чатов и его логика

Страница чатов необходима для отображения информации о чатах и перехода к конкретному из них. При получении сообщений, их количество отображается и в нижней панели, и непосредственно у чата-источника сообщений (Рисунок 23).

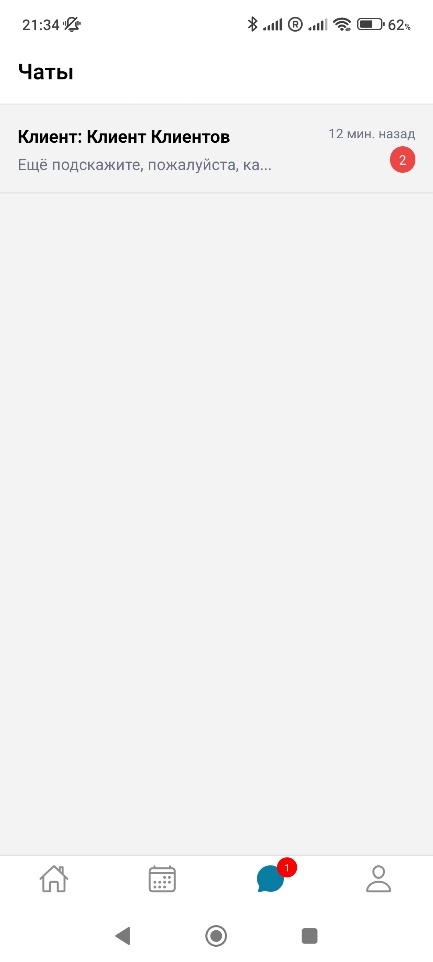


Рисунок 23 – Экран чатов с непрочитанными сообщениями

При этом важно отметить, что на панели вкладок отображается количество чатов с непрочитанными сообщениями, а у конкретного чата – количество непрочитанных в нем. Нажатие на участок с чатом провоцирует его открытие (Рисунок 24). На данном экране можем видеть пользователя, которому пишем, время его последнего нахождения в сети, а также историю сообщений с статусом (прочитано/не прочитано).

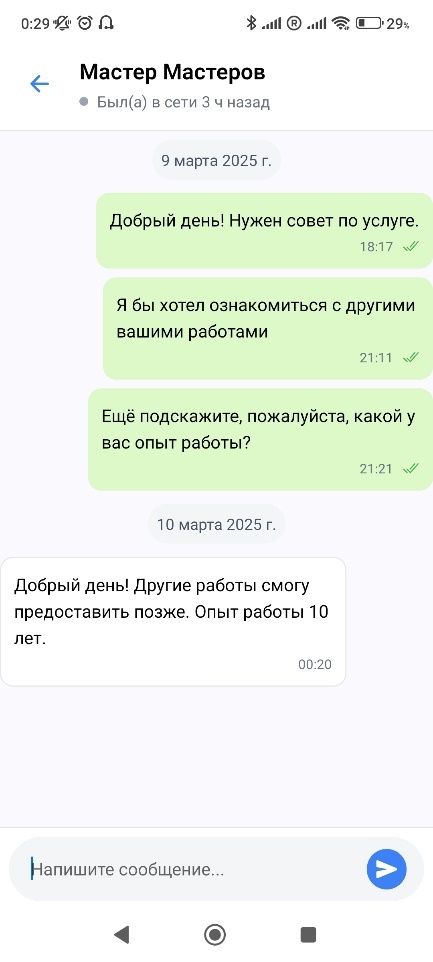


Рисунок 24 – Чат клиента с мастером

3.1.2.4 Экран профиля и его логика

Экран профиля отображает главную информацию о пользователе, позволяет редактировать информацию, настройки приложения, а также выходить из аккаунта (Рисунок 25). При нажатии на кнопку «История записей» происходит переход на вторую вкладку – экран записей.

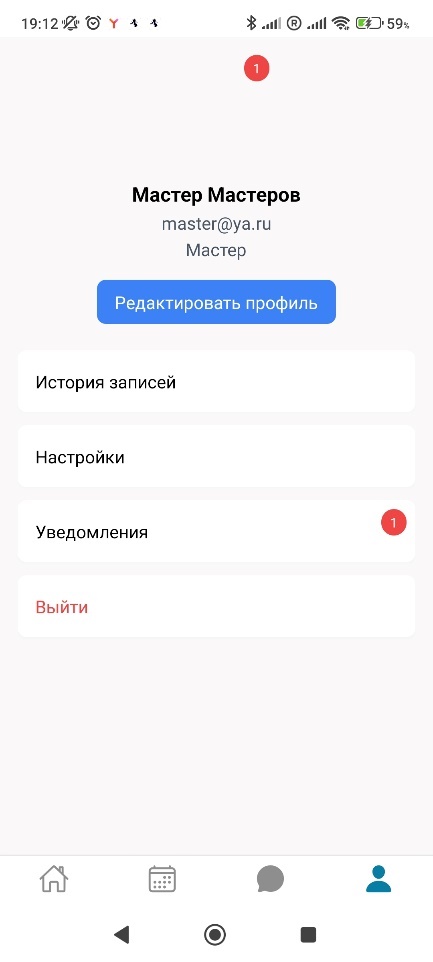


Рисунок 25 – Экран профиля

3.1.2.5 Уведомления

Как уже было отображено на рисунке 22, в приложении организованы уведомления. Их можно разделить на два вида:

1. Напоминания – организованы в серверной части, с определенной периодичностью отправляют мастерам и клиентам напоминания об их предстоящих записях.
2. Информационные – отправляются при каком-то событии, например переносе записи или ее создании.

Для организации уведомлений в клиентской части создан особый сервис уведомлений, который занимается запросом разрешений на отправку и обработкой уведомлений в целом.

Уведомление-напоминание имеет вид, отображенный на рисунке 26.

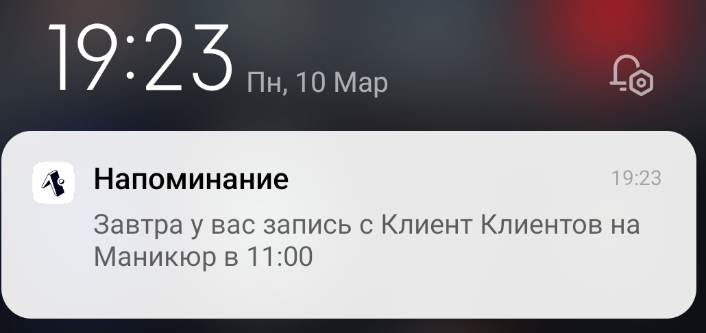


Рисунок 26 – Уведомление-напоминание о записи

Дополнительно для напоминания о непрочитанных уведомлениях организован экран, в который можно попасть через личный кабинет (Рисунок 27).

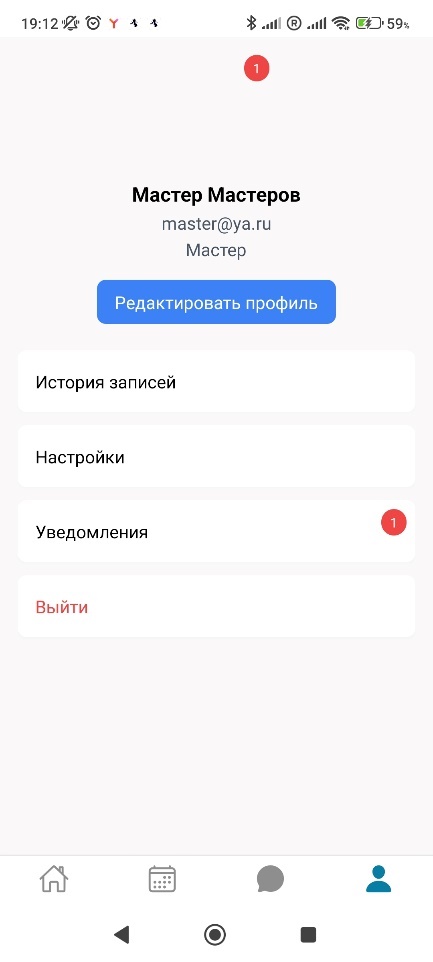


Рисунок 27 – Кнопка уведомлений в личном кабинете

Сам экран уведомлений позволяет читать их и удалять, помогая точно увидеть информацию пользователю, даже если push-уведомление было пропущено (Рисунок 28).

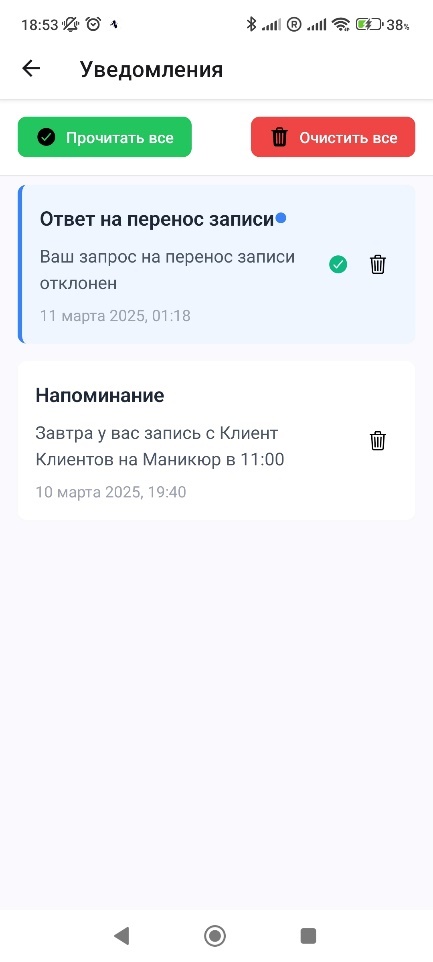


Рисунок 28 – Экран уведомлений

3.2 Реализация серверной части

Серверная часть реализуется на Java Spring Boot. Для ответственного подхода к написанию кода, его легкой отладке и пониманию было решено строить приложение на слоистой архитектуре (англ. Layered architecture) [23]. Зачастую, выделяется 3 стандартных слоя: слой представления, слой бизнес-логики и слой передачи данных. Рассмотрим каждый из них подробнее:

* Презентационный слой – слой, взаимодействующий непосредственно с пользователем. В нашем случае это контроллеры, обрабатывающие запросы от клиентов и отвечающие на них.
* Бизнес-слой – промежуточное звено между слоем представления и слоем передачи данных. Осуществляет основную обработку бизнес-правил, запросов от пользователя и расчетов.
* Слой передачи данных – содержит в себе основные сущности-абстракции, упрощающие работу с данными и обращение к средствам их хранения.

Преимущества использования слоистой архитектуры – понятная иерархия кодовой базы, простота внесения изменений и отладки.

Недостатки – возможная сниженная скорость работы ввиду необходимости прохождения данных через набор слоев, а также созависимость различных компонентов, вызывающая такие проблемы, как циклическая зависимость и другие.

3.2.1 Презентационный слой

Основная составляющая данного слоя – контроллеры. В моем случае их 9:

1. Контроллер аутентификации
2. Контроллер чатов
3. Контроллер файлов
4. Контроллер сообщений
5. Контроллер уведомлений
6. Контроллер записей
7. Контроллер переносов записей
8. Контроллер услуг
9. Контроллер пользователей

Рассмотрим часть их них подробнее.

3.2.1.1 Контроллер аутентификации

Контроллер аутентификации отвечает за регистрацию и авторизацию пользователей. Содержит в себе 4 эндпоинта: «/sign-up», «/sign-in», «/logout», «/refresh». Например, первый из них отвечает за процедуру регистрации, принимает JSON с данными пользователя и необязательный параметр – файл. В процессе работы обращается к сервису аутентификации и в качестве ответа отправляет сущность AuthResponse – токен доступа (англ. access token) и токен обновления (англ. refresh token). Обладая этими токенами, пользователь может взаимодействовать с другими контроллерами. Логично предположить, что эндпоинт «/sign-in» занимается авторизацией пользователя получая логин и пароль и возвращая токены. На листинге 2 представлена логика данного метода контроллера.

Листинг 2 – Логика эндпоинта «/sign-in» контроллера аутентификации

@PostMapping("/sign-in")  
public ResponseEntity<AuthResponse> signIn(@RequestBody @Valid SignInDto signInDto) {  
 *log*.debug("signIn[1]: UserDto: {}", signInDto);  
 return ResponseEntity.*ok*(authenticationService.signIn(signInDto));  
}

Эндпоинт «/refresh» получает на вход RefreshTokenRequest – токен обновления и обновляет токен доступа на его основе. Реализовано это по причине того, что access token обладает временем жизни, равным 30 минутам, для безопасности доступа. Токен обновления существует 7 суток, в течении которых можно обновлять истекшие токены доступа.

3.2.1.2 Контроллер пользователей

Данная часть презентационного слоя предназначена для организации работы с сущностями пользователей. Основные эндпоинты – получение пользователя по идентификатору, обновление данных, удаление аккаунта, получение текущего пользователя по токену доступа, смена статуса (онлайн/оффлайн), загрузка информации о пользователе-собеседнике для чата (англ. interlocutor).

Например, метод обновления пользователя получает идентификатор, DTO (data transfer object) пользователя и необязательное поле – файл, ответом возвращает успешно обновленную сущность пользователя из базы данных. Код, осуществляющий данный функционал, представлен на листинге 3.

Листинг 3 – Логика метода обновления пользователя контроллера пользователей

@PutMapping("/update/{userId}")  
public ResponseEntity<UserDto> updateUser(  
 @PathVariable UUID userId,  
 @RequestPart UserDto userDto,  
 @RequestPart(  
 required = false,  
 name = Constants.*FILE\_REQUEST\_PART\_ID* ) MultipartFile file  
) throws IOException {  
 return ResponseEntity.*ok*(userService.updateUser(userId, userDto, file));  
}

3.2.1.3 Контроллер услуг

Доступ к основной бизнес-логике приложения доступен через контроллер услуг и контроллер записей, рассмотрим первый подробнее. Основные эндпоинты дают доступ к основным CRUD (create-read-update-delete) операциям с услугами, а также к другим необходимым методам, например получению допустимых дат или времени для записи, услуг, созданных мастером и т.д. На листинге 4 представлен код эндпоинтов получения доступных дат и времени, которые получают идентификатор услуги и для второго случая – дату, для которой получить время, а возвращают, соответственно, доступные даты/время.

Листинг 4 – Логика получения доступных дат/времени для записи на услуги

@GetMapping("/{id}/available-slots")  
public ResponseEntity<List<TimeSlotResponse>> getAvailableSlots(  
 @PathVariable UUID id,  
 @RequestParam @DateTimeFormat(iso = DateTimeFormat.ISO.*DATE*) LocalDate date) {  
   
 List<TimeSlotResponse> slots = serviceService.getAvailableTimeSlots(id, date);  
 return ResponseEntity.*ok*(slots);  
}  
  
@GetMapping("/{id}/available-dates")  
public ResponseEntity<List<String>> getAvailableDates(@PathVariable UUID id) {  
 List<String> dates = serviceService.getAvailableDates(id);  
 return ResponseEntity.*ok*(dates);  
}

3.2.1.4 Контроллер записей

Вторым по важности для предоставления доступа к бизнес логики выступает контроллер записей, который помогает пользователям осуществлять запись на услуги. Аналогично предыдущему контроллеру, данный предоставляет возможность осуществлять CRUD операции, а также дополнительные, например смена статуса записи (состоялась/отменилась) или загрузку записей для мастера/клиента. На листинге 5 предоставлен код, осуществляющий функциональность получения всех записей для пользователей обоих ролей.

Листинг 5 – Код получения всех записей по ролям пользователей

@GetMapping  
public ResponseEntity<Map<String, Object>> getRecordsForUser() {  
 List<Record> records = recordService.getRecordsByUser();  
 String userRole = userService.getCurrentUser().getRole().toString();  
   
 Map<String, Object> response = new HashMap<>();  
 response.put("records", records);  
 response.put("userRole", userRole);  
   
 return ResponseEntity.*ok*(response);  
}

3.2.1.5 Контроллер уведомлений

Контроллер предоставляет доступ к рассылке уведомлений на устройства пользователей. Основные эндпоинты – регистрация push-token, CRUD операции с уведомлениями, получение количества непрочитанных и другие. Expo реализовали возможность не настраивать Google Firebase Cloud Messaging, а управлять уведомлениями по их токену и собственному API. Таким образом, наша система получает push-token на мобильном устройстве, отправляет на сервер, а затем, обращаясь к контроллеру, осуществляется рассылка. На листинге 6 представлен код эндпоинта «/send».

Листинг 6 – Метод отправки уведомления контроллера

@PostMapping("/send")  
public ResponseEntity<String> sendPushNotification(  
 @RequestParam String token,  
 @RequestParam String title,  
 @RequestParam String body) {  
  
 expoNotificationService.sendNotification(token, title, body);  
 return ResponseEntity.*ok*("Push notification sent successfully");  
}

3.2.2 Бизнес-слой

Основная бизнес-логика приложения заложена в этом слое. Для осуществления операций, представленных в контроллерах, используются компоненты – сервисы, которые оперируют с объектами сущностей и обрабатывают данные.

Сервисов в нашей системе так же много, как и контроллеров, для более четкой организации их иерархия организована в виде интерфейсов, методы которых имплементируются классами. Так у разработчика появляется возможность не вдаваться в подробности реализации тех или иных методов, а сразу увидеть их список, найти необходимый и использовать.

Рассмотрим некоторые из сервисов.

3.2.2.1 Сервис услуг

Основные методы интерфейса услуг изложены на листинге 7.

Листинг 7 – Методы сервиса услуг

public interface ServiceService {

Service createService(Service service, List<MultipartFile> files);

Service updateService(UUID serviceId, Service serviceDto, List<MultipartFile> newFiles, List<String> replaceFilesUrls) throws ApiException;

void deleteService(UUID serviceId);  
 Service getServiceById(UUID serviceId);  
 List<Service> getAllServices();  
 ServicesDto getServicesForMaster();  
 List<Service> getServicesByCategory(Category category);  
 List<TimeSlotResponse> getAvailableTimeSlots(UUID serviceId, LocalDate date);  
 List<String> getAvailableDates(UUID serviceId);  
 List<Service> getServicesByMasterId(UUID masterId);  
}

Видим, что сервис осуществляет основные операции создания, чтения, обновления, удаления, а также некоторые вспомогательные операции для удобства использования в презентационном слое. Имплементация метода создания услуги изложена на листинге 8.

Листинг 8 – Метод создания услуги

@Override  
public Service createService(Service serviceDto, List<MultipartFile> files) {  
 User master = userService.getCurrentUser();  
 List<String> fileNames;  
 List<String> serviceImageUrl = new ArrayList<>();  
  
 if(files != null) {  
 fileNames = fileService.uploadFileList(serviceFolder, files);  
 serviceImageUrl.addAll(fileNames);  
 }  
  
 Service service = Service.*builder*()  
 .price(serviceDto.getPrice())  
 .title(serviceDto.getTitle())  
 .description(serviceDto.getDescription())  
 .category(serviceDto.getCategory())  
 .master(master)  
 .photos(serviceImageUrl)  
 .estimatedDuration(serviceDto.getEstimatedDuration())  
 .availability(serviceDto.getAvailability())  
 .build();  
  
 *log*.info("createService[1]: Creating service {}", service);  
  
 Service savedService = serviceRepository.save(service);  
  
 *log*.info("createService[2]: Service created with ID {}", savedService.getId());  
  
 return savedService;  
}

Из кода метода видно, что осуществляется получение информации о создаваемой услуги из DTO, а также файлы изображений, которые будут прикреплены. Происходит обращение к сервису файлов для их сохранения, а затем создание сущности услуги из полученной информации. Финальным шагом выступает обращение к интерфейсу репозитория услуг, который обеспечивает удобную работу с базой данных, и сохранение сущности. Следует обратить внимание, что в сервисах предусмотрено логирование для отслеживания процесса работы приложения.

3.2.2.2 Сервис записей

Обратимся сразу к имплементации методов и приведем в пример функциональность обновления статуса записи, приведенную в Приложении А. Из реализации видим получение идентификатора и нового статуса в качестве параметров, а затем разветвление логики в зависимости от пользователя, изменяющего статус, ведь клиент не может отменять запись, если до ее назначенного времени осталось менее 24 часов, в отличие от мастера, которому такая возможность доступна. При изменении статуса второй стороне отправляется push-уведомление о том, что произошло изменение записи, в частности для случаев отмены записи. При подтверждении статуса записи, как состоявшаяся, клиенту не так важно быть об этом уведомленным.

3.2.2.3 Сервис сообщений

Данный сервис занимается взаимодействием с Websocket [24], в частности STOMP [25], которые позволяют отправлять информацию по особому протоколу мгновенного обмена, причем эта версия обмена данными отличается особенностью – возможностью сервера отправлять запросы не в качестве ответа на запрос клиента, а самостоятельно. Так, клиентское приложение подписывается на канал получения сообщений и может получать их оттуда. На листинге 9 приведен код, осуществляющий сохранение сообщения в базу данных.

Листинг 9 – Метод отправки сообщения

@Override  
public Message sendMessage(SendMessageRequest request) {  
 User user = userService.getUserById(request.getUserId());  
 Chat chat = chatService.getChatById(request.getChatId());  
  
 if (messageRepository.existsByContentAndSenderAndCreatedAtAfter(  
 request.getContent(),  
 user,  
 LocalDateTime.*now*().minusMinutes(1)  
 )) {  
 throw new ApiException(HttpStatus.*BAD\_REQUEST*, "Duplicate message");  
 }  
  
 Message message = Message.*builder*()  
 .chat(chat)  
 .sender(user)  
 .content(request.getContent())  
 .replyToMessageId(request.getReplyToMessageId())  
 .createdAt(LocalDateTime.*now*())  
 .isRead(false)  
 .build();  
  
 Message savedMessage = messageRepository.save(message);  
  
 List<Message> messages = findAllMessagesByChatId(chat.getId());  
 chatService.updateMessageCount(chat.getId(), messages.size());  
  
 return savedMessage;  
}

Из логики кода видно, что осуществляется проверка сообщения на дубликат, а затем сохранение в базу данных и увеличение счетчика сообщений в чате, в который отправлено сообщение. Стоит отметить, что сама отправка сообщения в STOMP была реализована в контроллере, который строит сущность сообщения и отправляет по Websocket (Листинг 10).

Листинг 10 – Метод отправки сообщения в контроллере сообщений

@MessageMapping("/chat")  
public void processMessage(@Payload SendMessageRequest sendMessageRequest) {  
 Message savedMessage = messageService.sendMessage(sendMessageRequest);  
  
 messagingTemplate.convertAndSend(  
 "/queue/messages/" + sendMessageRequest.getChatId().toString(),  
 ChatNotificationDto.*builder*()  
 .id(savedMessage.getId())  
 .chatId(savedMessage.getChat().getId())  
 .replyToMessageId(savedMessage.getReplyToMessageId())  
 .senderId(savedMessage.getSender().getId())  
 .content(savedMessage.getContent())  
 .build()  
 );  
}

3.2.3 Слой взаимодействия с базами данных

В нашей системе огромная часть данных должна быть сохранена в хранилище. Для того, чтобы не взаимодействовать с базами данных напрямую и не генерировать SQL-запросы для каждого случая, реализован механизм ORM (object relation mapping), при котором создаются абстракции сущностей, которые будут представлять собой таблицы. Для взаимодействия с сущностями в Hibernate ORM реализуются репозитории – интерфейсы, в которых определенным образом пишутся методы, реализацию которых вручную производить во многих случаях нет необходимости – они уже реализованы разработчиками JPA (java persistence API) и Hibernate. В случае особенных, более трудных случаев, предоставляется возможность написать запрос к базе данных вручную и реализовать в виде метода.

Например, сущность записи имеет вид (Листинг 11):

Листинг 11 – Сущность записи в системе

@Entity  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Table(name = "records")  
@Builder  
@Getter  
@Setter  
public class Record {  
 @Id  
 @GeneratedValue  
 @UuidGenerator(style = UuidGenerator.Style.*TIME*)  
 private UUID id;  
  
 @ManyToOne  
 @JoinColumn(name = "master\_id")  
 private User master;  
  
 @ManyToOne  
 @JoinColumn(name = "client\_id")  
 private User client;  
  
 @ManyToOne  
 @JoinColumn(name = "service\_id")  
 private Service service;  
  
 @Column(name = "record\_date")  
 @JsonFormat(pattern = Constants.*LOCAL\_DATETIME\_FORMAT*)  
 @JsonSerialize(using = LocalDateTimeSerializer.class)  
 private LocalDateTime recordDate;  
  
 @Column(name = "record\_status")  
 @Enumerated(value = EnumType.*STRING*)  
 private RecordStatus recordStatus;  
}

В сущности можно задавать основные параметры – название таблицы, столбцов, перечислений, а также типы данных, хранимые в столбцах и другие параметры, привычные для использования в базах данных.

Перейдем к примеру взаимодействия с сущностями. Как уже было сказано выше, нет необходимости реализовывать каждый метод вручную, есть набор синтаксических правил, например, как называть методы в интерфейсе, которые помогают обратиться к методам JPA или Hibernate. Код взаимодействия с сущностью записи отображен на листинге 12.

Листинг 12 – Репозиторий записей

@Repository

public interface RecordRepository extends JpaRepository<Record, UUID> {  
 Record save(Record record);  
  
 Optional<Record> findById(UUID recordId);  
  
 void deleteById(UUID recordId);  
  
 List<Record> findAllByClientId(UUID clientId);  
  
 List<Record> findAllByMasterId(UUID masterId);  
  
 List<Record> findAllByClientId\_OrMaster\_Id(UUID client, UUID master);  
  
 List<Record> findByServiceIdAndRecordDateAfter(UUID serviceId, LocalDateTime date);  
  
 boolean existsByServiceIdAndRecordDateAfter(UUID serviceId, LocalDateTime now);  
  
 List<Record> findByRecordDateBetween(LocalDateTime start, LocalDateTime end);  
}

Как можно заметить, обязательна аннотация @Repository для того, чтобы Spring Boot корректно понимал предназначение репозитория. Для каждого метода задаются возвращаемые данные, а также передаваемые параметры. Из названия каждой функции ясно, в чем состоит её предназначение. Однако в данном примере представлены только стандартные случаи, уже реализованные в библиотеках. Рассмотрим более сложный метод, который требовался при разработке (Листинг 13).

Листинг 13 – Метод, отмечающий все уведомления в базе данных прочитанными для данного пользователя

@Modifying

@Query("UPDATE Notification n SET n.read = true WHERE n.user.id = :userId AND n.read = false")  
void markAllAsReadForUser(UUID userId);

Из реализации видно, что посылается запрос на обновление таблицы уведомлений и установка полей прочитано в True там, где они ранее были False и id пользователя равен тому, что передан в параметрах.

3.2.4 Логирование

Логирование кода очень необходимо для отслеживания работы приложения, а также его отладки. В нашем приложении эта функциональность реализована с помощью библиотеки Lombok [26] и ее аннотации @Slf4j [27].

Аннотация открывает доступ к переменной log и её методам info, debug, error, параметры которых – строка, выводимая в лог. На листинге 14 приведен пример кода лога одного из методов.

Листинг 14 – Строка лога

*log*.info("updateRecordStatus[1]: Searching for record with id: {}", recordId);

Для удобства структуризации и понимания логов, я структурировал их сообщение по следующему шаблону: название\_метода[номер\_лога\_в\_функции]: сообщение.

3.2.5 Документирование

Помимо логирования очень важным является процесс документирования, который позволяет оставить для других разработчиков краткое описание того, что делает метод, а также его принимаемых и возвращаемых параметрах. Очень необходимо в случаях, когда давно не осуществлялась работа с кодом.

Реализована в нашем приложении документация с помощью JavaDoc []. Она позволяет в лаконичном и удобном формате оформить документацию прямо над методом. Пример одной из документаций приведен на листинге 15.

Листинг 15 – JavaDoc для метода создания записи на услугу

*/\*\*  
 \* Create a new record. The record is scheduled for the given date and time, and the master is set to the recipient  
 \* of the request. The method first checks that the date and time are valid, and then checks that the master is  
 \* available at that time. If any of these checks fail, an ApiException is thrown. If the record is successfully  
 \* created, the method sends a notification to the master about the new record.  
 \*  
 \* @param recordDto the record to create  
 \* @return the created record  
 \* @throws ApiException if the date and time are invalid, or if the master is not available at that time  
 \*/*

3.3 Вывод по главе

Данная глава отобразила основные детали реализации программной системы. Разработана клиентская часть приложения, ее реализация на React Native и основные экраны, взаимодействие с API, а также серверная часть, ее архитектура, контроллеры, сервисы и репозитории. Реализовано логирование и документация кода.

ГЛАВА 4. ТЕСТИРОВАНИЕ

4.1 Тестирование серверной части

Немногие приложения могут обойтись без этапа тестирования – оценки качества разработанного продукта с целью выявления недоработок и соответствия требуемой функциональности программы [29].

Тестирование серверной части приложения можно разделить на несколько видов:

1. Структурное – проверка соответствия данных в системе ожидаемым, после процесса перевода сущностей в другие или просто работы с ними;
2. Функциональное – проверка функциональности отдельных модулей приложения (например, проверка работоспособности модуля авторизации на соответствие ожидаемому результату);
3. Нефункциональное тестирование – другими словами его чаще всего называют нагрузочным – проверка способности приложения выдерживать большой поток пользователей и вычислительных процессов.

В контексте приложения больше всего нас интересует фунциональное тестирование (англ. unit – модуль).

4.2 Разработка unit-тестов

Для проверки выбраны основные функции контроллеров и сервисов. В случае с контроллерами это позволит протестировать отправку запроса на эндпоинт и получение корректного ответа от сервера, а при проверке сервисов можно будет более точечно просмотреть корректную трансформацию данных.

Самыми популярными инструментами для тестирования серверной части приложений Java выступают Junit [30] и Mockito [31]. При этом в нашем случае важна их комбинация, ведь первый инструмент позволяет проводить тесты в целом и предоставляет доступ к таким аннотациям, как @Test и @BeforeEach, которые дают возможность простроить функциональность теста и провести нужные проверки соответствия ожидаемых и полученных результатов, а второй позволяет осуществлять мокирование – процесс имитации компонентов приложения. То есть вместо того, чтобы разворачивать систему в целом, мы можем проверять отдельные компоненты, просто задавая им необходимые зависимости и их поведение. Это незаменимо в приложениях с разветвленной архитектурой.

4.2.1 Тестирование контроллеров

Контроллер – элемент презентационного слоя приложения, соответственно он получает данные от клиента и отдает ему результат их обработки. Это и проверяется в тестах, однако важно понимать, как происходит процесс. С помощью аннотаций @Mock, @InjectMocks и других мы можем объявлять компоненты, поведение которых будем имитировать, например в случае с тестом авторизации сервис аутентификации будет моком и будет инжектироваться в контроллер аутентификации (Листинг 16).

Листинг 16 – Мокирование компонентов

@Mock  
private AuthenticationService authenticationService;  
  
@InjectMocks  
private AuthController authController;  
  
@BeforeEach  
void setUp() {  
 MockitoAnnotations.*openMocks*(this);  
}

С помощью аннотации @BeforeEach можно настроить предварительную конфигурацию перед каждым тестом, в нашем случае это открытие моков.

Основной код теста находится в листинге 17, из его компоновки можно выяснить, что создается пример сущности DTO, которая будет получена от клиента и создается ответ на регистрацию. Затем моку сервиса аутентификации говорится, что при вызове метода с данными, содержащими класс DTO, созданный выше, необходимо вернуть тот самый ответ, уже созданный нами. Затем с этими же данными вызывается метод контроллера и проверяется его код ответа и тело, при совпадении тест считается пройденным, что в нашем случае и произошло.

Листинг 17 – Код тестирования регистрации

@Test  
void testSignUp() throws IOException {  
 UserDto userDto = new UserDto();  
 userDto.setEmail("test@example.com");  
 userDto.setPassword("password");  
  
 AuthResponse authResponse = new AuthResponse();  
 authResponse.setAccessToken("token");  
  
 *when*(authenticationService.signUp(*any*(UserDto.class), *any*())).thenReturn(authResponse);  
  
 ObjectMapper objectMapper = new ObjectMapper();  
 String userJson = objectMapper.writeValueAsString(userDto);  
  
 MockMultipartFile file = new MockMultipartFile("file", "test.txt", "text/plain", "test".getBytes());  
  
 ResponseEntity<AuthResponse> response = authController.signUp(userJson, file);  
  
 *assertEquals*(200, response.getStatusCodeValue());  
 *assertEquals*("token", response.getBody().getAccessToken());  
}

4.2.2 Тестирование сервисов

Проверив контроллеры, следует убедиться, что и компоненты, использующиеся в них, исправны. Для этого реализованы тесты сервисов. Например, на листинге 18 приведен код теста сервиса сообщений, в частности проверки их корректной отправки и сохранения. Для этого мы создаем необходимые случайные сущности (чат, пользователя и сообщение) и отправляем запрос на их отправку, а затем сверяем результаты.

Листинг 18 – Код теста отправки сообщения

@Test  
void testSendMessage() {  
 SendMessageRequest request = new SendMessageRequest();  
 request.setChatId(UUID.*randomUUID*());  
 request.setContent("Test message");  
  
 User user = new User();  
 user.setId(UUID.*randomUUID*());  
  
 Chat chat = new Chat();  
 chat.setId(request.getChatId());  
  
 Message message = new Message();  
 message.setId(UUID.*randomUUID*());  
  
 *when*(userService.getCurrentUser()).thenReturn(user);  
 *when*(chatService.getChatById(*any*(UUID.class))).thenReturn(chat);  
 *when*(messageRepository.save(*any*(Message.class))).thenReturn(message);  
  
 Message sentMessage = messageService.sendMessage(request);  
  
 *assertNotNull*(sentMessage);  
 *assertNotNull*(sentMessage.getId());  
}

4.3 Ручное тестирование

Разумеется, при написании приложения не всегда целесообразно под каждый случай писать соответствующий тест, хотя это и считается хорошей практикой. Для таких случаев быстрее вручную проверить, работоспособно ли приложение – частично или в целом.

Незаменимый инструмент, который используется практически каждым разработчиком серверной части – Postman [32].

Функционал приложения огромен, но основной из них, необходимый нам – отправка различного рода запросов на те или иные эндпоинты нашей программной системы. На рисунке приведен пользовательский интерфейс приложения, отображающий запрос к API на получение текущего пользователя (Рисунок 29).

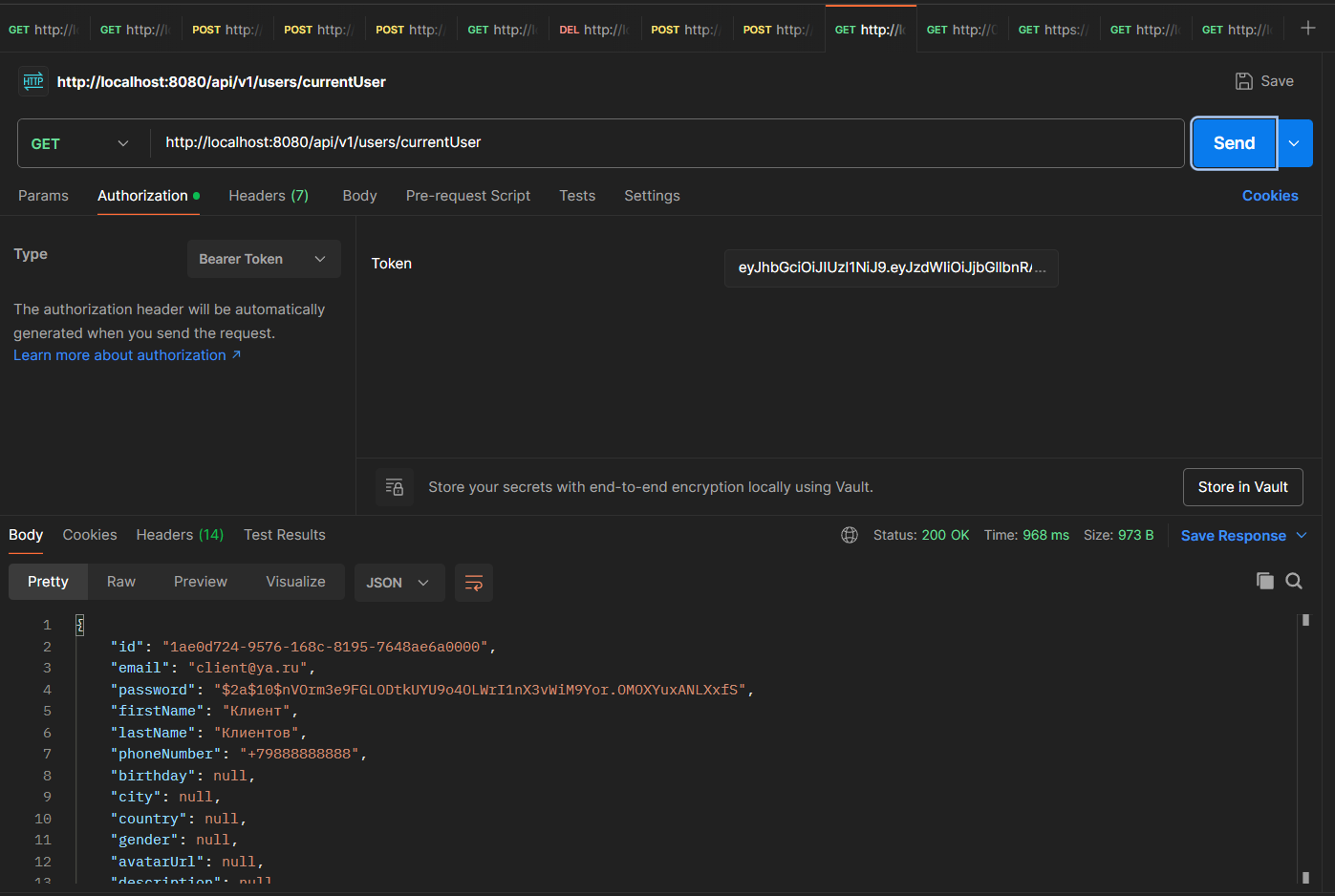


Рисунок 29 – Запрос текущего пользователя в Postman и его результаты

4.4 Выводы по главе

Таким образом, в текущей главе была рассмотрена прикладная задача тестирования и его виды. Проведено функциональное тестирование основных модулей приложения, а также тестирование с помощью инструмента Postman. В результате проверки работоспособности приложения выявлено, что оно выполняет свои основные задачи и корректно обрабатывает пограничные случаи.

ГЛАВА 5. БЕЗОПАСНОСТЬ ЖИЗНЕДЕЯТЕЛЬНОСТИ

5.1 Понятие безопасности жизнедеятельности

Безопасность жизнедеятельности – это набор мер, направленных на защиту здоровья, снижение риска несчастных случаев и обеспечение условий для эффективной работы. Для программиста это становится ключевым, потому что длительная работа за компьютером может нанести физические травмы, но создание удобного и функционального рабочего места способны помочь избежать негативных последствий.

5.2 Основные правила безопасности жизнедеятельности при обращении с компьютером

Для сохранения здоровья при работе за компьютером важно соблюдать следующие правила:

1. Экран компьютера должен находиться на расстоянии 50–70 см от глаз, а его верхняя часть – на уровне глаз.
2. Использовать комбинацию естественного и искусственного света для освещения.
3. Регулярно проветривать помещение, поддерживать комфортную температуру и влажность.
4. Необходимо делать короткие перерывы каждые 50–60 минут, чтобы снизить нагрузку на зрение и мышцы.
5. Необходимо проводить небольшие упражнения для глаз и лёгкую разминку.
6. Использовать специальные программы для настройки яркости и контрастности экрана.
7. Применять правило «20-20-20»: каждые 20 минут делайте паузу и в течение 20 секунд смотрите на объект, находящийся примерно в 6 метрах от вас.
8. Расположите клавиатуру и мышь так, чтобы минимизировать нагрузку на запястья и плечи.
9. Используйте регулируемые столы и кресла, а также подставки для ноутбуков, чтобы поддерживать правильную осанку.
10. Планируйте свой рабочий день и равномерно распределяйте задачи.
11. Регулярно занимайтесь физическими упражнениями, гуляйте на свежем воздухе и уделяйте время любимым занятиям.
12. Регулярно проверяйте состояние электропроводки и оборудования.

5.3 Выводы по главе

Соблюдение правил безопасности жизнедеятельности – неотъемлемая часть работы программиста. Организовав удобное и эргономичное рабочее место, правильно распределяя время работы и отдыха, а также соблюдая основные правила безопасности, можно не только сохранить здоровье, но и значительно повысить эффективность труда. Внедрение этих простых мер помогает минимизировать профессиональные риски и создавать условия для стабильной и продуктивной работы.

ГЛАВА 6. ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ

6.1 Расчет затраченных средств на разработку

Практически каждый ИТ-проект разрабатывается с целью получения прибыли, прямо или косвенно. Именно поэтому рассмотрим среднюю стоимость разработки данного приложения и способы его монетизации.

Наш программный продукт разрабатывался единолично и квалификация разработчика – fullstack. Средняя заработная плата начинающего разработчика такой специализации начинается от 60 тысяч рублей в месяц. На разработку при этом ушел приблизительно год, соответственно на разработку программного продукта затрачено приблизительно 720 тысяч рублей.

При этом следует учитывать ежемесячную аренду виртуальной машины, на которой будет развернут docker контейнер с серверной частью.

Согласно платформе Yandex Cloud, машина с публичным IP-адресом, SSD объемом 256 Гб, двумя ядрами платформы Intel Compute Lake и 4 Гб оперативной памяти будет стоить около 6000 рублей в месяц.

На рекламу приложения в различных социальных сетях и площадках стоит заложить еще около 100 тысяч рублей для набора пользователей.

Итого для минимального запуска приложения понадобится 826 тысяч рублей, без учета дальнейших ежемесячных трат.

6.2 Монетизация приложения

Самый удобный способ монетизировать приложение – предоставлять приложение по подписке для мастеров стоимостью 100 рублей в месяц. Соответственно, если приложением будут пользоваться хотя бы 1000 мастеров, то приносить ежемесячно они будут 100 тысяч рублей. Срок окупаемости такого приложения с учетом ежемесячной аренды виртуальной машины составит около 9 месяцев.

Дополнительный вариант монетизации – продвижение мастеров и их услуг. Разработка функционала для этого и примитивной панели администратора составит еще 1-2 месяца разработки, что равно 60-120 тысячам рублей дополнительно. Однако начинающим мастерам будут интересны предложения, которые увеличат их клиентскую базу за счет более частой рекомендации пользователям по цене 5 рублей в сутки или 75 рублей в месяц.

Предположим, что из 1000 мастеров, около 100 решили приобрести продвижение по 75 рублей в месяц. Прибыль, полученная таким образом, равна 7500 рублей в месяц и полностью покрывает ежемесячные расходы на аренду виртуальной машины, а в совокупности с ежемесячной подпиской способна сократить окупаемость на срок около месяца.

6.3 Выводы по главе

Таким образом в данной главе были рассчитаны примерные затраты на разработку нашего приложения и срок окупаемости. При внедрении подписки за публикацию услуг мастеров в нашем приложении в размере 100 рублей в месяц и активном количестве публикующих около 1000 человек срок окупаемости может составить около 9 месяцев. Более того, при внедрении дополнительных услуг, таких как продвижение, можно сократить окупаемость практически на месяц.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Прохорова М. А. О ВОЗМОЖНОСТЯХ CRM - СИСТЕМ ДЛЯ МАЛОГО БИЗНЕСА // Экономика и социум. 2016. №4-2 (23). URL: https://cyberleninka.ru/article/n/o-vozmozhnostyah-crm-sistem-dlya-malogo-biznesa (дата обращения: 26.10.2024).
2. YCLIENTS — онлайн-запись и автоматизация | CRM-система для салонов красоты и других типов бизнеса. — Текст : электронный // YCLIENTS : [сайт]. — URL: https://www.yclients.com/ (дата обращения: 26.10.2024).
3. Магазин интеграций партнерских сервисов YCLIENTS. — Текст : электронный // YCLIENTS : [сайт]. — URL: https://www.yclients.com/marketplace/overview (дата обращения: 26.10.2024).
4. SimplyBook.me - бесплатная система предварительной записи. — Текст : электронный // SimplyBook : [сайт]. — URL: https://simplybook.me/ru/ (дата обращения: 26.10.2024).
5. DIKIDI - сервис онлайн записи клиентов, автоматизация и продвижение в сфере услуг. — Текст : электронный // DIKIDI : [сайт]. — URL: https://dikidi.ru/ (дата обращения: 26.10.2024).
6. Rient — онлайн-запись клиентов + CRM. — Текст : электронный // Rient : [сайт]. — URL: https://rient.ru/ (дата обращения: 26.10.2024).
7. Черкашин, П. А. Стратегия управления взаимоотношениями с клиентами (CRM) : учебное пособие / П. А. Черкашин. — 2-е изд. — Москва : ИНТУИТ, 2016. — 420 с. — ISBN 978-5-94774-643-3. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/100766 (дата обращения: 26.10.2024). — Режим доступа: для авториз. пользователей.
8. Лучшие частные мастера красоты России. — Текст : электронный // moiprofi : [сайт]. — URL: https://moiprofi.ru/ (дата обращения: 26.10.2024).
9. Мобильные ОС в России | Яндекс.Радар. — Текст : электронный // Яндекс.Радар : [сайт]. — URL: https://radar.yandex.ru/mobile?period=all&group=quarter (дата обращения: 15.11.2024).
10. Колчанова Светлана Анатольевна СРАВНЕНИЕ ФРЕЙМВОРКОВ FLUTTER И REACT NATIVE, ИСПОЛЬЗУЕМЫХ В РАЗРАБОТКЕ ГИБРИДНЫХ ПРИЛОЖЕНИЙ // E-Scio. 2022. №4 (67). URL: https://cyberleninka.ru/article/n/sravnenie-freymvorkov-flutter-i-react-native-ispolzuemyh-v-razrabotke-gibridnyh-prilozheniy (дата обращения: 26.11.2024).
11. Шилдт, Г. Java : Полное руководство / Шилдт Герберт. — 12-е изд. : Пер. с англ. — СПб : ООО "Диалектика", 2023. — 1344 c. — Текст : непосредственный (дата обращения: 26.11.2024).
12. Здитовец А. Л. ОСНОВНЫЕ ТЕХНОЛОГИИ И ФРЕЙМВОРКИ ДЛЯ БЕКЕНД-РАЗРАБОТКИ НА JAVA // Инновационная наука. 2023. №8-2. URL: https://cyberleninka.ru/article/n/osnovnye-tehnologii-i-freymvorki-dlya-bekend-razrabotki-na-java (дата обращения: 26.11.2024).
13. Visual Studio vs Visual Studio Code - What to Choose in 2024?. — Текст : электронный // GeeksForGeeks : [сайт]. — URL: https://www.geeksforgeeks.org/visual-studio-vs-visual-studio-code/ (дата обращения: 01.12.2024).
14. Eclipse Vs IntelliJ IDEA Vs NetBeans for Java Development. — Текст : электронный // GeeksForGeeks : [сайт]. — URL: https://www.geeksforgeeks.org/eclipse-vs-intellij-idea-vs-netbeans-for-java-development/ (дата обращения: 01.12.2024).
15. PostgreSQL: The World's Most Advanced Open Source Relational Database. — Текст : электронный // PostgreSQL : [сайт]. — URL: https://www.postgresql.org/ (дата обращения: 01.12.2024).
16. UML: обзор основных типов диаграмм, диаграмма Классов. Часть 1. — Текст : электронный // Хабр : [сайт]. — URL: https://habr.com/ru/articles/738428/ (дата обращения: 12.12.2024).
17. Использование диаграммы вариантов использования UML при проектировании программного обеспечения. — Текст : электронный // Хабр : [сайт]. — URL: https://habr.com/ru/articles/566218/ (дата обращения: 12.12.2024).
18. Class Diagram | Unified Modeling Language (UML). — Текст : электронный // GeeksforGeeks : [сайт]. — URL: https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/ (дата обращения: 12.12.2024).
19. Component Based Diagram – Unified Modeling Language (UML). — Текст : электронный // GeeksforGeeks : [сайт]. — URL: https://www.geeksforgeeks.org/component-based-diagram/ (дата обращения: 12.12.2024).
20. UML: обзор основных типов диаграмм, диаграмма компонентов. Часть 2. — Текст : электронный // Хабр : [сайт]. — URL: https://habr.com/ru/articles/756552/ (дата обращения: 12.12.2024).
21. Deployment Diagram in Unified Modeling Language(UML). — Текст : электронный // GeeksforGeeks : [сайт]. — URL: https://www.geeksforgeeks.org/deployment-diagram-unified-modeling-languageuml/ (дата обращения: 12.12.2024).
22. Tabs - Expo Documentation. — Текст : электронный // Expo Docs : [сайт]. — URL: https://docs.expo.dev/router/advanced/tabs/ (дата обращения: 12.01.2025).
23. Bonocore, G. Hands-On Software Architecture with Java / G. Bonocore. — 1st Edition. — Birmingham : Packt Publishing Ltd., 2022. — 510 c. — Текст : непосредственный (дата обращения: 12.01.2025).
24. The Websocket Protocol | Websocket.org. — Текст : электронный // Websocket.org : [сайт]. — URL: https://websocket.org/guides/websocket-protocol/ (дата обращения: 10.02.2025).
25. STOMP Support :: Spring Integration. — Текст : электронный // Docs Spring : [сайт]. — URL: https://docs.spring.io/spring-integration/reference/stomp.html (дата обращения: 10.02.2025).
26. Project Lombok. — Текст : электронный // Project Lombok : [сайт]. — URL: https://projectlombok.org/ (дата обращения: 18.02.2025).
27. @Log (and friends). — Текст : электронный // Project Lombok : [сайт]. — URL: https://projectlombok.org/features/log (дата обращения: 18.02.2025).
28. Javadoc Tool Home Page. — Текст : электронный // Oracle : [сайт]. — URL: https://www.oracle.com/java/technologies/javase/javadoc.html (дата обращения: 18.02.2025).
29. Винтерингем М. Тестирование веб-API / М. Винтерингем — СПб.: Питер, 2024. — 304 с. — Текст: непосредственный (дата обращения: 22.02.2025).
30. Overview (JUnit 5.12.1 API). — Текст : электронный // JUnit : [сайт]. — URL: https://junit.org/junit5/docs/current/api/ (дата обращения: 22.02.2025).
31. mockito-core 5.16.0 javadoc (org.mockito). — Текст : электронный // Mockito Core JavaDoc : [сайт]. — URL: https://javadoc.io/doc/org.mockito/mockito-core/latest/org.mockito/module-summary.html (дата обращения: 22.02.2025).
32. Postman: The World's Leading API Platform. — Текст : электронный // Postman : [сайт]. — URL: https://www.postman.com/ (дата обращения: 22.02.2025).

ПРИЛОЖЕНИЕ А

**Метод обновления статуса записи в сервисе записей**

@Override  
public Record updateRecordStatus(UUID recordId, String newStatus) {  
 *log*.info("updateRecordStatus[1]: Searching for record with id: {}", recordId);  
 Record record = recordRepository.findById(recordId)  
 .orElseThrow(() -> new ResourceNotFoundException("Record", "id", recordId));  
   
 // Проверка на отмену записи  
 if (newStatus.equals("CANCELLED")) {  
 User currentUser = userService.getCurrentUser();  
 LocalDateTime now = LocalDateTime.*now*();  
   
 // Если отменяет клиент  
 if (currentUser.getRole() == Role.*ROLE\_CLIENT*) {  
 // Проверка, что до записи осталось больше 24 часов  
 if (record.getRecordDate().isBefore(now.plusHours(24))) {  
 throw new ApiException(HttpStatus.*FORBIDDEN*, "Клиент может отменить запись только за 24 часа до начала");  
 }  
 // Отправка уведомления мастеру  
 notificationService.sendRecordCancellationNotification(  
 record.getMaster().getId(),  
 "Клиент отменил запись",  
 String.*format*("Клиент %s отменил запись на %s",   
 record.getClient().getFirstName(), record.getClient().getLastName(),  
 record.getRecordDate().format(DateTimeFormatter.*ofPattern*("dd.MM.yyyy HH:mm"))  
 )  
 );  
 }   
 // Если отменяет мастер  
 else if (currentUser.getRole() == Role.*ROLE\_MASTER*) {  
 // Отправка уведомления клиенту  
 notificationService.sendRecordCancellationNotification(  
 record.getClient().getId(),  
 "Мастер отменил запись",  
 String.*format*("Мастер %s отменил вашу запись на %s",   
 record.getMaster().getFirstName(), record.getMaster().getLastName(),  
 record.getRecordDate().format(DateTimeFormatter.*ofPattern*("dd.MM.yyyy HH:mm"))  
 )  
 );  
 }  
 }  
  
 *log*.info("updateRecordStatus[2]: Updating status from {} to {}", record.getRecordStatus(), newStatus);  
 record.setRecordStatus(RecordStatus.*valueOf*(newStatus));  
   
 *log*.info("updateRecordStatus[3]: Saving updated record");  
 return recordRepository.save(record);  
}