МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждения образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Информационных систем и технологий

Специальность 1-98 01 03 «Программное обеспечение информационной безопасности мобильных систем»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКАКУРСОВОГО ПРОЕКТ:**

По дисциплине «Программирование Мобильных Систем»

Тема **«**Мобильное приложение авиакомпании»

**Исполнитель**

студент 3 курса 7 группы \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д. И. Лешук

подпись, дата

**Руководитель**

ассистент \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н. И. Уласевич

должность, учен. степень, ученое звание подпись, дата

Допущен(а) к защите \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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

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

дата, подпись

Курсовой проект защищен с оценкой

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_ Н. И. Уласевич

подпись дата инициалы и фамилия

**Содержание**

[ВВЕДЕНИЕ 5](#_Toc166867101)

[1. Постановка задачи 6](#_Toc166867102)

[2. Обзор аналогичных решений 7](#_Toc166867103)

[2.1 Анализ приложения Qatar Airways 7](#_Toc166867104)

[2.2 Анализ приложения Turkish Airlines Flight Ticket 12](#_Toc166867105)

[2.3 Выводы по разделу 15](#_Toc166867106)

[3. Проектирования приложения 16](#_Toc166867107)

[3.1 Проектирование базы данных 16](#_Toc166867108)

[3.2 Проектирование ролей 19](#_Toc166867109)

[3.3 Проектирование навигации 20](#_Toc166867110)

[3.4 Описание структуры проекта 21](#_Toc166867111)

[3.5 Выводы по разделу 22](#_Toc166867112)

[4. Реализация приложения 23](#_Toc166867113)

[4.1 База данных и взаимодействие с ней 23](#_Toc166867114)

[4.2 Авторизация, регистрация, личный кабинет пользователя 25](#_Toc166867115)

[4.3 Панель администратора 27](#_Toc166867116)

[4.4 Процесс поиска и бронирования билета 29](#_Toc166867117)

[4.5 Страница бронирований пользователя 32](#_Toc166867118)

[4.6 Выводы по разделу 33](#_Toc166867119)

[5. Тестирование приложения 34](#_Toc166867120)

[5.1 Гость 34](#_Toc166867121)

[5.2 Авторизация и бронирование билета 35](#_Toc166867122)

[5.3 Использование панели администратора 37](#_Toc166867123)

[5.4 Выводы по разделу 39](#_Toc166867124)

[6. Краткое описание приложения для демонстрации 40](#_Toc166867125)

[7. Руководство пользователя 41](#_Toc166867126)

[ЗАКЛЮЧЕНИЕ 44](#_Toc166867127)

[Список используемых источников 45](#_Toc166867128)

[Приложение А. Взаимодействие с базой данных 46](#_Toc166867129)

[Приложение Б. Виджеты страниц 59](#_Toc166867130)

# **ВВЕДЕНИЕ**

В современном мире авиационная индустрия играет ключевую роль в обеспечении глобальной связности и мобильности, а также является крупным сектором мировой экономики. В связи с этим возникает необходимость в эффективном управлении авиаперевозками и улучшении качества обслуживания. Рост рынка мобильных устройств привел к тому, что мобильные приложения стали неотъемлемой частью повседневной жизни, а большинство людей предпочитают использовать смартфоны и планшеты для выполнения различных задач. Пассажиры авиакомпаний также ожидают удобства при планировании своих поездок. В этом контексте разработка мобильного приложения для авиакомпаний на платформе Flutter становится критически важной задачей и для бизнеса.

Целью данного курсового проекта является создание мобильного приложения, которое позволит пассажирам удобно и быстро бронировать билеты, получать информацию о рейсах и управлять своими бронированиями. Проектирование приложения будет основано на принципах модульности, гибкости и адаптивности.

Использование такой технологии, как база данных позволит нам эффективно хранить и управлять большим объемом информации о полетах, пассажирах и билетах.

Адаптивная разметка, подходящая для большинства устройств, позволит охватить максимально возможное количество пользователей. Механизмы аутентификации и авторизации обеспечат нам доверие пользователей, управление доступом и безопасность данных.

В пояснительной записке будут рассмотрены основные этапы разработки мобильного приложения, включая анализ требований и аналогичных решений, проектирование приложения, реализация и тестирование приложения, а также руководство пользователя.

1. **Постановка задачи**

Для разработки мобильного приложения на платформе Flutter для авиакомпании следует учесть следующие функциональные требования:

1. Авторизация и регистрация пользователей

Все пользователи должны быть разделены на роли – пользователь и администратор (или же менеджер). Пользователи в свою очередь будут делиться на 2 группы – гость и авторизованный пользователь (пользователь с учётной записью).

* гость может просматривать информацию о рейсах без возможности бронирования и выполнять регистрацию;
* авторизованный пользователь может бронировать билеты, просматривать статус своего бронирования и получать электронный билет.

1. Поиск рейса с условием

Пользователь может искать рейсы с определенными параметрами (например, дата, направление).

3.  Роль администратора (менеджера) подразумевает возможность выполнять следующие действия:

* управление рейсами (удалять, добавлять, изменять);
* добавлять новые транспортные узлы;
* управление данными иных таблиц базы данных.

В общем, его задача заключается в работе с данными системы – заполнение, обслуживание таблиц, обеспечение системы корректными данными с помощью интерактивных элементов приложения

1. Личный кабинет пользователя

Личный кабинет должен представлять собой отдельную страницу, на которой авторизированный пользователь сможет выполнять свои прямые функции, связанные с персональными данными либо бронированием.

1. Реализация информационного раздела по туризму и правилам авиаперевозок.

Информационный раздел – несколько отдельных страниц содержащие статические данные и изображения, которые носят информационный характер для пользователя.

6. Должна присутствовать функция генерации документа с данными о билете (электронный билет).

Также следует отметить, что приложение должно быть кросс-платформенным на Flutter на основе Widget с адаптивным интерфейсом и навигацией внутри приложения.

Все данные должны храниться в реляционной базе данных: должны быть спроектированы таблицы и их ограничения целостности а также обоснована их необходимость.

Необходимо сгенерировать apk-файл для распространения приложения на платформе Android. Перед этим шагом работа приложения должна быть полность. протестирована.

# **2. Обзор аналогичных решений**

В данном разделе рассмотрим функционал мобильных приложений авиакомпаний, которые выполняют аналогичные пользовательские функции, которые необходимо разработать в рамках данного курсового проекта. Первым аналогом было выбрано приложение QatarAirways, вторым – Turkish Airlines Flight Ticket. Оба приложения были протестированы на устройстве Xiaomi Redmi Note 7, Android 10 Q.

* 1. **Анализ приложения Qatar Airways**

При запуске приложения нас встречает приветственный слайдер, в котором проиллюстрированы и описаны основные функции приложения – рисунок 2.1

****

Рисунок 2.1 – Приветственный слайдер

Далее нас встречает главная страница приложения – слева на рисунке 2.2..На ней мы видим панель навигации снизу, нашу текущую вкладку, бургер-меню слева сверху, пару текстовых блоков. В центре внимания тут – слайдер (или же карусель). Основной упор на этой странице делается на продвижение какого-либо места для путешествия на основании местоположения (можно изменить страну вылета).. Также мы видим, что остальные блоки на некоторый процент показаны на экране, чтобы пользователь понимал, что этот элемент можно пролистать.

Ещё стоти обратить внимание, что приложение сразу адаптируется под системную тему (на устройстве стоит тёмная тема).

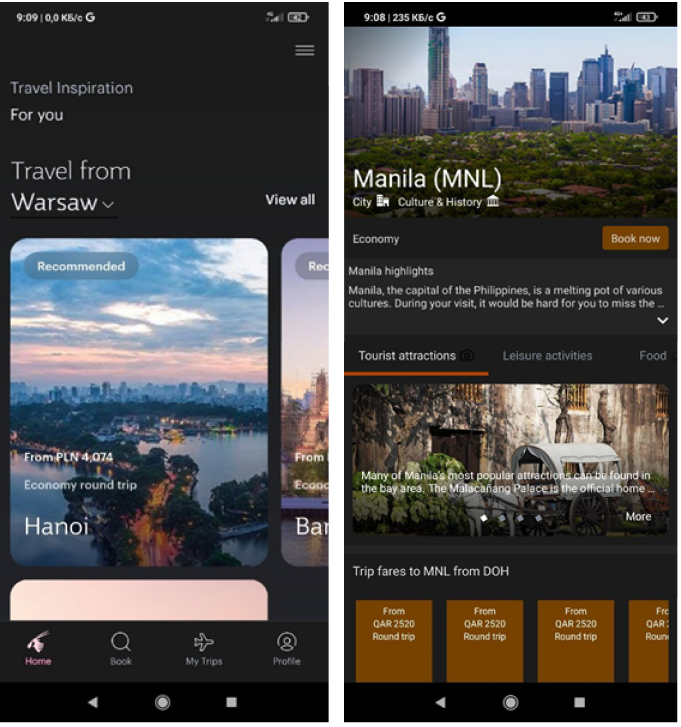


Рисунок 2.2 – Главная страница приложения

При нажатии на блок карусели открывается тематическая страница этого места со всей сводной информацией – показано справа на рисунок 2.2. Отметим, что на ней довольно много интерактивных элементов.

При переходе на страницу «Book» через панель навигации открывается следующая страница с формой на весь экран, где предлагается выбрать следующие опции для поиска билетов: тип билета (в одну сторону, туда-обратно), аэропорт отправления, аэропорт прибытия, дата, количество пассажиров, класс, а также вариант ввода промокода – рисунок 2.3.

В текстбоксе выбора аэропорта имеются подсказки, которые меняются в зависимости от пользовательского ввода (по стране и по названию города), элемент datepicker здесь не стандартный, а специально стилизован под приложение и расположен на «выезжающем» элементе.

Как мы можем видеть на изображении, при активной тёмной теме один из элементов всё же остался светлым, а именно граница вокруг кнопки для поиска рейсов. Даже в самом проработанном приложении могут быть небольшие недочёты.

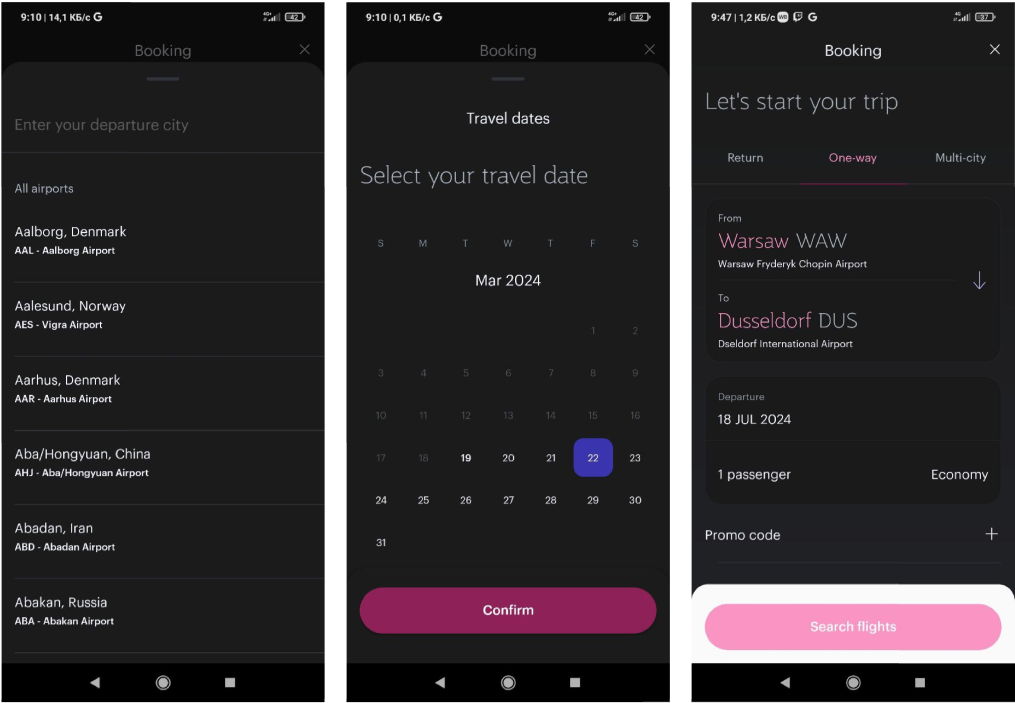


Рисунок 2.3 – Форма поиска билетов на определённый рейс и её элементы

После нажатия на кнопку «Search flights» открывается новая страница с результатами поиска рейсов – рисунок 2.4 Каждый рейс, удовлетворяющий условию представлен отдельным блоком, на котором расположена часть информации о рейсе. Также мы можем выбрать дату на один день позже и один день раньше от указанной, что может упростить процесс поиска билетов на рейс, если на нужный день активных рейсов нету (не запланированы или нету свободных мест).

При нажатии на блок открывается страница подтверждения, что пользователь хочет выбрать именно этот рейс. Она содержит всю ту же информацию, что и на блоке рейса из прошлой странице, а также кнопку, которая выполняет переход на новую стадию процесса бронирования.

Заметим, что только на этом моменте мы увидели логотип компании в данном приложении

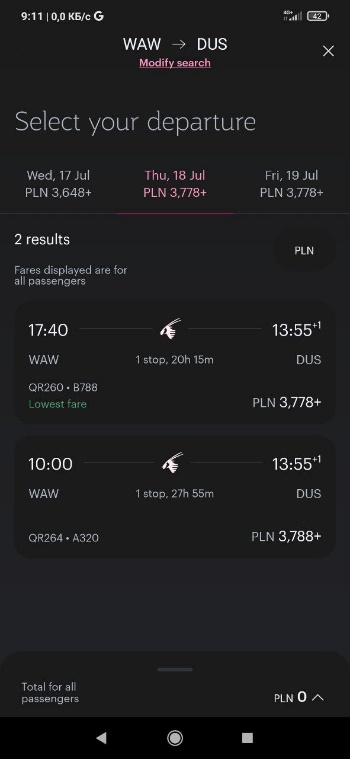


Рисунок 2.4 – Результат поиска билетов

Далее нам предлагает выполнить вход, чтобы сэкономить время, либо при каждом бронировании выполнять ввод данных о пассажире вручную – рисунок 2.5.

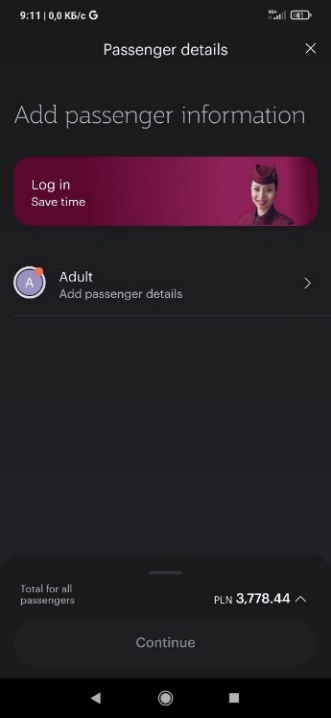


Рисунок 2.5 – Добавление информации о пассажире

Бронирование возможно и без учётной записи, что будет удобно пользователю, который единожды воспользуется услугами данной компании.

После нажатия на блок с информацией о пассажире во всплывающем окне появляется довольно простая форма с различными полями ввода.

На этапе всех действий пользователь может отменить текущее действие с помощью нажатия на иконку креста в правом верхнем экране. При нажатии на него появляется всплывающее окно с подтверждением действия.

Так как наше приложение не ставит перед собой цель обеспечения оплаты билетов и взаимодействия с банковскими счетами, не будем рассматривать дальнейшие действия. Вернёмся к основным страницам, а осталась из них только вкладка «Profile», на которой мы можем выполнить вход или зарегистрироваться – рисунок 2.6.

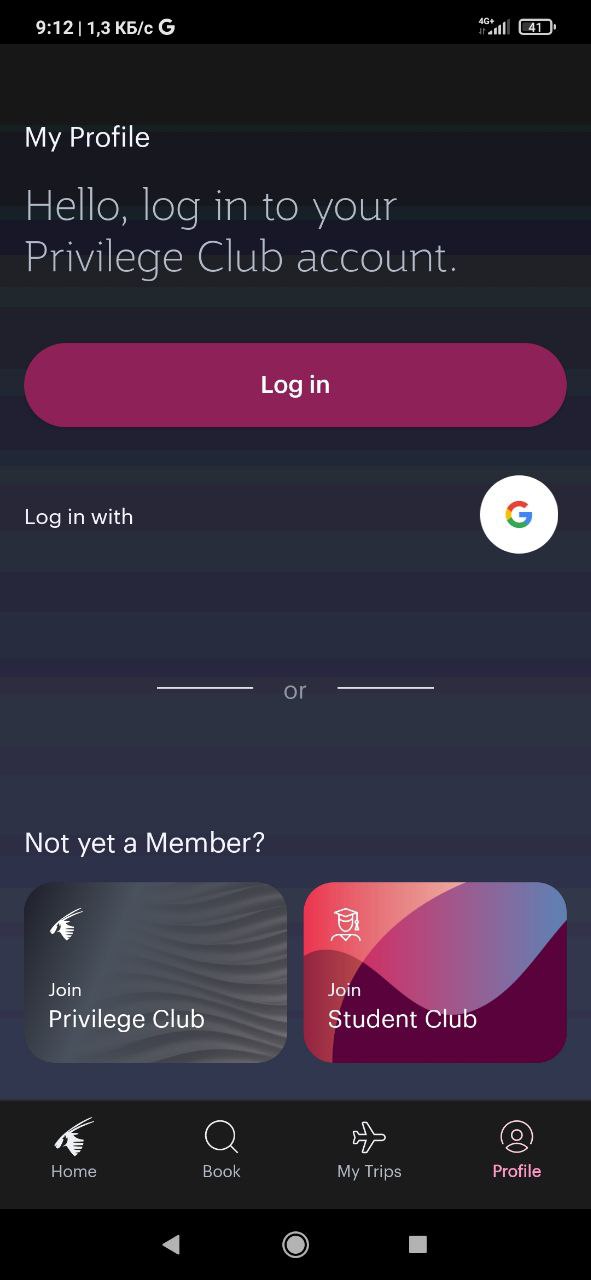


Рисунок 2.6 – Страница регистрации и авторизации

Стоит отметить, что привычное создание учётной записи тут заменено «вступлением в клуб привилегий», что обеспечивает пользователя какими-то бонусами. Этот этап включает в себя последовательное заполнение нескольких форм персональными данными, на этой же странице расположен элемент progress bar, по которому пользователь может ориентироваться в стадии регистрации. Удобным также является, что мы можем войти через учётную запись Google.

Бургер-меню, которое было упомянуто выше, в основном содержит ссылки на вебсайт авиакомпании, а также отвечает за настройку уведомлений, темы и других сопутствующих параметров.

* 1. **Анализ приложения Turkish Airlines Flight Ticket**

Здесь нас также встречает приветственный слайдер, как и в первом приложении, отличие лишь в том, что здесь нам сразу предоставляют выбор – зайти как гость, или выполнить вход. Главная страница у данного приложения сильно отличается от первого аналога – рисунок 2.7.

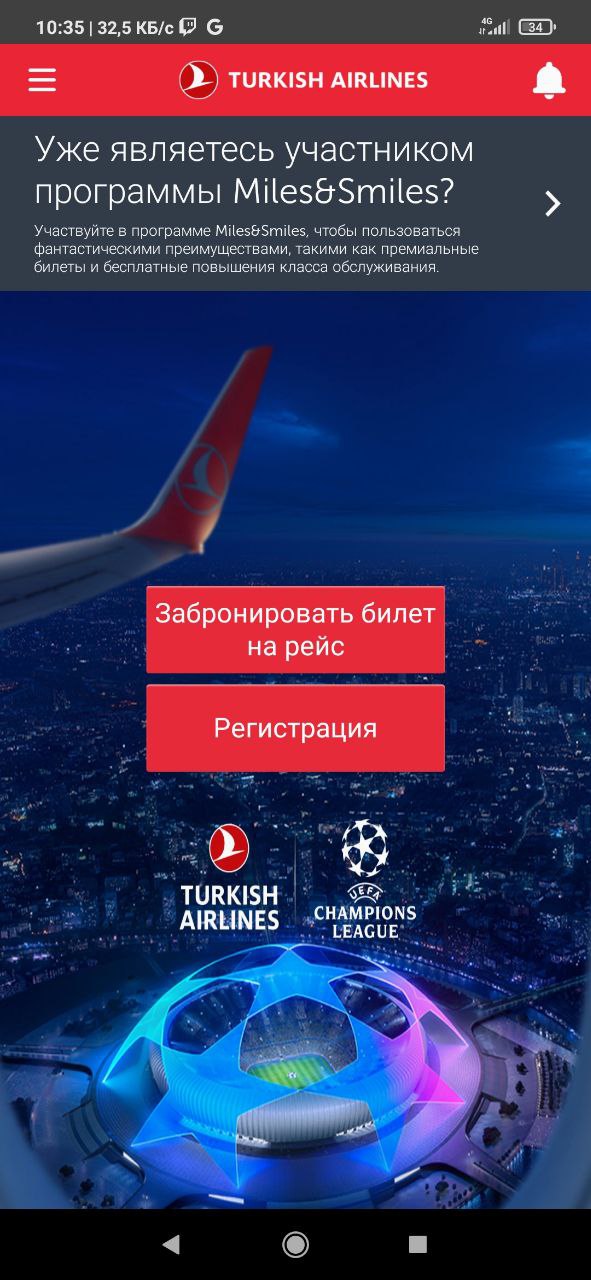


Рисунок 2.7 – Главная страница приложения Turkish Airlines Flight Ticket

Функционала здесь, скажем, не так много – можем лишь зарегистрироваться и перейти к брони. Также в верхней панели присутствует раздел уведомлений и меню, которое содержит в себе довольно много пунктов, которые открывают прямо в приложении страницы с веб-ресурса авиакомпании, большинство из них носит информационный характер, например «Развлечения на борту», «Праздники Turkish Airlines», «Открытия» и другие.

На странице поска рейсов присутствует несколько секций, которые полностью совпадают с Qatar Airways (стандартный набор условий для поиска рейса) – рисунок 2.8.

Отметим, что при заполнении одного условия автоматически открывается для выбора другое, и у рассмотренного ранее аналога такой функции нету.

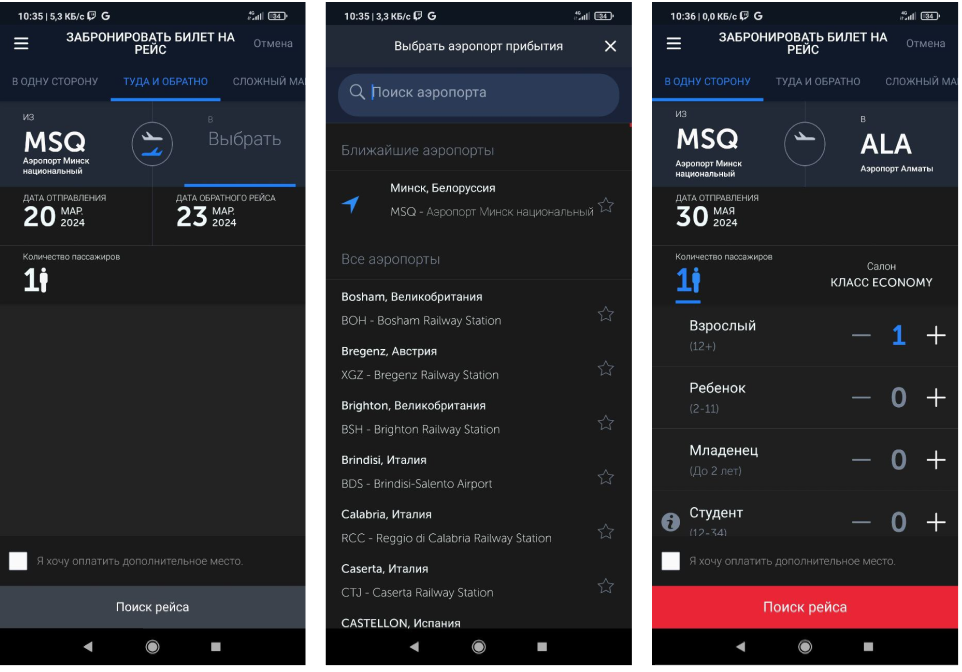


Рисунок 2.8 – Заполнение условий поиска рейса и элементы формы

Далее открывается страница с найденными рейсами – рисунок 2.9.

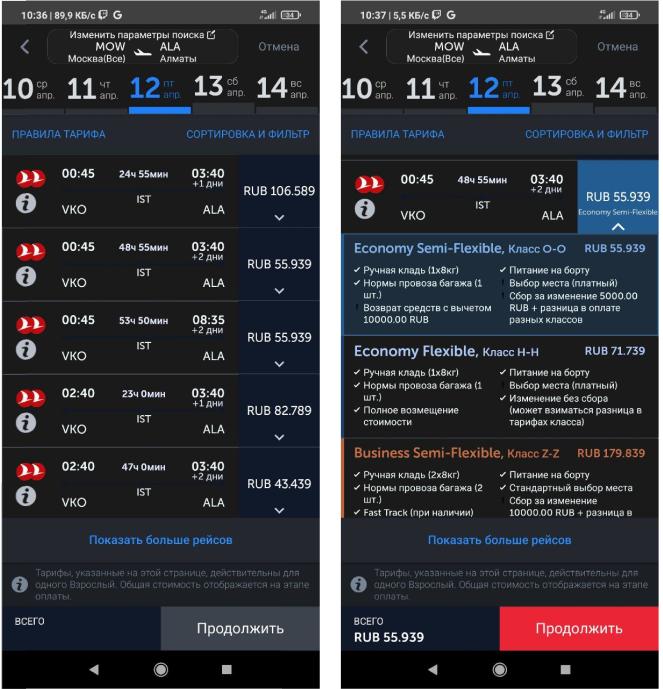


Рисунок 2.9 – Результат поиска

Здесь мы можем переключиться на результаты на 2 дня вперёд и назад, что довольно удобно. Ниже довольно плотно без отступов расположены информационные блоки с описанием рейсов, при нажатии на которые открывается меню выбора класса, кнопка продолжить становится активной. Для продолжения бронирования необходимо ввести персональные данные в появившуюся форму – рисунок 2.10.

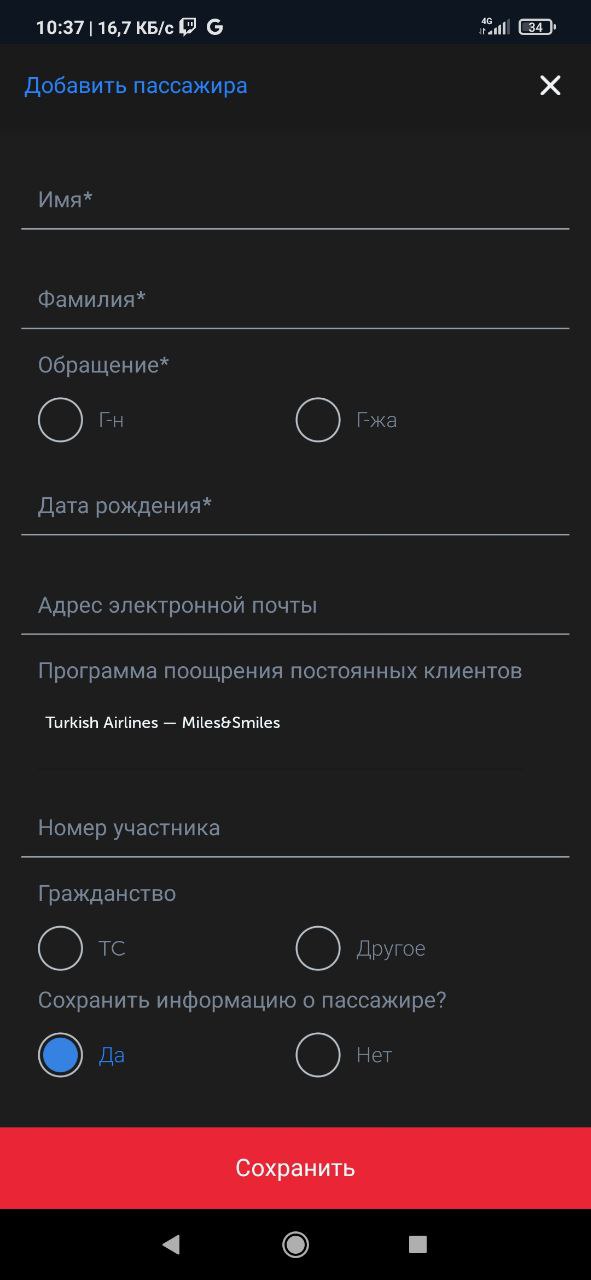


Рисунок 2.10 – Форма добавления пассажира

В это форме мы вводим типичные персональные данные пассажира для бронирования билета.

Вернёмся к окну регистрации – рисунок 2.11. Данная форма выполняет несколько функций – переход к регистрации, авторизация пользователя по несколькими способами, а также сброс пароля.

Как и у прошлого аналогичного решения, почти на каждой странице при выполнении действия справа сверху есть кнопка «Отмена», которая после подтверждения вернёт пользователя на главную страницу.

Несмотря на то, что у этой формы довольно простое оформление, у неё очень широкий функционал

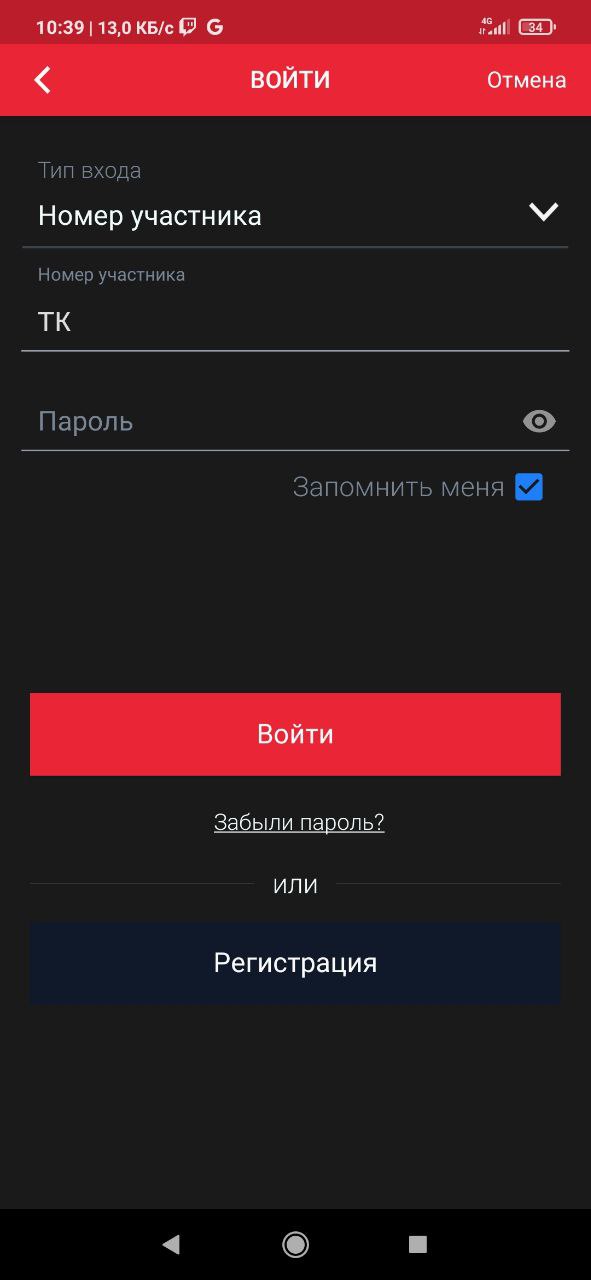


Рисунок 2.11 – Форма авторизации

Данное приложение нацелен непосредственно на продажу билетов, а не ведение туристической деятельности, вследствие чего его функционал ограничен. Но тем не менее, команда разработчиков, которая трудилась над этим приложением, успешно достигла своих целей. Как минимум, это приложение поддерживает несколько языков интерфейса, в отличие от Qatar Airways, где по умолчанию и единственный язык – английский.

* 1. **Выводы по разделу**

Проанализированные приложения полностью реализуют все функции, рассмотренным в постановке задач. В обоих приложениях используются элементы навигации, механизмы авторизации и регистрации, поиска рейсов с заданными условиями. В обоих приложениях качественно проработана разметка.

В приложении должен быть реализован элемент навигации, должны быть разработаны все необходимые формы: регистрации, авторизации, поиска рейса, бронирования билета на рейсы.

Необходимо внедрить интуитивно понятный интерфейс, современный дизайн.

1. **Проектирования приложения**

Раздел проектирования приложения будет рассматривать вопросы создания таблиц (и содержащихся в них столбцах с ограничениями целостности), общую структуру проекта (модули, библиотеки и другие зависимости), а также схему навигации в приложении. Данный раздел отражает весь пройденный путь проектирования программного средства а также взаимодействие с выбранными технологиями для обеспечения функциональных требований продукта.

* 1. **Проектирование базы данных**

Отобразим сущности и связи между ними на логической схеме базы данных, которая представлена на рисунке 3.1.

Все таблицы были спроектированы с учётом необходимых функциональных требований а также с соблюдением принципов нормализации.

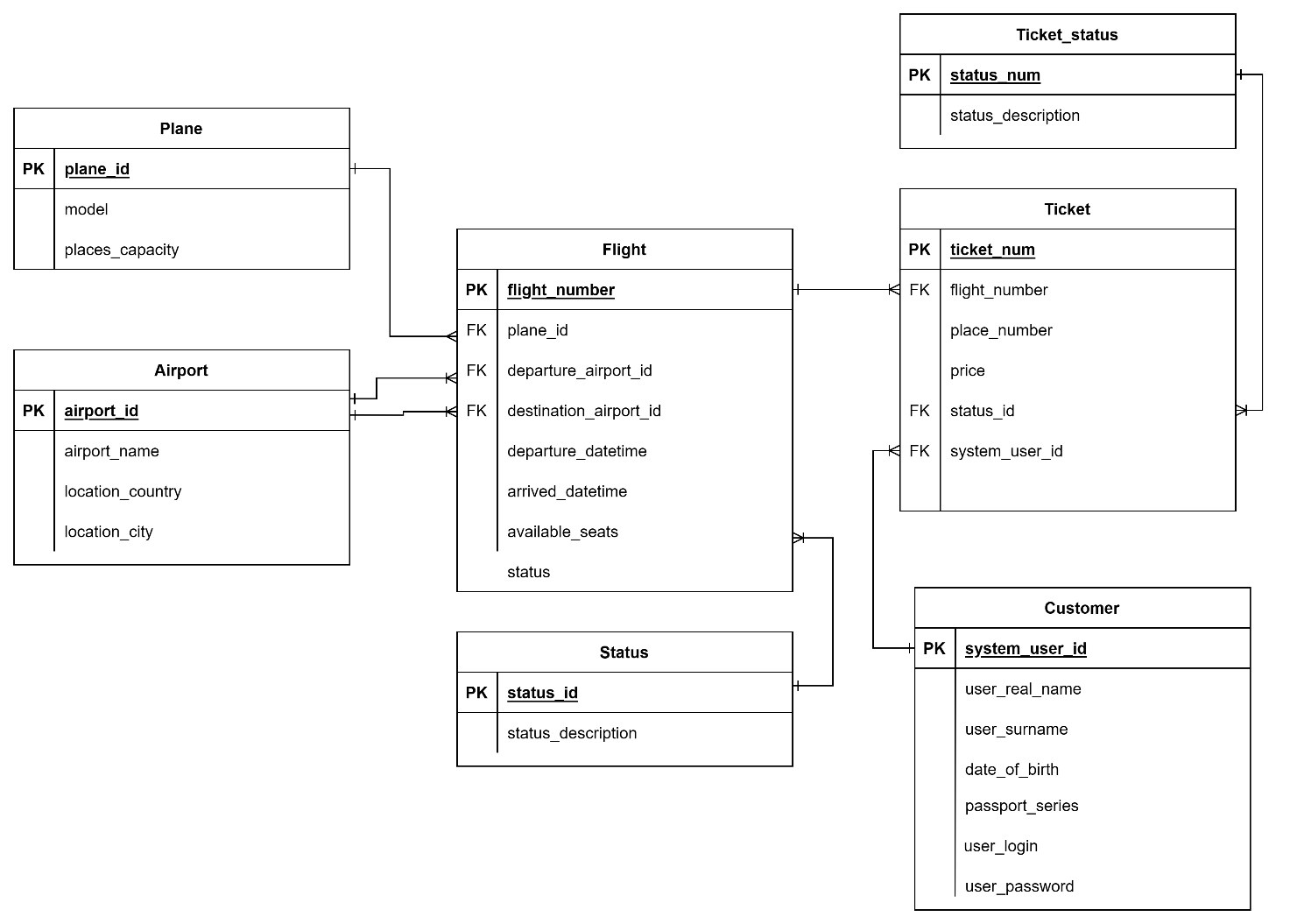


Рисунок 3.1 – Логическая схема базы данных

Таблицы Status и Ticket\_status реализуют логику перечисления. Первичным ключом в каждой таблице выступает уникальный идентификатор записи таблицы.

В качестве СУБД выберем SQLite – самую используемую СУБД для мобильных устройств за счёт своей простоты и легковесности.

Опишем все таблицы, перейдя к физической реализации – типы данных а также описание того, что будет храниться в конкретном столбце – таблицы 3.1 – 3.7. Первая таблица – Аэропорт, будет хранить в себе все самые необходимые данные об аэропортах.

Таблица 3.1 – Содержание таблицы Airport (Аэропорт)

|  |  |  |
| --- | --- | --- |
| Название | Тип данных | Описание |
| airport\_id | INTEGER | Уникальный идентификатор аэропорта |
| airport\_name | TEXT | Название аэропорта |
| location\_country | TEXT | Название страны расположения |
| location\_city | TEXT | Название города расположения |

Таблица Plane нам нужна в основном для того, чтобы определять количество мест на данном рейсе у данного самолёта, так как авиапарк компании будет ограниченным.

Таблица 3.2 – Содержание таблицы Plane (Самолёт)

|  |  |  |
| --- | --- | --- |
| Название | Тип данных | Описание |
| plane\_id | INTEGER | Уникальный идентификатор самолёта |
| model | TEXT | Модель самолёта |
| business\_seats\_capacity | INTEGER | Количество мест обычного класса |
| business\_seats\_capacity | INTEGER | Количество мест бизнес класса |

Далее нам нужна таблица статусов полёта, что бы мы могли удобно получать информацию о рейсе и логика наших последующих запросов значительно упростилась.

Таблица 3.3 – Содержание таблицы Status(Статус полёта)

|  |  |  |
| --- | --- | --- |
| Название | Тип данных | Описание |
| status\_id | INTEGER | Уникальный идентификатор статуса полёта |
| status\_description | TEXT | Описание статуса полёта |

Следующая таблица – одна из важнейших в проекте, она будет описывать сущность рейса. Мы должны включить в неё такие данные, как время вылета и прибытия, самолёт какой модели будет выполнять данный рейс, для вычисления количества мест.

Таблица 3.4 – Содержание таблицы Flight (Рейс)

|  |  |  |
| --- | --- | --- |
| Название | Тип данных | Описание |
| flight\_number | INTEGER | Уникальный идентификатор рейса |
| plane\_id | INTEGER | Уникальный идентификатор самолёта |
| departure\_airport\_id | INTEGER | Уникальный идентификатор самолёта отправления |
| destination\_airport\_id | INTEGER | Уникальный идентификатор самолёта прибытия |
| departure\_datetime | TEXT | Время отправления |
| arrive\_datetime | TEXT | Время прибытия |
| available\_seats | INTEGER | Текущее количество свободных мест |
| status | INTEGER | Уникальный идентификатор статуса |

Теперь переключимся на пользователей. Нам нужна таблица, которая будет в себе хранить данные о пользователях, прошедших регистрацию в нашем приложении.

Таблица 3.7 – Содержание таблицы Customer (Клиенты)

|  |  |  |
| --- | --- | --- |
| Название | Тип данных | Описание |
| system\_user\_id | INTEGER | Уникальный идентификатор пользователя |
| user\_real\_name | TEXT | Имя пользователя |
| user\_surname | TEXT | Фамлия пользователя |
| gender | TEXT | Гендер пользователя |
| date\_of\_birth | TEXT | Дата рождения пользователя |
| passport\_series | TEXT | Серия паспорта клиента |
| user\_email | TEXT | Электронная почта пользователя |
| user\_login | TEXT | Логин пользователя |
| user\_password | TEXT | Пароль пользователя (в хеше) |
| user\_role | TEXT | Роль пользователя в системе |

Для организации бронирования билета нам также нужна какая-то таблица, которая будет хранить в себе статус бронирования – забронирован билет, возвращён, использован.

Таблица 3.6 – Содержание таблицы Ticket\_status(Статус билета)

|  |  |  |
| --- | --- | --- |
| Таблица PK | Тип данных | Описание |
| Status\_num | INTEGER | Уникальный идентификатор статуса |
| status\_description | TEXT | Описание статуса |

Теперь опишем таблицу бронирований – Ticket.

Таблица 3.5 – Содержание таблицы Ticket (Билет)

|  |  |  |
| --- | --- | --- |
| Название | Тип данных | Описание |
| ticket\_num | NUMBER | Уникальный идентификатор билета |
| flight\_number | NUMBER | Уникальный идентификатор рейса |
| place\_number | NUMBER | Номер места |
| price | NUMBER | Стоимость билета |
| status\_id | NUMBER | Уникальный идентификатор статуса |
| system\_user\_id | NUMBER | Уникальный идентификатор клиента |
| ticket\_class | TEXT | Класс билета |

Теперь опишем ограничения целостности и другие особенности таблиц:

* на всех столбцах с идентификатором стоит autoincrement при создании;
* столбцу «user\_login» таблицы Customer присвоено ограничение целостности unique – у каждого пользователя логин должен быть уникальным;
* всем столбцам, не являющимися первичным ключом, назначено ограничение целостности not null – это поможет избежать «пустоты»;
* столбцу user\_role и gender таблицы Customer назначено ограничение целостности check. В столбце user\_role у нас может храниться только 2 значения ролей: пользователь и администратор (менеджер), в соответствии с условием. Аналогично – в gender будут храниться только 3 варианта значений – «Male», «Female», «Other». Данное ограничение будет использовать и столбец ticket\_class таблицы Ticket – будет 2 типа билетов: обычный и бизнес-класса.

Данные таблицы обеспечивают необходимый функционал, который требуется от программного средства.

Для организации базы данных приложения использовать пакет sqflite для flutter.

* 1. **Проектирование ролей**

Детальная информация о функциях каждой роли представлена на диаграмме использования на рисунке 1.1.

Кратко опишем данную схему: «Гость» (неавторизованный пользователь) сможет просматривать информационные элементы по правилам авиаперевозок и часто задаваемым вопросам, выполнять поиск рейса по интересующему направлению и на интересующую дату, а также регистрироваться в приложении. «Авторизованный пользователь» расширяет функционал гостя возможностью авторизации, бронирования билета и управления своим бронированием. «Менеджер» же в свою очередь обладает возможностью взаимодействия с данными таблиц – просматривать, обновлять и добавлять записи.

Во избежание исключительных ситуаций менеджер не будет наделяться привилегией удаления записей.

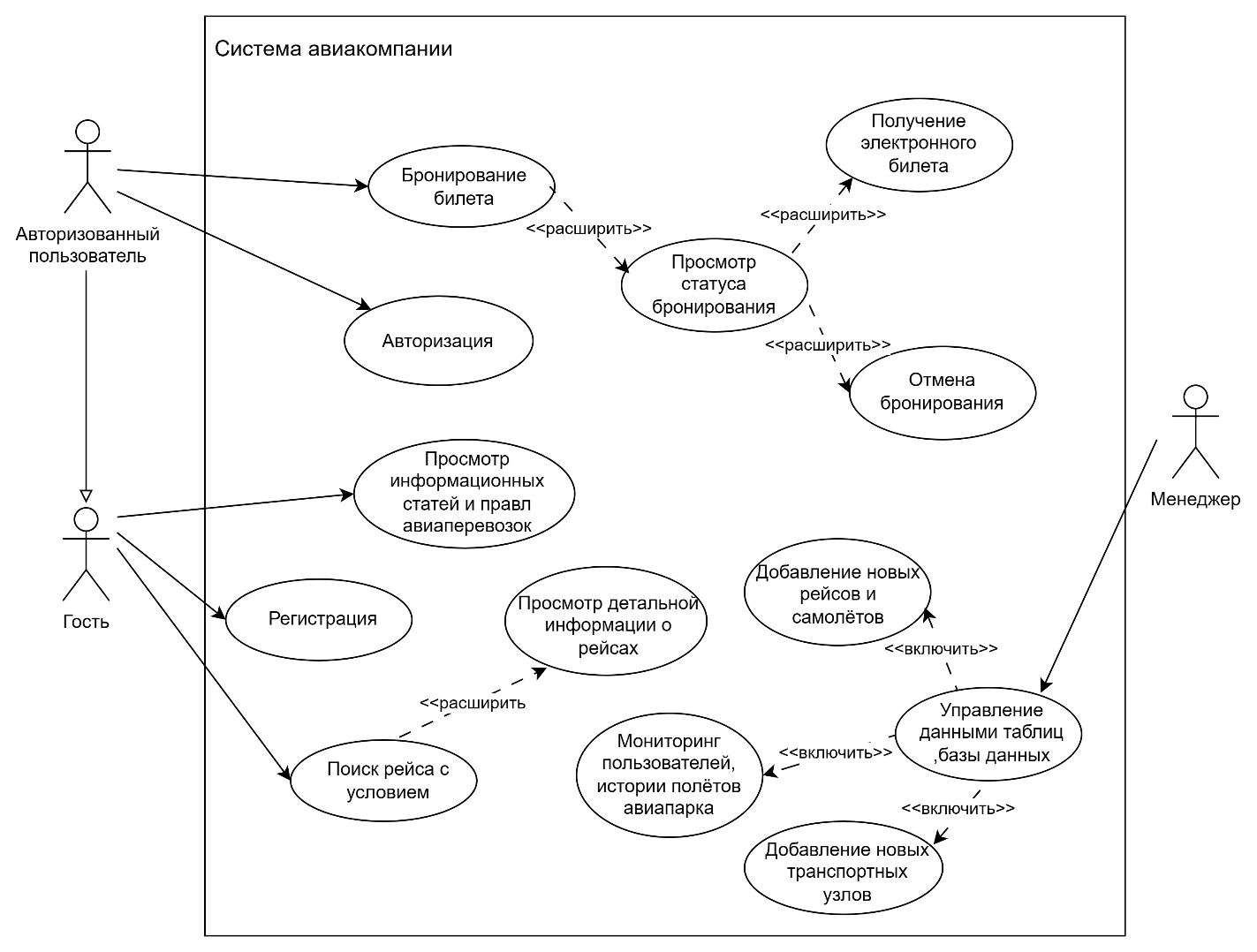


Рисунок 4.1 – Диаграмма вариантов использования

Данная диаграмма соответствует задачам, установленным в разделе постановки задач.

* 1. **Проектирование навигации**

Наше приложение будет содержать некоторое количество страниц, которые должны быть как-то связаны между собой. Опишем в общих чертах схему навигации – на главной странице у нас будет элемент с навигационным меню – BottomNavigationBar, каждая его вкладка представляет собой отдельную страницу. Следовательно, с главной страницы мы можем получить доступ к четырём основным страницам приложения. Страница профиля имеет условный рендер – если пользователь авторизован, отображаем страницу профиля, если нет – авторизации. Со страницы авторизации можно попасть на страницу регистрации. От страницы поиска рейса идёт цепочка страниц, которая реализует процесс бронирования билета на рейс с выбором сопутствующих параметров бронирования. Панель администратора будет доступна со страницы профиля после входа в учётную запись, с учётом того, что роль текущего пользователя – «admin».

* 1. **Описание структуры проекта**

Pubspec.yaml – файл с зависимостями проекта выглядит следующим образом – листинг 3.1. Пакет intl позволит нам работать с датой и временем, carousel\_slider предоставляет виджет слайдера с удобной реализацией. С помощь Path\_provider мы сможем предавать между контекстами страниц приложения наш единственный экземпляр класса, который будет обеспечивать взаимодействие с базой данных – databaseHelper.

dependencies:

flutter:

sdk: flutter

sqflite: any

intl: ^0.19.0

carousel\_slider: ^4.0.0

cupertino\_icons: ^1.0.6

path\_provider: ^2.0.2

shared\_preferences: ^2.0.6

provider: ^5.0.0

dev\_dependencies:

flutter\_test:

sdk: flutter

flutter\_lints: ^3.0.0

build\_runner: ^2.1.2

Листинг 3.1 – Зависимости проекта

Организация папки lib проекта представлена на рисунке 3.1. Она разделена на 2 части: первая – model, в которой содержатся классы, аналогичные таблицам базы данных для упрощения операций вставки, чтения, удаления и обновления, а также других вспомогательных классов, представляющих, например, данных для детализации рейса, так как там будут использоваться значения столбцов из некоторых таблиц и упрощения написания методов доступа к данным с использованием объектно-ориентированного подхода, но не будут перенимать недостатки технологии ORM. И вторая часть – страницы приложения, которые представляют собой описание UI-части и некоторой бизнес-логики, например динамическое создание виджетов или условный рендеринг.

Для обеспечения некоторых функций необходимо создание нескольких страниц, в том числе для улучшения читаемости кода.

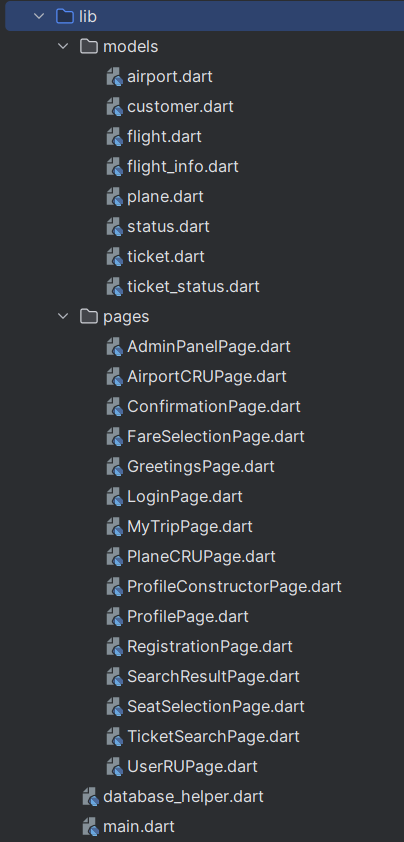


Рисунок 3.2 – Структура папки lib проекта

Файл main.dart определяет процесс инициализации приложения, а databaseHelper, как было описано выше, организует взаимодействие с базой данных.

* 1. **Выводы по разделу**

В данном разделе были описаны основные аспекты структуры проекта – модель навигации приложения с использованием схемы, физическая и логическая структура базы данных приложения, которая будет обеспечивать весь функционал приложения все необходимые зависимости проекта с обоснованием необходимости их использования, а также файловая структура проекта.

1. **Реализация приложения**

Данный раздел содержит в себе описание некоторых шагов разработки приложения и его элементов. Разработка приложения будет выполнена в IDE Android Studio, версии Hedgehod 2023.1.1 Patch 2, версия Flutter SDK – 3.19. Первоначальная реализация будет протестирована на Android-устройстве Google Pixel 3a.

* 1. **База данных и взаимодействие с ней**

Необходимо определить класс, который позволит нам создать базу данных для нашего приложения и взаимодействовать с ней – DatabaseHelper. Рассмотрим его основное содержимое – Листин 4.1. Полный скрипт данного класса представлен в приложении А.

class DatabaseHelper {

static final DatabaseHelper instance = DatabaseHelper.\_init();

static Database? \_database;

DatabaseHelper.\_init();

Future<Database> get database async {

if (\_database != null) return \_database!;

\_database = await \_initDB('mdair.db');

return \_database!;

}

Future<Database> \_initDB(String filePath) async {

final dbPath = await getDatabasesPath();

final path = join(dbPath, filePath);

return

Future<void> \_createDB(Database db, int version) async {

await db.execute('''CREATE TABLE Airport (<…>)

''');

await db.execute('''CREATE TABLE Flight\_status (<…>)

''');

await db.execute('''CREATE TABLE Plane (<…>)

''');

await db.execute('''CREATE TABLE Flight (<…>)

''');

await db.execute('''CREATE TABLE Ticket (<…>)

''');

…

}

Листинг 4.1 – Основа класса DatabaseHelper

В данном классе мы определяем поле instance, которое позволит нам создать только один экземпляр этого класса, метод get, который мы будем использовать в методах, в которых нам понадобится делать операции чтения, вставки, обновления, удаления данных из базы данных, а также в методе \_createDB описываем структуру и содержание наших таблиц. По мере необходимости данный класс будет дополняться методами доступа к данным и переменным, которые будут хранить промежуточные результаты, например, авторизованного пользователя или идентификатор выбранного рейса при бронировании.

Будем использовать экземпляр данного класса как глобальное состояние, чтобы передавать один и тот же экземпляр между разными страницами. Для этого воспользуемся пакетом Provider. Он позволит нам получать экземпляр класса DatabaseHelper из контекста виджета, предварительно поместив его в контекст приложения следующим способом – Листинг 4.2.

void main() {

runApp(

Provider(

create: (context) => DatabaseHelper.instance,

child: MyApp(),

),

);

}

…

late DatabaseHelper databaseHelper = Provider.of<DatabaseHelper> (context);

Листинг 4.2 – Настройка и использование класса Provider

Далее мы сможем выполнить любой метод, содержащийся в классе DatabaseHelper исходя из наших нужд. Но так как нам нужно передавать контекст какого-то виджета в параметры для получения данного класса, мы сможем вызывать его только в методе build виджета, представляющего нашу страницу. Однако, не во всех случаях такой способ нам подходит. На некоторых страницах нам нужно будет задать начальное состояние с помощью setState() или вовсе инициализировать переменную до построения виджета с помощью экземпляра класса DatabaseHelper. Вместо Provider в данных ситуациях мы просто будем передавать экземпляра класса DatabaseHelper как параметр конструктора страницы – это решит нашу проблему.

Также нам необходимо определить классы, которые будут представлять модель нашей базы данных. Их названия и наличие можно увидеть на рисунке 3.2, который был рассмотрен ранее. Пример представления класса модели можно найти в приложении А.

* 1. **Авторизация, регистрация, личный кабинет пользователя**

Авторизация, регистрация и личный кабинет призваны решить один вопрос – разделение ролей, следовательно и должны быть связаны между собой – располагаться в одном месте. В данном случае мы будем иметь дело с условным рендерингом – если пользователь не авторизован – отображать ему страницу входа, а если авторизован – личный кабинет. Решение будет несколько нетривиальное – Листинг 4.3.

class ProfileConstructorPage extends StatefulWidget {

@override

\_ProfileConstructorPageState createState() => \_ProfileConstructorPageState();}

class \_ProfileConstructorPageState extends State<ProfileConstructorPage> {

bool showFirstLayout = true;

late DatabaseHelper databaseHelper;

void toggleLayout() {

setState(() {

showFirstLayout = !showFirstLayout;

});}

@override

Widget build(BuildContext context) {

databaseHelper = Provider.of<DatabaseHelper>(context);

if(databaseHelper.getLoggedInUser() != null){

showFirstLayout = false;

}

return Scaffold(

appBar: AppBar(

title: Text('Your Profile'),

backgroundColor: Colors.yellow.shade400,

),

body: Center(

child: showFirstLayout

? LoginPage(toggleCallback: toggleLayout)

: ProfilePage(toggleCallback: toggleLayout),

),); }}

Листинг 4.3 – Страница условного рендеринга

Имеем страницу ProfileConstructorPage, на которой есть булевая переменная showFirstLayout, которая отвечает за то, отображать ли вместо себя страницу LoginPage с формой авторизации или же ProfilePage – личный кабинет пользователя. Переключение между ними достигается за счёт вызова callback-функции toggleLayout, которая обновляет состояние булевой переменной showFirstLayout и рендерит ту страницу, которая нам нужна в данный момент. Мы передаёт эту callback-функцию через конструктор на обе этих страницы, где вызываем их при определенных условиях: на странице авторизации – если введённые в TextField гостем логин и пароли от учётной записи соответствуют какому-либо пользователю в базе данных, на странице же профиля – при нажатии кнопки выхода из учётной записи.

Страница авторизации включает в себя 2 текстовых поля для ввода логина и пароля, кнопку для входа и кнопку перехода на страницу регистрации. Проверка данных от учётной записи проверяется следующим методом – листинг 4.3.

Future<Customer?> authenticateUser(String login, String password) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query(

'Customer',

where: 'user\_login = ? AND user\_password = ?',

whereArgs: [login, password], );

if (maps.isNotEmpty) {

final customerMap = maps.first;

final customer = Customer.fromMap(customerMap);

if(customer.userRole == "admin"){

\_isAdmin = true;

}

else{

\_isAdmin = false;

}

DatabaseHelper.instance.setLoggedInUser(customer);

return customer;

} else {

return null;

}

}

Листинг 4.3 – Метод аутентификации пользователя

Если метод возвращает какой-то результат, то этот объект фиксируется в DatabaseHelper для дальнейшего взаимодейтсвия. Если метод не возвращает никаких данных – делаем вывод, что пользователя с таким логином и паролем нету.

Страница регистрации, RegistrationPage, включает в себя довольно объёмную форму – поля ввода для заполнения данных для каждой строки в соответствии со схемой таблицы Customer, а также кнопку подтверждения регистрации и отмены регистрации и возвращения на страницу авторизации.

На странице профиля пользователь может редактировать данные, введённые им в процессе регистрации, например, изменить электронную почту, пароль или другие данные, если они были указаны ошибочно, а также выйти из своей учётной записи.

* 1. **Панель администратора**

На странице профиля также присутствует условный рендеринг – листинг 4.4.

if (!databaseHelper.isAdmin())

SizedBox(height: 10,),

ElevatedButton(

style: ElevatedButton.styleFrom(

padding: EdgeInsets.symmetric(vertical: 10),

minimumSize: Size(double.infinity, 50),

backgroundColor: Color.fromRGBO(0, 0, 0, 0.8)

),

onPressed: () {

Navigator.push(

context,

MaterialPageRoute(builder: (context) =>

AdminPanelPage()),

);

},

child: Text('Open Admin Panel',

style: TextStyle(color:Colors.yellow,fontSize: 18),), )

Листинг 4.4 – Условный рендеринг на странице профиля

В соответствии от того, выполнен ли вход под пользователем с ролью обычного пользователя или же администратора будет отрисована кнопка перехода на страницу с панелью администратора – AdminPanelPageфв . Выглядит она следующим образом – рисунок 4.1. Как мы видим, на ней есть навигационная панель BottomNavigationBar, где каждая вкладка – одна из сущностей нашей базы данных. На каждой из этих вкладок есть TabBar в верхней части страницы, в котором каждая вкладка – определённое действие: Retrive, Insert, Update. Функция удаления записей специально не была реализована с целью обеспечения целостности данных таблиц – чтобы невозможно было удалить записи, которые связаны ограничением внешнего ключа с другой таблицей, что может привести к серьёзным проблемам и необходимости решения этой исключительной ситуации.

На вкладке Retrieve каждой записи можно увидеть панель фильтров – с её помощью мы сможем выполнить поиск по значению определённого поля – идентификатору, названию, фамилии и т.д. интересующего нас объекта.

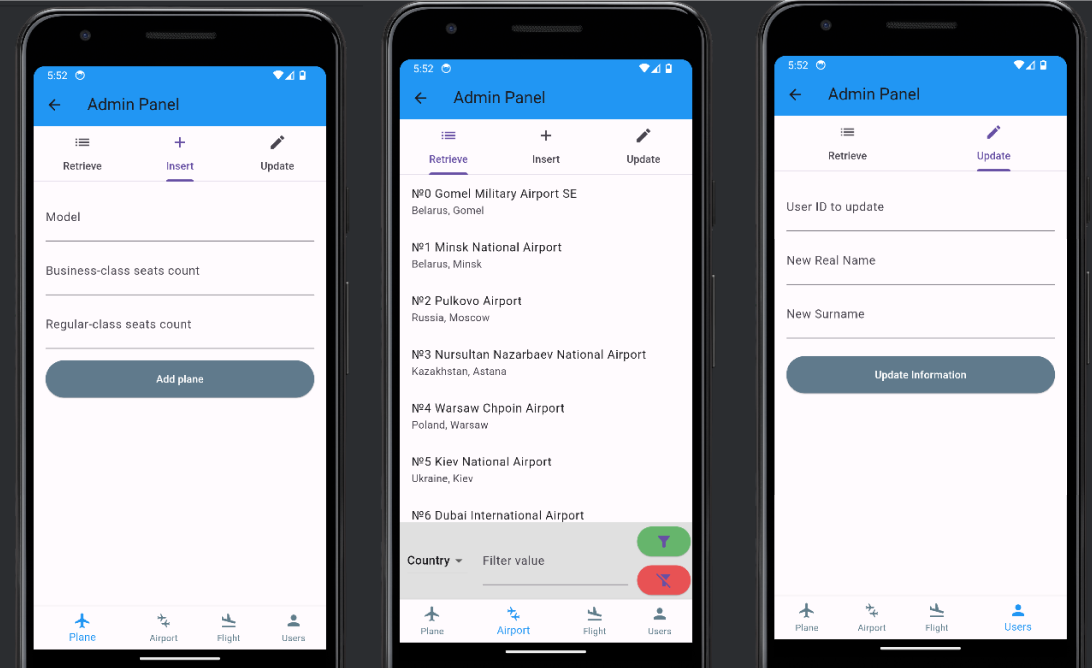


Рисунок 4.1 – Панель администратора

Также здесь присутствует функция быстрого выбора записи для обновления – достаточно нажать на интересующую запись на вкладке извлечения записей, после чего автоматически выполнится переход на вкладку обновления, а значения textField заполнятся данными этого объекта автоматически – остаётся только подкорректировать нужные значения полей и нажать на кнопку обновления записи.

Данная панель администратора полностью реализует функционал роли «Администратор», рассмотренной разделе постановки задач пояснительной записки и соответствует функционалу, описанному на диаграмме вариантов использования.

* 1. **Процесс поиска и бронирования билета**

В данном процессе будет задействовано несколько страниц нашего приложения, первая из них – TicketSearchPage – рисунок 4.2, на ней будет расположена форма поиска, в которую пользователь будет вводить желаемое место отправления и прибытия, а также дату отправления и предпочитаемый класс полёта.

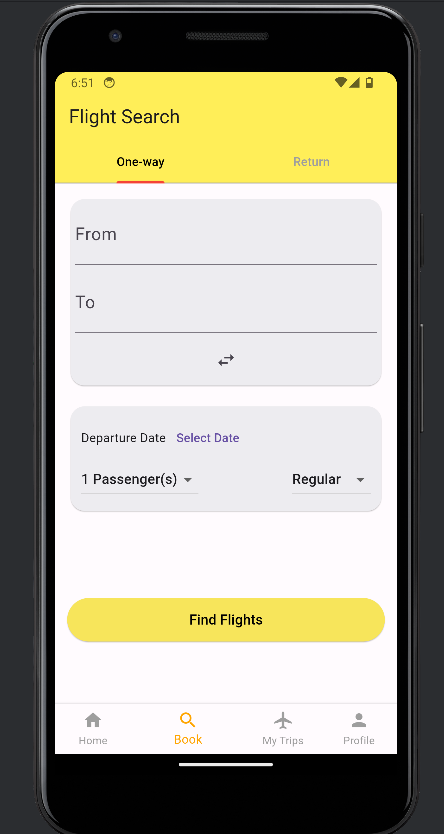


Рисунок 4.2 – Страница поиска

По нажатию на кнопку «Find Flights» будет выполнен метод класса DatabaseHelper, который проведёт выборку из таблиц «Flights», «Airports», «Planes»>, а также будет инициирован переход на следующую страницу. SearchResultPage, на ней будут отображены результаты поиска на три дня – указанный пользователем, на день раньше и на день позже. Построение будет происходить динамически – на основании результата поиска рейсов с необходимыми условиями в виджете ListView, каждый элемент которого будет представлен независимым виджетом без состояния – Листинг 4.5.

Padding(

padding: const EdgeInsets.symmetric(horizontal: 5),

child: ListView.builder(

itemCount: \_flightInfoLists[\_currentIndex].length,

itemBuilder: (context, index) {

return FlightResultTile(flightInfo: \_flightInfoLists[\_currentIndex][index],passengerCount: widget.\_passengerCount);

},

),

),

Листинг 4.5 – Построение списка с результатами поиска

При нажатии на кнопку «Book» на карточке с результатом поиска выполнится переход на страницу выбора класса полёта. FareSelectionPage – страница выбора класса полёта. Формат выбора класса полёта реализован с помощью элемента carousel\_slider, слайды которого формируются на основе массива с названием класса, его описанием и стоимостью за всех пассажиров – Листинг 4.6

CarouselSlider.builder(

itemCount: \_flightClasses.length,

itemBuilder: (context, index, realIdx) {

return FlightClassTile(flightClass: \_flightClasses[index]);

},

options: CarouselOptions(

height: MediaQuery.of(context).size.height,

enlargeCenterPage: true,

onPageChanged: (index, reason) {

setState(() {

\_currentIndex = index;

});

},

),

)

Листинг 4.6 – Построение слайдера

Когда пользователь остановится на интересующем его классе и нажмёт кнопку «Select This Fare», методом Navigator.push он будет перенаправлен на страницу выбора места – SeatSelectionPage, полный скрипт данной страницы можно найти в приложении Б. Выглядит она следующим образом – рисунок 4.3.

С помощью метода getSeatsByClass получаем количество мест на данном самолёте для выбранного класса, и через конструктор страницы получаем занятые места для этого рейса из таблицы «Ticket» соответствующим методом класса DatabaseHelper. После чего нам нужно «отметить» столько мест – виджетов FilterChip, сколько мы указали на моменте поиска рейса.

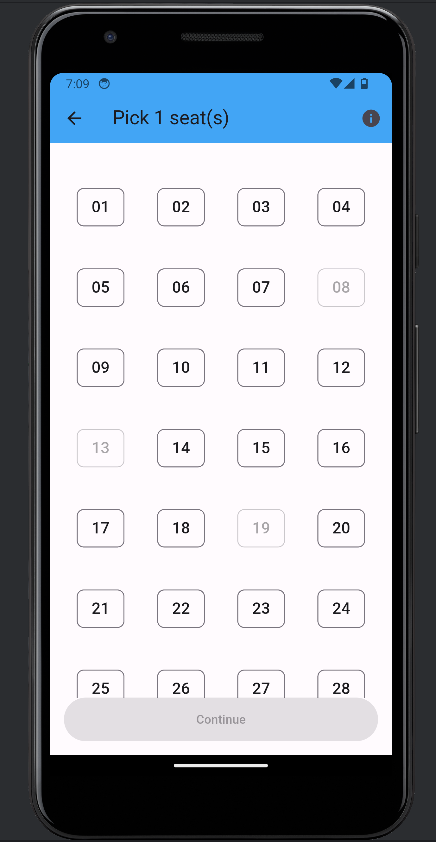


Рисунок 4.3 – Страница выбора места

После выбора указанного количества мест кнопка «Continue» станет доступной к нажатию и можно будет перейти к странице подтверждения – ConfirmationPage. На ней будет расположена краткая информация о рейсе, билет на который хочет забронировать пользователь, далее будет идти блок с персональными данными пассажиров, на которых будет бронироваться билет. После нажатия на кнопку «Confirm», пользователь будет перенаправлен на страницу MyTrips – с активными бронированиями и историей бронирования.

* 1. **Страница бронирований пользователя**

На данной странице расположен TabBar, его вкладки – «Active», «History» не имеют содержимого если пользователь не авторизован – лишь блок с подсказкой, что пользователю необходимо войти в учётную запись. После входа в учётную запись пользователь может найти на этой странице свои активные бронирования, а также историю бронирований – рисунок 4.4.

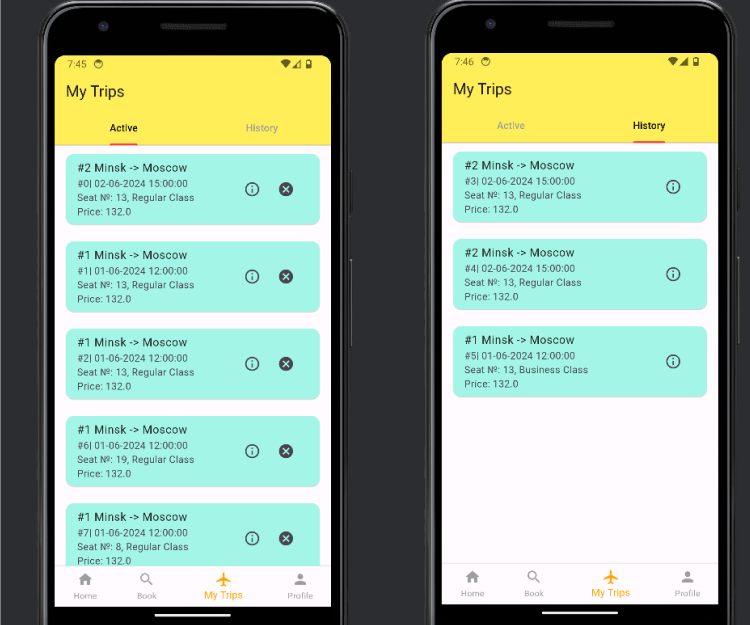


Рисунок 4.4 – Страница бронирований пользователя

При нажатии на иконку информации появится всплывающий блок с полной информацией о рейсе. Активную бронь можно отменить, нажав на иконку отмены и подтвердив действие в диалоговом окне. Данные о бронированиях получаются асинхронно методом getActiveBookings() и getBookingHistory() класса DatabaseHelper, которые возвращают Future<List<FlightResult>>. В интерфейсе блоки с информацией о брони строятся динамически, на основе виджета FutureBuilder, который позволяет удобно обрабатывать результаты асинхронных операций и на основе их генерировать пользовательский интерфейс.

Организация первой вкладки с активными бронированиями представлена в листинге 4.7.

FutureBuilder<List<FlightResult>>(

future: databaseHelper.getActiveBookings(databaseHelper.getLoggedInUser()!.systemUserId),

builder: (BuildContext context, AsyncSnapshot<List<FlightResult>> snapshot) {

if (snapshot.hasData) {

List<FlightResult> flightResults = snapshot.data!;

List<Widget> bookings = flightResults.map((flightResult) {

return Padding(

padding: const EdgeInsets.all(8.0),

child: \_buildBookingCard(flightResult),

);

}).toList();

return SingleChildScrollView(

child: Column(children: bookings),

);

} else if (snapshot.hasError) {

return Text('Error: ${snapshot.error}');

} else {

return CircularProgressIndicator();

}

},

),

Листинг 4.6 – Динамическое построение разметки в FutureBuilder

Вкладка с историей бронирования практически идентична этой, однако она использует другой виджет построения карточки бронирования, которая не включает функцию отмены.

* 1. **Выводы по разделу**

В данном разделе были рассмотрены основные аспекты реализации главных функций приложения – бронирование, управление бронированием, панель администратора, регистрация, авторизация, личный кабинет пользователя, а также технологии и подходы, которые обеспечили их реализацию.

1. **Тестирование приложения**

После реализации приложения необходимо проверить, что все элементы приложения работают так, как задумано. Данный раздел рассмотрит пошаговое исполнение сценария, выполняемого зарегистрированным пользователем – авторизация, бронирование билета, отмена билета, гостем – поиск рейса с условием и регистрация а также администратора – использование панели администратора.

* 1. **Гость**

Хронология действий поиска рейса продемонстрирована на рисунке 5.1. При запуске приложения гость будет направлен на главную страницу – первый кадр рисунка. Далее с помощью навигационного меню пользователь перейдёт на страницу поиска. Проверим, выполнится ли поиск с пустыми значениями формы – кадр 2.

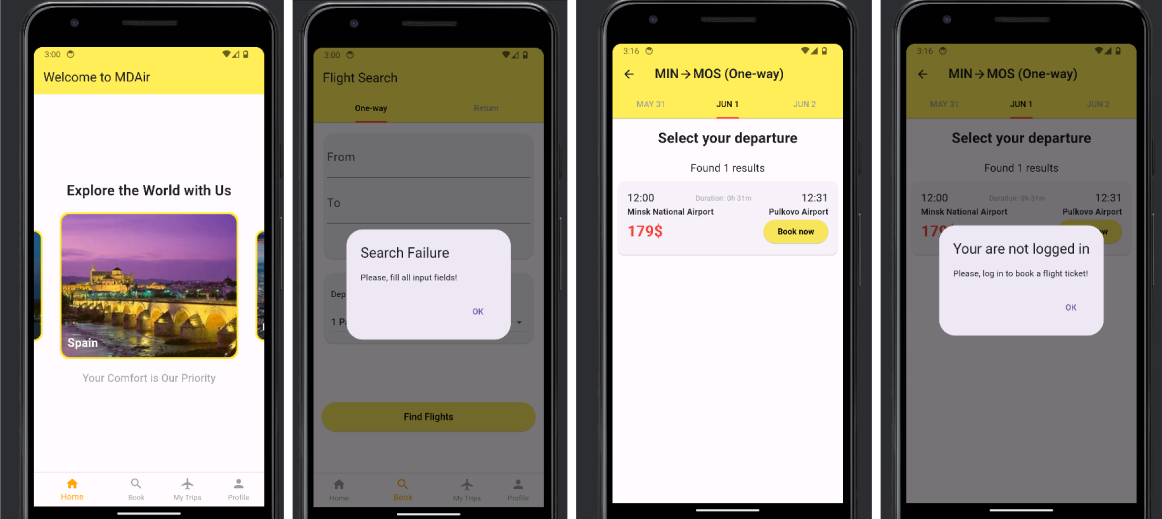


Рисунок 5.1 – Поиск рейса с условием

Как мы видим, нужно заполнить все поля, чтобы выполнить поиск. После заполнения всех полей пользователь нажмёт на кнопку «Search Flights». Далее его ждёт страница с результатами поиска. Если пользователь решит нажать на кнопку «Book Now» – кадр третий, то появится всплывающее уведомление, и при нажатии на кнопку «OK» выполнится переход на страницу входа. С этого места данный процесс заканчивается.

Далее пользователю предстоит процесс регистрации – рисунок 5.2. После нажатия кнопки для перехода на страницу регистрации пользователю необходимо заполнить форму с персональными данными.

Если пользователь ввёл не все значения из формы – он также получит об этом уведомление. После успешного заполнения всех полей он будет перенаправлен обратно на страницу авторизации.

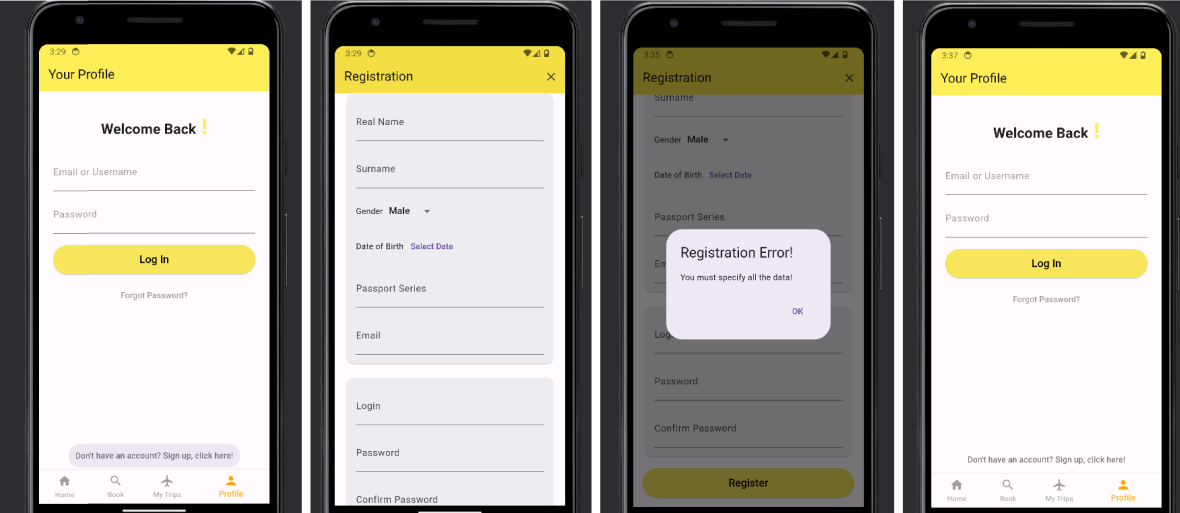


Рисунок 5.2 – Процесс регистрации

После этого пользователь уже имеет учётную запись и может полноценно взаимодействовать с бронированием билетов.

* 1. **Авторизация и бронирование билета**

На странице авторизации вкладки «Profile» пользователь выполняет авторизацию, причём, если не существует пользователя с таким логином и паролем, он получает ошибку – рисунок 5.3. После ввода нужных данных учётной записи пользователь попадает на страницу профиля с персональными данными – второй кадр рисунка. Тут при желании он может отдельно изменить какие-нибудь персональные данные, указанные на моменте регистрации, например, сменить почту или изменить имя, если оно было указано неправильно.

Процесс поиска билета авторизованным пользователем идентичен поиску билета гостем. После успешного входа в учётную запись и поиска интересующих рейсов. пользователь может перейти к бронированию билета .

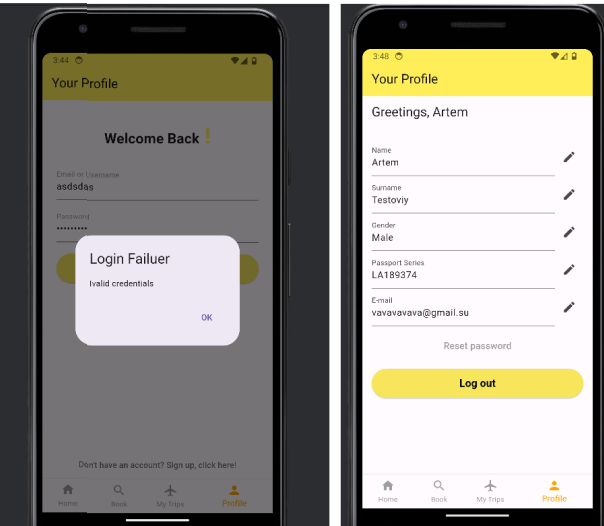


Рисунок 5.3 – Авторизация

На рисунке 5.4 описан весь процесс бронирования.

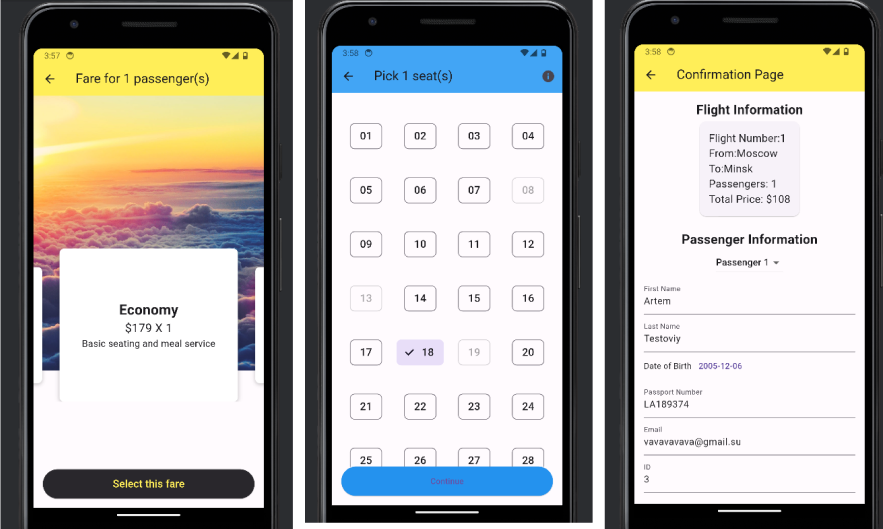


Рисунок 5.4 – Процесс бронирования

Как мы можем видеть, он состоит из трёх этапов. Первым делом идёт выбор класса полёта с помощью слайдера (по умолчанию – «эконом класс»).

На следующей странице появляется список мест, где пользователь должен выбрать столько мест, сколько пассажиров он указал. Иначе он не сможет продолжить процесс бронирования – кнопка «Continue» останется заблокированной. Последний этап – подтверждение бронирования. Пользователь указывает информацию о пассажире или пассажирах, если их несколько, используя форму. После нажатия на кнопку внизу страницы произойдёт бронирование билета и возврат пользователя на страницу активных бронирований – рисунок 5.5.

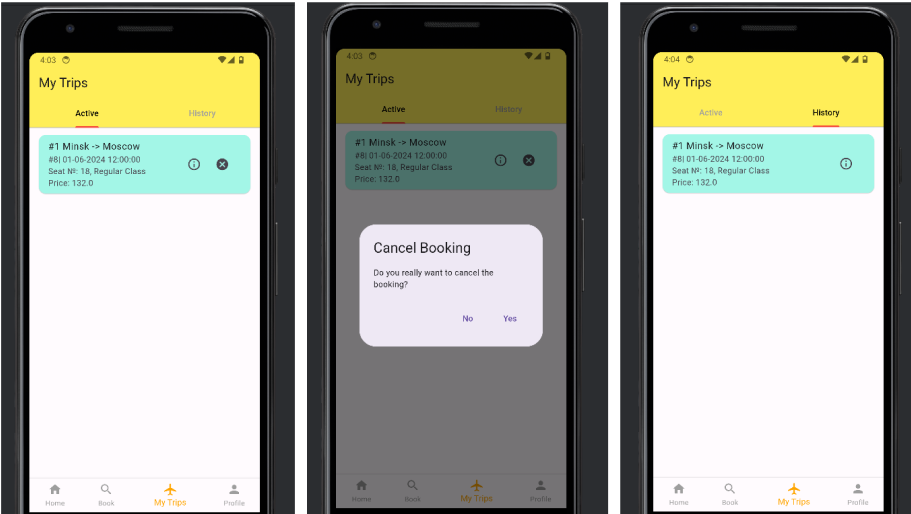


Рисунок 5.5 – Просмотр и отмена активной брони

Пользователь нажимает на иконку креста для отмены бронирования. После подтверждения этого действия интерфейс обновляется, и данная запись попадает на вкладку «History» данной страницы.

* 1. **Использование панели администратора**

Открыть панель администратора можно со страницы профиля – для этого, естественно, нужно выполнить вход под пользователем с ролью «admin» и нажать на кнопку «Open Admin Panel». Внешний вид панель администратора был показан ранее на рисунке 4.1. Так как здесь довольно однотипные страницы, протестируем одну сущность – аэропорт, операция вставки продемонстрированы на рисунке 5.6.

Как мы видим, новая запись успешно была добавлена и извлечена.

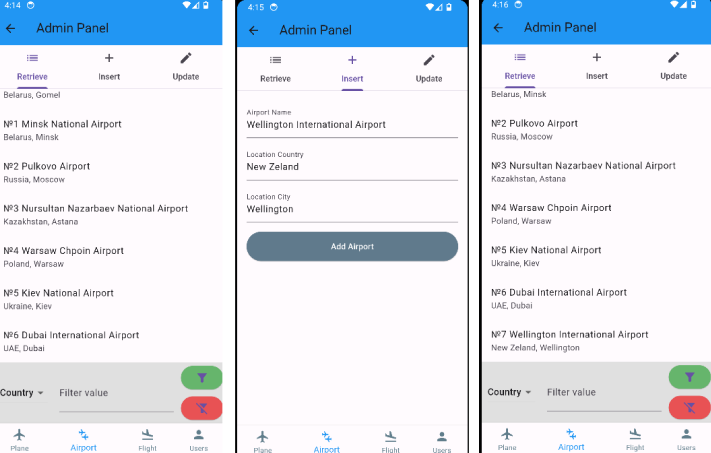


Рисунок 5.6 – Выполнение операции вставки

Далее проверим работоспособность фильтров – рисунок 5.7

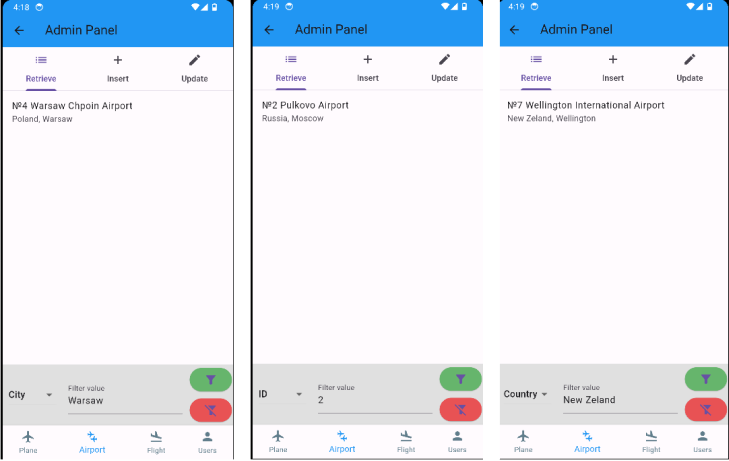


Рисунок 5.7 – Работоспособность фильтров

Функции фильтрации а также рендер работают, как и ожидается.

Далее попробуем изменить запись, предварительно выбрав её в фильтре – рисунок 5.8.

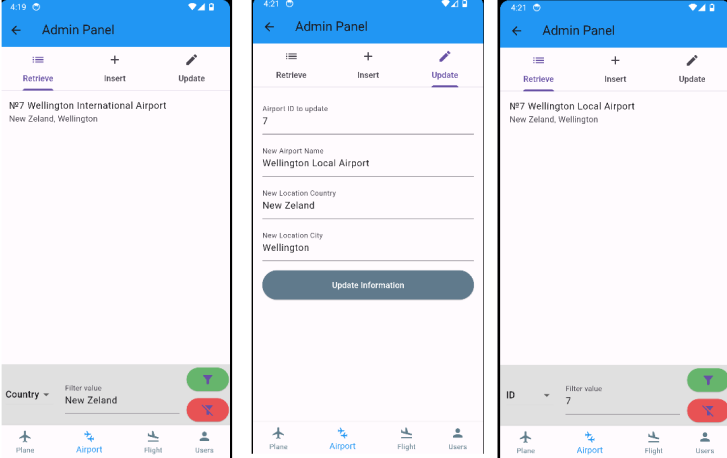


Рисунок 5.8 – Обновление данных

Название аэропорта было успешно изменено, следовательно делаем вывод, что панель администратора реализует заявленный функционал.

* 1. **Выводы по разделу**

В данном разделе было описано тестирование работоспособности основных функций приложения – поиск рейса с условием, регистрация, авторизация, бронирование билета и управление бронированием, взаимодействие с данными через панель администратора.

1. **Краткое описание приложения для демонстрации**

Приложение авиакомпании «MDAir» поставляется как .apk файл для установки на Android-смартфоне. Поддержка iOS будет внедрена сразу же после получения доступа к сборщику приложений для данной операционной системы. Приложение полностью реализует заявленные функции: авторизация и регистрация пользователей, поиск билетов на рейс с условием, бронирование билета, управление бронированием (просмотр, отмена), управление содержимым страниц базы данных для роли администратора.

При запуске приложения пользователь попадает на домашнюю страницу, где может ознакомиться с определённой информацией об авиакомпании. С помощью панель навигации снизу экрана пользователь может перемещаться между основными страницами приложения: «TicketSearchPage», на которой расположена форма поиска билета, «MyTripPage», на которой авторизованный пользователь может получить информацию о своих активных бронированиях, истории бронирования, а также отменить бронь, «ProfileConstructionPage», которая в зависимости от того, вошёл ли пользователь в свою учётную запись, отображает форму для входа в аккаунт либо страницу с персональными данными пользователя, на которой он может обновить основные персональные данные. Со страницы входа в учётную запись имеется доступ к странице регистрации, на которой расположена форма, которую пользователь должен заполнить, чтобы зарегистрироваться в приложении.

Процесс бронирования рейсов состоит из нескольких шагов: авторизация, поиск нужного билета, выбор интересующего рейса, выбор нужного класса полёта, выбор места, подтверждение данных. Все формы сопровождаются валидацией и автоматической навигацией для удобства пользователя.

Панель администратора доступна только авторизованному пользователю с ролью «admin», доступ к которой он может получить из своего профиля. Данная страница содержит несколько вкладок, на которых можно выполнять некоторые из «CRUD»-операций над следующими сущностями базы данных – Airports (аэропорты), Customers (пользователи), рейсы (Flights), самолёты (Planes).

1. **Руководство пользователя**

В этом разделе будут описаны основные шаги по взаимодействию пользователя с приложением «MDAir».

Для начала работы установите приложение с помощью .apk-файла и откройте его с рабочего стола своего смартфона. Вы видите перед собой главную страницу – рисунок 7.1

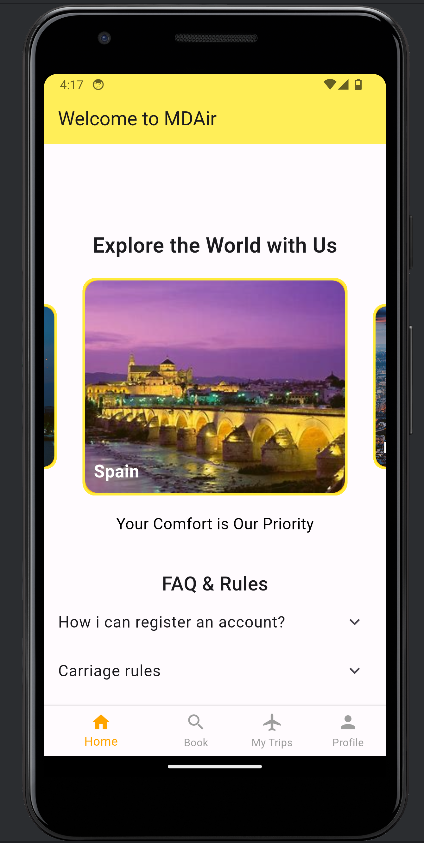


Рисунок 7.1 – Главная страница приложения

Дотсуп к другим основным страницам Вы можете получить из навигационной панели внизу экрана, оно будет закреплено в этой позиции на всех основных страницах.

На вкладке профиля Вы можете выполнить вход в свою учётную запись или зарегистрироваться, нажав на кнопку с подсказкой внизу страницы и заполнив форму необходимыми данными – рисунок 7.2. У всех полей есть свои условия для заполнения, например – обязательные, соответстующие какому-либо формату, ограничение минимальной длины.

По ходу заполнения форм и выполнения проверки вводимых данных вы будете уведомлены о несоответствиях, ошибках или успешном выполнении.

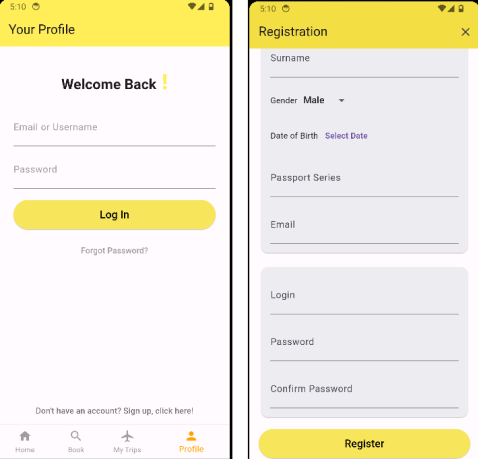


Рисунок 7.2 – Страница авторизации и регистрации.

После успешной регистрации и авторизации со вкладки «Profile» доступен личный кабинет пользователя – рисунок 7.3.

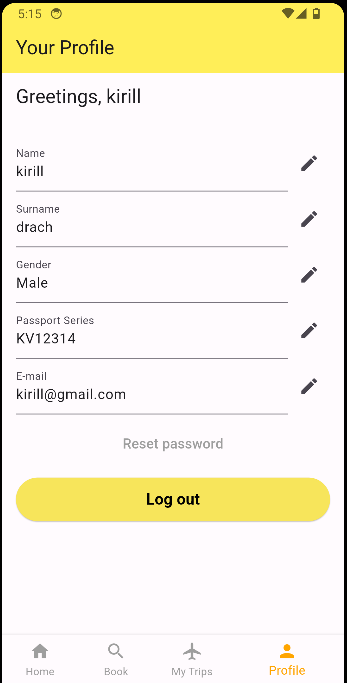


Рисунок 7.3 – Личный кабинет

В профиле вы можете редактировать свои персональные данные нажав на соответствующую иконку напротив определённого поля.

Поиск билетов начинается на вкладке «Book» и продолжается дальше на связанных с ней страницах – рисунок 7.4, Вам нужно лишь следовать указаниям из «шапки» страницы.

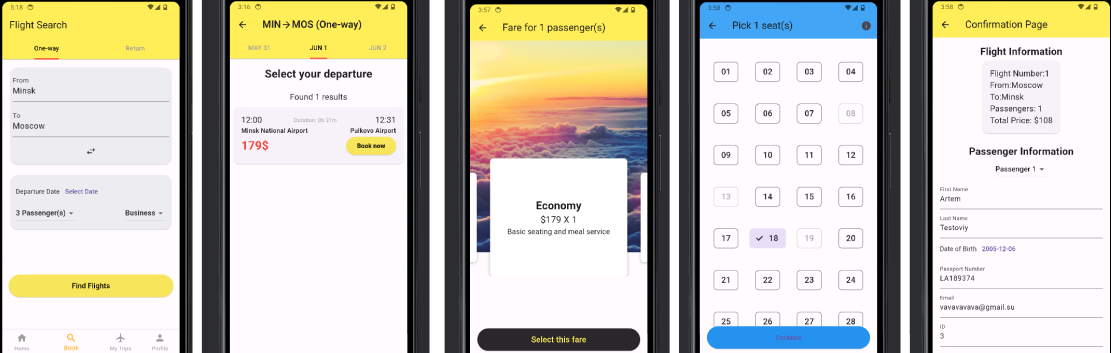


Рисунок 7.4 – Процесс бронирования билета

Свои активные бронирования и историю бронирований вы можете просмотреть со вкладки «My Trips» навигационной панели – рисунок 7.5, и тут же отменить активную бронь.

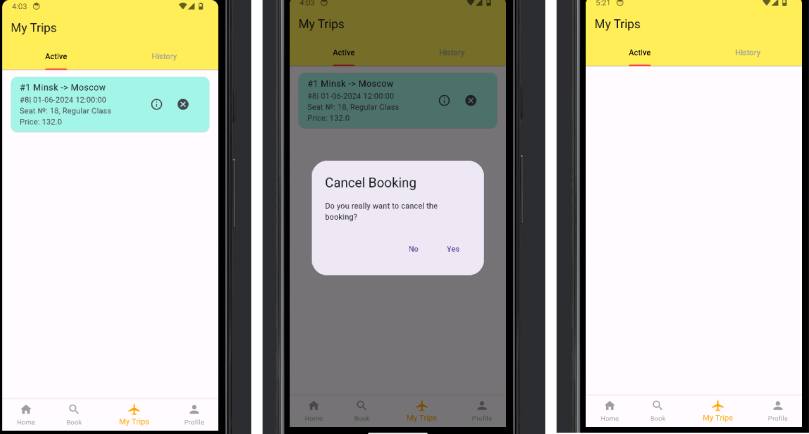


Рисунок 7.5 – Просмотр и управление бронированиями

На этом функционал пользователя заканчивается. Функция техподдержки появится в следующих версиях приложения.

# **ЗАКЛЮЧЕНИЕ**

В ходе разработки мобильного приложения для авиакомпании на платформе Flutter были успешно реализованы следующие функциональные требования: авторизация и регистрация пользователей, поиск рейсов с условием, управление рейсами, личный кабинет пользователя.

Спроектирована логическая и физическая структура базы данных, описывающая необходимые объекты для полноценной реализации заявленных функций с разделением ролей пользователей. Приложение разработано с использованием пакета sqflite для хранения данных в базе данных SQLite.

Применён функционал класса Provider пакета Provider, который позволил передавать единственный экземпляр класса DatabaseHelper, позволяющий взаимодействовать с базой данных приложения и реализующий порождающий шаблон проектирования – Singleton.

Реализована страница с панелью администратора, на которой расположены элементы взаимодействия с со следующими сущностями базы данных – самолёты, аэропорты, пользователи, рейсы, с помощью которых администратор может просматривать, обновлять и редактировать информацию о них.

Разработана оптимальная схема навигации в приложении, согласно которой пользователь сможет самым удобным способом получать доступ к страницам с нужной информацией.

Определена UI/UX составляющая дизайна каждой страницы приложения – расположения интерактивных и информационных элементов, их стилей, внешнего вида, обработка событий взаимодействия.

Получены навыки работы с асинхронными методами, способами их обработки, а также динамическое построение интерфейса на основе результата их выполнения с помощью виджета FutureBuilder.

Получены навыки взаимодействия с формами и элементами пользовательского ввода и способами валидации входных данных.

Исследована целесообразность использования виджетов без состояния и с состоянием в разных ситуациях, а также получены практические навыки внедрения данных типов виджетов в приложение.

Успешно протестирована работоспособность приложения на разных программно-аппаратных платформах.

# **Список используемых источников**

1. Андрей Алеев, Быстрый старт Flutter-разработчика/ А.В. Алеев. – Москва: Ridero, 2020. – 72 с. – Дата доступа: 29.02.2024

2. Rich Rose. Flutter and Dart Cookbook: Developing Full-Stack Applications for the Cloud / R. Rose. – O'Reilly, 2023. – 129 с.Дата доступа – 05.03.2024.

3. Блинова Е.А. Курс лекций по базам данных [Электронный ресурс]. – Режим доступа: https://diskstation.belstu.by:5001/. – Дата доступа: 17.03.2024.

4. Документация пакета sqflite [Электронный ресурс]. – Режим доступа: https://pub.dev/documentation/sqflite/latest/ – Дата доступа: 22.03.2024.

5. Flutter API reference documentation [Электронный ресурс]. – Режим доступа: https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html – Дата доступа: 14.04.2024

6. StackOverflow [Электронный ресурс] / Режим доступа: https://ru.stackoverflow.com/questions/Условный-рендеринг-во-flutter – Дата доступа: 18.04.2024

# **Приложение А. Взаимодействие с базой данных**

import 'package:sqflite/sqflite.dart';

import 'package:path/path.dart';

import 'package:mdair/models/customer.dart';

import 'package:mdair/models/ticket.dart';

import 'package:intl/intl.dart';

import 'models/airport.dart';

import 'models/flight\_info.dart';

import 'models/plane.dart';

class DatabaseHelper {

static final DatabaseHelper instance = DatabaseHelper.\_init();

static Database? \_database;

DatabaseHelper.\_init();

Future<Database> get database async {

if (\_database != null) return \_database!;

\_database = await \_initDB('mdair.db');

return \_database!;

}

Future<Database> \_initDB(String filePath) async {

final dbPath = await getDatabasesPath();

final path = join(dbPath, filePath);

return await openDatabase(path, version: 1, onCreate: \_createDB);

}

Customer? \_loggedInUser;

bool \_isAdmin = false;

void setLoggedInUser(Customer user) {

\_loggedInUser = user;

\_isAdmin = (user.userRole == 'admin');

}

Customer? getLoggedInUser() {

return \_loggedInUser;

}

Future<void> logout() async {

\_loggedInUser = null;

\_isAdmin = false;

}

late int \_selectedFlightid;

int get selectedFlightid => \_selectedFlightid;

set selectedFlightid(int value) {

\_selectedFlightid = value;

}

late int \_passengersCount;

int get passengersCount => \_passengersCount;

set passengersCount(int value) {

\_passengersCount = value;

}

late String \_departureCity;

late String \_destingationCity;

String get departureCity => \_departureCity;

set departureCity(String value) {

\_departureCity = value;

}

Future<List<FlightInfo>> findFlights(String departureCity, String destinationCity, DateTime departureDate, int passengerCount) async {

final db = await database;

var departureAirports = await db.rawQuery('SELECT airport\_id FROM Airport WHERE location\_city = ?', [departureCity]);

var destinationAirports = await db.rawQuery('SELECT airport\_id FROM Airport WHERE location\_city = ?', [destinationCity]);

String formattedDate = DateFormat('dd-MM-yyyy').format(departureDate);

List<FlightInfo> flightInfos = [];

for (var departureAirport in departureAirports) {

for (var destinationAirport in destinationAirports) {

var flights = await db.rawQuery('''

SELECT

Flight.flight\_number,

Flight.departure\_datetime,

Flight.arrive\_datetime,

Flight.available\_seats,

DepartureAirport.airport\_name AS departure\_airport,

DestinationAirport.airport\_name AS destination\_airport

FROM Flight

INNER JOIN Airport AS DepartureAirport ON Flight.departure\_airport\_id = DepartureAirport.airport\_id

INNER JOIN Airport AS DestinationAirport ON Flight.destination\_airport\_id = DestinationAirport.airport\_id

WHERE Flight.departure\_airport\_id = ?

AND Flight.destination\_airport\_id = ?

AND substr(Flight.departure\_datetime, 1, 10) = ?

AND Flight.available\_seats >= ?

''', [

departureAirport['airport\_id'],

destinationAirport['airport\_id'],

formattedDate,

passengerCount

]);

for (var flight in flights) {

flightInfos.add(FlightInfo(

flightId: int.parse(flight["flight\_number"].toString()),

departureTime: flight["departure\_datetime"].toString().substring(11, 16),

arrivalTime: flight["arrive\_datetime"].toString().substring(11, 16),

duration: calculateTimeDifference(flight["departure\_datetime"].toString().substring(11, 16),

flight["arrive\_datetime"].toString().substring(11, 16)),

price: "179",

departureAirport: flight["departure\_airport"].toString(),

destinationAirport: flight["destination\_airport"].toString(),

availableSeats: int.parse(flight["available\_seats"].toString()),

));

}

}

}

return flightInfos;

}

Future<List<int>> getOccupiedSeats(int flightNumber, String ticketClass) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.rawQuery('''

SELECT DISTINCT place\_number

FROM Ticket

WHERE flight\_number = ? AND ticket\_class = ? AND status\_id = 1

''', [flightNumber, ticketClass]);

return List.generate(maps.length, (i) {

return maps[i]['place\_number'];

});

}

Future<void> bookTicket(int place\_number,double price, fareType) async {

Ticket ticket = Ticket(ticketNum: 1, flightNumber: selectedFlightid, placeNumber: place\_number, price: price, statusId: 1, systemUserId: \_loggedInUser!.systemUserId, ticketClass: fareType);

final db = await database;

await db.transaction((txn) async {

await txn.insert(

'Ticket',

ticket.toMap(),

conflictAlgorithm: ConflictAlgorithm.replace,

);

List<Map<String, dynamic>> flights = await txn.query(

'Flight',

where: 'flight\_number = ?',

whereArgs: [ticket.flightNumber],

);

int availableSeats = flights[0]['available\_seats'] - 1;

await txn.update(

'Flight',

{'available\_seats': availableSeats},

where: 'flight\_number = ?',

whereArgs: [ticket.flightNumber],

);

});

}

Future<List<FlightResult>> getActiveBookings(int userId) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.rawQuery('''

SELECT

Ticket.ticket\_num,

Ticket.place\_number,

Ticket.price,

Flight.departure\_datetime,

DepartureAirport.location\_city || ', ' || DepartureAirport.location\_country || ', ' || DepartureAirport.airport\_name AS departure\_airport,

ArrivalAirport.location\_city || ', ' || ArrivalAirport.location\_country || ', ' || ArrivalAirport.airport\_name AS arrival\_airport,

Flight.flight\_number,

Ticket.ticket\_class

FROM Ticket

INNER JOIN Flight ON Ticket.flight\_number = Flight.flight\_number

INNER JOIN Airport AS DepartureAirport ON Flight.departure\_airport\_id = DepartureAirport.airport\_id

INNER JOIN Airport AS ArrivalAirport ON Flight.destination\_airport\_id = ArrivalAirport.airport\_id

WHERE Ticket.system\_user\_id = ? AND Ticket.status\_id = 1

''', [userId]);

return List.generate(maps.length, (i) {

return FlightResult(

ticketNum: maps[i]['ticket\_num'],

seatNumber: maps[i]['place\_number'],

price: maps[i]['price'],

departureDate: maps[i]['departure\_datetime'],

departureAirport: maps[i]['departure\_airport'],

arrivalAirport: maps[i]['arrival\_airport'],

flightNumber: maps[i]['flight\_number'],

ticketClass: maps[i]['ticket\_class'],

);

});

}

Future<List<FlightResult>> getInactiveBookings(int userId) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.rawQuery('''

SELECT

Ticket.ticket\_num,

Ticket.place\_number,

Ticket.price,

Flight.departure\_datetime,

DepartureAirport.location\_city || ', ' || DepartureAirport.location\_country || ', ' || DepartureAirport.airport\_name AS departure\_airport,

ArrivalAirport.location\_city || ', ' || ArrivalAirport.location\_country || ', ' || ArrivalAirport.airport\_name AS arrival\_airport,

Flight.flight\_number,

Ticket.ticket\_class

FROM Ticket

INNER JOIN Flight ON Ticket.flight\_number = Flight.flight\_number

INNER JOIN Airport AS DepartureAirport ON Flight.departure\_airport\_id = DepartureAirport.airport\_id

INNER JOIN Airport AS ArrivalAirport ON Flight.destination\_airport\_id = ArrivalAirport.airport\_id

WHERE Ticket.system\_user\_id = ? AND Ticket.status\_id != 1

ORDER BY Ticket.ticket\_num DESC

LIMIT 20

''', [userId]);

return List.generate(maps.length, (i) {

return FlightResult(

ticketNum: maps[i]['ticket\_num'],

seatNumber: maps[i]['place\_number'],

price: maps[i]['price'],

departureDate: maps[i]['departure\_datetime'],

departureAirport: maps[i]['departure\_airport'],

arrivalAirport: maps[i]['arrival\_airport'],

flightNumber: maps[i]['flight\_number'],

ticketClass: maps[i]['ticket\_class'],

);

});

}

Future<void> cancelBooking(int ticketNum) async {

final db = await database;

await db.update(

'Ticket',

{'status\_id': 2},

where: 'ticket\_num = ?',

whereArgs: [ticketNum],

);

}

bool isAdmin() {

return \_isAdmin;

}

Future<Customer?> authenticateUser(String login, String password) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query(

'Customer',

where: 'user\_login = ? AND user\_password = ?',

whereArgs: [login, password],

);

if (maps.isNotEmpty) {

final customerMap = maps.first;

final customer = Customer.fromMap(customerMap);

if(customer.userRole == "admin"){

\_isAdmin = true;

}

else{

\_isAdmin = false;

}

DatabaseHelper.instance.setLoggedInUser(customer);

return customer;

} else {

return null;

}

}

late int \_seatsCountByClass;

Future<int> getSeatsByClass(int flightId, String classType) async {

final db = await database;

var result = await db.rawQuery('''

SELECT Plane.${classType.toLowerCase()}\_seats\_capacity

FROM Flight

INNER JOIN Plane ON Flight.plane\_id = Plane.plane\_id

WHERE Flight.flight\_number = ?

''', [flightId]);

if (result.isNotEmpty) {

int totalSeats = int.parse(result.first["${classType.toLowerCase()}\_seats\_capacity"].toString());

\_seatsCountByClass = totalSeats;

return totalSeats;

} else {

throw Exception('Flight not found');

}

}

late String \_fareType;

String get fareType => \_fareType;

set fareType(String value) {

\_fareType = value;

}

Future<void> insertCustomer(Customer customer) async {

final db = await database;

await db.insert('Customer', customer.toMap());

}

Future<List<Plane>> getPlanes() async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query('Plane');

return List.generate(maps.length, (i) {

return Plane.fromMap(maps[i]);

});

}

Future<void> insertPlane(Plane plane) async {

final db = await database;

await db.insert(

'Plane',

plane.toMap(),

conflictAlgorithm: ConflictAlgorithm.replace,

);

}

Future<void> updatePlane(Plane plane) async {

final db = await database;

await db.update(

'Plane',

plane.toMap(),

where: "plane\_id = ?",

whereArgs: [plane.planeId],

);

}

Future<void> insertAirport(Airport airport) async {

final db = await database;

await db.insert(

'Airport',

airport.toMap(),

conflictAlgorithm: ConflictAlgorithm.replace,

);

}

Future<void> updateAirport(Airport airport) async {

final db = await database;

await db.update(

'Airport',

airport.toMap(),

where: "airport\_id = ?",

whereArgs: [airport.airportId],

);

}

Future<List<Airport>> getAirports() async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query('Airport');

return List.generate(maps.length, (i) {

return Airport.fromMap(maps[i]);

});

}

Future<List<Airport>> getAirportsFilteredByCountry(String country) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query(

'Airport',

where: 'location\_country = ?',

whereArgs: [country],

);

return List.generate(maps.length, (i) {

return Airport.fromMap(maps[i]);

});

}

Future<List<Airport>> getAirportsFilteredByCity(String city) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query(

'Airport',

where: 'location\_city = ?',

whereArgs: [city],

);

return List.generate(maps.length, (i) {

return Airport.fromMap(maps[i]);

});

}

Future<List<Airport>> getAirportsFilteredByID(int id) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query(

'Airport',

where: 'airport\_id = ?',

whereArgs: [id],

);

return List.generate(maps.length, (i) {

return Airport.fromMap(maps[i]);

});

}

Future<List<Customer>> getCustomers() async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query('Customer');

return List.generate(maps.length, (i) {

return Customer.fromMap(maps[i]);

});

}

Future<void> updateCustomer(Customer customer) async {

final db = await database;

await db.update(

'Customer',

customer.toMap(),

where: "system\_user\_id = ?",

whereArgs: [customer.systemUserId],

);

}

Future<List<Customer>> getCustomersFilteredById(int id) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query(

'Customer',

where: 'system\_user\_id = ?',

whereArgs: [id],

);

return List.generate(maps.length, (i) {

return Customer.fromMap(maps[i]);

});

}

Future<List<Customer>> getCustomersFilteredBySurname(String surname) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query(

'Customer',

where: 'user\_surname = ?',

whereArgs: [surname],

);

return List.generate(maps.length, (i) {

return Customer.fromMap(maps[i]);

});

}

Future<List<Customer>> getCustomersFilteredByName(String name) async {

final db = await database;

final List<Map<String, dynamic>> maps = await db.query(

'Customer',

where: 'user\_real\_name = ?',

whereArgs: [name],

);

return List.generate(maps.length, (i) {

return Customer.fromMap(maps[i]);

});

}

Future<void> \_createDB(Database db, int version) async {

await db.execute('''CREATE TABLE Airport (

airport\_id INTEGER PRIMARY KEY AUTOINCREMENT,

airport\_name TEXT,

location\_country TEXT,

location\_city TEXT

)

''');

await db.execute('''CREATE TABLE Flight\_status (

status\_id INTEGER PRIMARY KEY AUTOINCREMENT,

status\_description TEXT

)

''');

await db.execute('''CREATE TABLE Plane (

plane\_id INTEGER PRIMARY KEY AUTOINCREMENT,

model TEXT NOT NULL,

business\_seats\_capacity INTEGER NOT NULL,

economy\_seats\_capacity INTEGER NOT NULL

)

''');

await db.execute('''CREATE TABLE Flight (

flight\_number INTEGER PRIMARY KEY AUTOINCREMENT,

plane\_id INTEGER NOT NULL,

departure\_airport\_id INTEGER NOT NULL,

destination\_airport\_id INTEGER NOT NULL,

departure\_datetime TEXT NOT NULL,

arrive\_datetime TEXT NOT NULL,

available\_seats INTEGER NOT NULL,

status INTEGER NOT NULL,

FOREIGN KEY (plane\_id) REFERENCES Plane(plane\_id),

FOREIGN KEY (departure\_airport\_id) REFERENCES Airport(airport\_id),

FOREIGN KEY (destination\_airport\_id) REFERENCES Airport(airport\_id),

FOREIGN KEY (status) REFERENCES Flight\_status(status\_id)

)

''');

await db.execute('''CREATE TABLE Ticket (

ticket\_num INTEGER PRIMARY KEY AUTOINCREMENT,

flight\_number INTEGER NOT NULL,

place\_number INTEGER NOT NULL,

price REAL NOT NULL,

status\_id INTEGER NOT NULL,

system\_user\_id INTEGER NOT NULL,

ticket\_class TEXT CHECK(ticket\_class IN ('Regular', 'Business')) NOT NULL,

FOREIGN KEY (flight\_number) REFERENCES Flight(flight\_number),

FOREIGN KEY (status\_id) REFERENCES Ticket\_status(status\_num),

FOREIGN KEY (system\_user\_id) REFERENCES Customer(system\_user\_id)

)

''');

await db.execute('''CREATE TABLE Ticket\_status (

status\_num INTEGER PRIMARY KEY AUTOINCREMENT,

status\_description TEXT NOT NULL

)

''');

await db.execute('''CREATE TABLE Customer (

system\_user\_id INTEGER PRIMARY KEY AUTOINCREMENT,

user\_real\_name TEXT NOT NULL,

user\_surname TEXT NOT NULL,

gender TEXT CHECK(gender IN ('Male', 'Female', 'Other')) NOT NULL,

date\_of\_birth TEXT NOT NULL,

passport\_series TEXT NOT NULL,

user\_email TEXT NOT NULL UNIQUE,

user\_login TEXT NOT NULL UNIQUE,

user\_password TEXT NOT NULL,

user\_role TEXT CHECK(user\_role IN ('user', 'admin')) NOT NULL

)

''');

}

String get destingationCity => \_destingationCity;

set destingationCity(String value) {

\_destingationCity = value;

}

}

Листинг А.1 – Класс DatabaseHelper

class Plane {

late int planeId;

late String model;

late int businessSeatsCapacity;

late int economySeatsCapacity;

Plane({

required this.planeId,

required this.model,

required this.businessSeatsCapacity,

required this.economySeatsCapacity,

});

factory Plane.fromMap(Map<String, dynamic> map) {

return Plane(

planeId: map['plane\_id'],

model: map['model'],

businessSeatsCapacity: map['business\_seats\_capacity'],

economySeatsCapacity: map['economy\_seats\_capacity'],

);

}

Map<String, dynamic> toMap() {

return {

'model': model,

'business\_seats\_capacity': businessSeatsCapacity,

'economy\_seats\_capacity': economySeatsCapacity,

};

}

@override

String toString() {

return 'Plane: $model (Business: $businessSeatsCapacity, Economy: $economySeatsCapacity)';

}

}

Листинг А.2 – Адаптация таблицы Plane в класс

# **Приложение Б. Виджеты страниц**

import 'package:flutter/material.dart';

import 'package:carousel\_slider/carousel\_slider.dart';

import 'package:provider/provider.dart';

import 'SeatSelectionPage.dart';

import 'package:mdair/database\_helper.dart';

class FareSelectionPage extends StatefulWidget {

late int \_flightId;

late int \_price;

late int \_passengersCount;

FareSelectionPage(int flightId, String price, int passengersCount) {

\_flightId = flightId;

\_price = int.tryParse(price)!;

\_passengersCount = passengersCount;

}

@override

\_FareSelectionPageState createState() => \_FareSelectionPageState();

}

class \_FareSelectionPageState extends State<FareSelectionPage> {

late int price = widget.\_price;

late int passengersCount = widget.\_passengersCount;

int \_currentIndex = 0;

late final List<FlightClass> \_flightClasses = [

FlightClass(name: 'Economy', price: price.toString(), details: 'Basic seating and meal service', passendersNumber: passengersCount),

FlightClass(name: 'Business', price: ((price \* 1.4).truncate()).toString(), details: 'Spacious seating and gourmet meals', passendersNumber: passengersCount),

];

late DatabaseHelper databaseHelper;

@override

Widget build(BuildContext context) {

databaseHelper = Provider.of<DatabaseHelper>(context);

databaseHelper.selectedFlightid = widget.\_flightId;

print(databaseHelper.selectedFlightid);

return Scaffold(

appBar: AppBar(

backgroundColor: Colors.yellow.shade400,

leading: IconButton(

icon: Icon(Icons.arrow\_back),

onPressed: () => Navigator.pop(context),

),

title: Text('Fare for ${passengersCount} passenger(s)'), // Updated to show flight direction

),

body: Stack(

children: <Widget>[

Column(

children: <Widget>[

Expanded(

flex: 2,

child: Image.network('https://wallup.net/wp-content/uploads/2016/01/68375-clouds-nature-sunrise-sunlight-sky.jpg', fit: BoxFit.cover), // Replace with your image

),

Expanded(

flex: 1,

child: Container(),

),

],

),

Positioned(

top: MediaQuery.of(context).size.height / 3, // Adjust the position to overlay the image

left: 0,

right: 0,

child: CarouselSlider.builder(

itemCount: \_flightClasses.length,

itemBuilder: (context, index, realIdx) {

return FlightClassTile(flightClass: \_flightClasses[index]);

},

options: CarouselOptions(

height: MediaQuery.of(context).size.height / 3, // Adjust the height to fit the screen

enlargeCenterPage: true,

onPageChanged: (index, reason) {

setState(() {

\_currentIndex = index;

});

},

),

),

),

Positioned(

bottom: 16,

left: 16,

right: 16,

child: ElevatedButton(

onPressed: () async {

String fareType = \_flightClasses[\_currentIndex].name == "Economy" ? "Regular" : "Business";

databaseHelper.fareType = fareType;

int totalSeatsCountByClass = await databaseHelper.getSeatsByClass(databaseHelper.selectedFlightid, \_flightClasses[\_currentIndex].name);

List<int> places = await databaseHelper.getOccupiedSeats(databaseHelper.selectedFlightid, fareType);

for(int place in places){

print(place);

}

print(totalSeatsCountByClass);

Navigator.push(

context,

MaterialPageRoute(builder: (context) => SeatSelectionPage(databaseHelper,

\_flightClasses[\_currentIndex].name,

\_flightClasses[\_currentIndex].passendersNumber,

totalSeatsCountByClass,

places)),

);

},

child: Text('Select this fare', style: TextStyle(color: Colors.yellow.shade400, fontSize: 18),),

style: ElevatedButton.styleFrom(

minimumSize: Size(double.infinity, 50),

backgroundColor: Color.fromRGBO(0, 0, 0, 0.8)

),

),

),

],

),

);

}

}

class FlightClass {

final String name;

final String price;

final String details;

final int passendersNumber;

FlightClass({

required this.name,

required this.price,

required this.details,

required this.passendersNumber

});

}

class FlightClassTile extends StatelessWidget {

final FlightClass flightClass;

FlightClassTile({required this.flightClass});

@override

Widget build(BuildContext context) {

return Container(

width: MediaQuery.of(context).size.width,

margin: EdgeInsets.symmetric(horizontal: 5.0),

padding: EdgeInsets.all(16.0), // Added padding for the text

decoration: BoxDecoration(

color: Colors.white,

borderRadius: BorderRadius.circular(8),

boxShadow: [

BoxShadow(

color: Colors.black26,

blurRadius: 6,

offset: Offset(0, 2),

),

],

),

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: <Widget>[

Text(flightClass.name, style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold)),

Text("\$${flightClass.price} X ${flightClass.passendersNumber}", style: TextStyle(fontSize: 20)),

Text(flightClass.details, style: TextStyle(fontSize: 16)),

],

),

);

}

}

Листинг Б.1 – Страница FareSelectionPage