МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение высшего образования

«Тольяттинский государственный университет»

Кафедра «Прикладная математика и информатика»

(наименование)

09.03.03 Прикладная информатика

(код и наименование направления подготовки / специальности)

Разработка программного обеспечения

(код и наименование направления подготовки / специальности)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему: «Разработка мобильного приложения прогноза погоды»

Обучающийся В.А. Гриванова

(Инициалы Фамилия) (Подпись)

Руководитель к. п. н., доцент, О.В. Оськина

(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант к. ф. н., доцент, М.В. Дайнеко

(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2025

Аннотация

Тема бакалаврской работы: «Разработка мобильного приложения прогноза погоды».

Объектом исследования является процесс разработки мобильного приложения для прогноза погоды.

Предметом исследования являются методы и технологии разработки мобильных приложений, а также алгоритмы и источники данных для прогноза погоды.

Целью выпускной квалификационной работы является разработка мобильного приложения для прогноза погоды.

В первом разделе рассматриваются функциональные и архитектурные особенности мобильных приложений прогноза погоды, формулируются требования к мобильному приложению, проводится и анализ существующих аналогов.

Во втором разделе описывается процесс проектирования архитектуры мобильного приложения, пользовательского интерфейса и выбор инструментов для реализации мобильного приложения. Была составлена диаграмма вариантов использования и диаграмма компонентов.

В третьем разделе представлена реализация и тестирование мобильного приложения.

Практическая значимость данной работы заключается в разработке мобильного приложения прогноза погоды, которое может отображать данные без подключения к интернету и отправлять условные уведомления.

Данная бакалаврская работа состоит из 49 страниц, 32 рисунков, 4 таблиц и 25 источников.

Abstract

The topic of the graduation work is Developing a mobile weather forecast application.

This graduation work consists of an introduction, 3 parts, 32 figures, 4 tables and a list of 25 sources references.

The object of the research is the process of developing a mobile application for weather forecast.

The subject of the research is the methods and the technologies of developing mobile applications, as well as the algorithms and the data sources for weather forecast.

The purpose of the study is the development of the mobile application for weather forecast.

The first part of the research examines the functional and architectural features of the mobile weather forecast applications. In this part, the requirements for the mobile application are established, and the existing analogues are analyzed as well.

The second part describes the process of designing the architecture of the mobile application and the user interface, as well as the choice of the tools for implementing the mobile application. In this part, a use cases diagram and a diagram of components are presented.

The third part is devoted to implementing and testing the mobile application.

In conclusion, it should be highlighted that the developed mobile weather forecast application successfully meets the key requirements. The application features a basic set of the necessary functions and ensures the display of the current weather data even when there is no Internet connection thanks to the implemented cache mechanism. Additionally, a system of sending weather forecast notifications has also been implemented.

Содержание

Введение

В условиях современного мира, где информация становится все более доступной и быстро распространяется, мобильные приложения занимают важное место в повседневной жизни людей. Особую актуальность приобретают приложения с информацией о погоде, поскольку погодные условия оказывают влияние на различные сферы нашей жизни, такие как туризм, сельское хозяйство, транспорт и другие. В связи с этим такие приложения становятся все более популярными и востребованными среди пользователей.

Актуальность выбранной темы обусловлена растущим спросом на мобильные приложения, предоставляющие информацию о погоде. В условиях изменения климата мы все чаще сталкиваемся с неожиданными ливнями, аномальной жарой или резкими похолоданиями. В таких условиях мобильное приложение с актуальной информацией о погоде становится не просто удобным инструментом, но и необходимостью для планирования каждого дня. При этом современные технологии позволяют создавать умные решения с полезными функциями и удобным интерфейсом. Однако многие существующие приложения не используют все возможности в полной мере. Именно это делает разработку нового мобильного приложения актуальной задачей.

Объектом исследования является процесс разработки мобильного приложения для прогноза погоды.

Предметом исследования являются методы и технологии разработки мобильных приложений, а также алгоритмы и источники данных для прогноза погоды.

Целью данной дипломной работы является разработка мобильного приложения, предоставляющего пользователям актуальную и точную информацию о погоде, новые функциональные возможности и эффективную визуализацию информации.

Методами исследования являются анализ, проектирование, программирование, тестирование.

Практическая значимость данной работы заключается в разработке мобильного приложения прогноза погоды, которое может отображать данные без подключения к интернету и отправлять условные уведомления.

Для достижения заданной цели работы необходимо осуществить следующие задачи:

- провести анализ предметной области;
- сформировать требования к мобильному приложению;
- выбрать средства разработки;
- спроектировать архитектуру мобильного приложения;
- разработать пользовательский интерфейс;
- выполнить реализацию приложения;
- провести тестирование;

Бакалаврская работа состоит из введения, трех разделов, заключения, списка используемой литературы и используемых источников.

1 Постановка задачи на разработку мобильного приложения для прогноза погоды

1.1 Функциональные и архитектурные особенности мобильных приложений прогноза погоды

Разработка мобильных приложений для прогнозирования погоды представляет собой сложную задачу, требующую многих аспектов. Необходимо обеспечить как функциональность приложения, так и удобство использования. Для того чтобы приложение стало качественным и востребованным среди пользователей, необходимо найти надежный источник актуальных и точных данных, а также создать гибкую архитектуру.

Базовой функцией подобных приложений является отображение текущих погодных условий, таких как температура, влажность, скорость ветра. Также необходимо представление о детальном прогнозе погоды на различные временные промежутки, включая почасовой прогноз, прогноз на несколько дней вперед и на месяц. Использование геолокационных сервисов необходимо для автоматического определения местоположения пользователя. Доступ к необходимым климатическим данным обеспечивает использование погодного API.

Помимо основных функций, мобильные приложения могут включать дополнительные возможности, такие как настраиваемые параметры, например единицы измерения, персонализированные данные, интерактивные карты или поддержка нескольких местоположений.

Архитектурными особенностями являются несколько важных характеристик. Разделение ответственности позволяет каждому модулю выполнять определенные задачи, что упрощает поиск кода и этапы разработки и тестирования. Модульность архитектуры обеспечивает гибкость и ускоряет разработку приложения. Тестируемость позволит легко проверять работу отдельных компонентов и приложения в целом. Масштабируемость сможет

гарантировать способность обрабатывать большие объемы данных без большой потери производительности.

Компоненты архитектуры обычно включают в себя клиентскую и серверную части. База данных может быть использована для хранения необходимых данных.

1.2 Формирование требований к мобильному приложению

Разработка требований является важным этапом, который определяет успех проекта. «Требования должны быть четко сформулированы и логически обоснованы, чтобы избежать ошибок на последующих стадиях разработки» [7].

Модель FURPS+ помогает анализировать и классифицировать требования к качеству программного обеспечения. Символ «+» в конце аббревиатуры обозначает дополнительные требования и ограничения, которые могут быть важны для конкретного проекта или компании. Это могут быть аспекты безопасности, соблюдение стандартов и другие важные моменты, которые следует учитывать при разработке и тестировании продукта. Основные преимущества использования данной модели заключаются в ее компактности и наличии наиболее востребованных характеристик. Модель FURPS+ чаще всего используется в большинстве небольших проектов [22].

Важно отметить, что требования могут уточняться или детализироваться по мере развития проекта. Однако учет всех необходимых требований на начальных этапах разработки позволяет избежать потенциальных проблем или серьезных изменений в будущем.

На основе этой классификации были сформированы требования к мобильному приложению, которые представлены в таблице 1.

Таблица 1 – Требования к мобильному приложению для прогноза погоды

Требование	Приоритет
Функциональные требования	
Отображение текущей погоды	Высокий
Отображение прогноза погоды на 14 дней	Высокий
Отображение почасового прогноза	Высокий
Поиск по названию города	Высокий
Автоматическое определение местоположения	Высокий
Настройка уведомлений	Высокий
Работа в офлайн-режиме: - Отображение кэшированных данных - Отображение ошибки статуса соединения и времени последнего обновления	Высокий
Удобство использования	
Эргономичный и удобный интерфейс	Средний
Надежность	
Доступность 24/7 и устойчивость к сбоям	Высокий
Производительность	
Загрузка данных меньше 2 секунд и минимальное время отклика	Высокий
Поддерживаемость	
Доступность на устройствах с операционной системой Android 10+	Высокий

1.3 Обзор и анализ существующих аналогов

Яндекс Погода – это бесплатный погодный сервис, использующий собственную технологию Метеум, основанную на нейронных сетях. На рисунке 1 показан скриншот главной страницы приложения.

Преимущества:

– интерактивная карта с отображением осадков в режиме реального времени;

– информация об уровне загрязнения воздуха, воды и ультрафиолетового излучения;

– точные местные прогнозы с точностью до квартала;

– функции обратной связи;

– возможность выбора нескольких мест для отслеживания погоды.

Недостатки:

– необходимость подключения к интернету;

– наличие рекламы, которую нельзя отключить.

Рисунок 1 – Экран приложения «Яндекс Погода»

Gismeteo представляет собой кроссплатформенное приложение, в официальной документации которого отмечается, что данные собираются из

различных источников, включая информацию Гидрометцентра России [1]. Скриншот приложения представлен на рисунке 2.

Преимущества:

– прогноз погоды на 14 дней с почасовой детализацией;
– возможность добавления нескольких городов для отслеживания;
– наличие разделов с новостями о погоде и климате.

Недостатки:

– ограниченное количество функций в бесплатной версии;
– необходимость подключения к интернету.

Рисунок 2 – Экран приложения «Gismeteo»

Yahoo погода – это кроссплатформенное приложение, разработанное компанией Yahoo Inc., которая в настоящее время принадлежит Appollo Global Management [2]. Сервис является частью экосистемы продуктов Yahoo и использует данные авторитетных международных метеорологических служб, включая The Weather Channel [3]. Рисунок 3 демонстрирует скриншот страниц приложения.

К преимуществам можно отнести отличительный интерфейс с красивыми изображениями, прогнозы погоды на 10 дней вперед с почасовой разбивкой и интеграцию с другими сервисами личного бренда. Недостатками

являются навязчивая реклама, ограниченные функции в бесплатной версии и необходимость подключения к интернету.

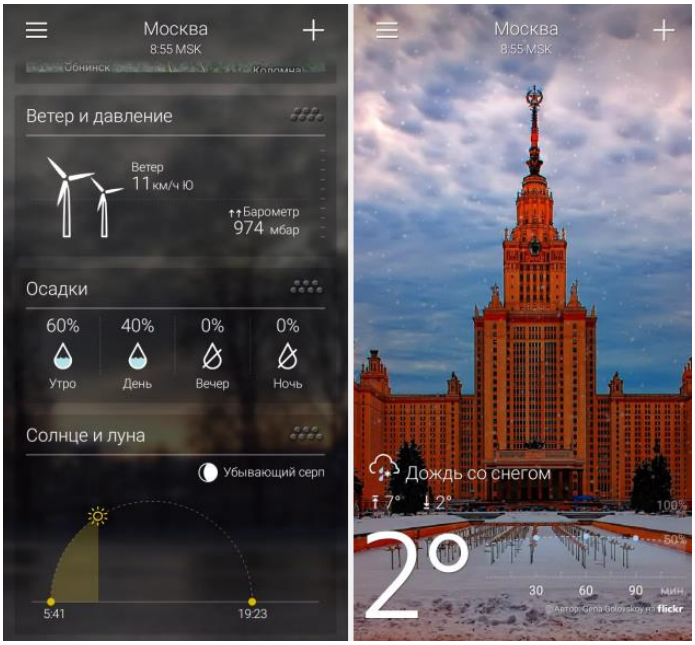


Рисунок 3 – Экран приложения «Yahoo погода»

Анализ показал, что каждое из рассмотренных приложений имеет свои сильные и слабые стороны. Это понимание позволило определить направления для совершенствования. В связи с этим было принято решение создать конкурентоспособное мобильное приложение, которое будет отличаться удобством использования и обладать новыми функциями.

Выводы по разделу 1

В первой части были рассмотрены функциональные и архитектурные особенности мобильных приложений, предназначенных для прогнозирования погодных условий.

В соответствии с классификацией FURPS+ были определены функциональные и нефункциональные требования к разрабатываемому мобильному приложению.

Анализ существующих аналогов мобильных приложений позволил выявить ключевые характеристики и проверить их соответствие разработанным требованиям.

2 Проектирование мобильного приложения

2.1 Выбор средств разработки мобильного приложения

Чтобы проект был реализован успешно, необходимо провести детальный анализ доступных фреймворков разработки и выбрать наиболее оптимальный. При выборе следует учитывать такие параметры, как кроссплатформенность, производительность, скорость разработки, доступность библиотек и компонентов, а также простоту разработки и поддержки. Рассмотрим три популярных кроссплатформенных фреймворка.

React Native — это фреймворк, разработанный компанией Facebook [8]. Он часто применяется в мобильной разработке. Его ключевыми особенностями являются доступ к нативным API устройств через специальные JavaScript-мосты и поддержка сообществом с большим количеством готовых компонентов [9].

Xamarin — это инструмент для разработки мобильных приложений, разработанный корпорацией Microsoft. Он предлагает два подхода к созданию приложений: «Xamarin Forms позволяет создать пользовательский интерфейс, который будет работать на разных устройствах, а Xamarin Native позволяет создать универсальный отдельный интерфейс для каждой платформы, используя общую бизнес-логику» [10].

Flutter — это фреймворк, разработанный компанией Google. Он позволяет создавать красивые и высокопроизводительные мобильные приложения с использованием языка Dart [11]. Flutter обладает широким выбором готовых виджетов, что значительно упрощает настройку интерфейса. Flutter имеет функцию «горячей перезагрузки», которая позволяет быстро вносить изменения в код и видеть результаты в режиме реального времени.

Проведем сравнение фреймворков в таблице 2 и определим наиболее подходящий.

Таблица 2 – Сравнительный анализ фреймворков

Критерий	Flutter	React Native	Xamarin
Язык программирования	Dart	JavaScript	C#
Платформы	Android, iOS, Web, Desktop (Windows, macOS, Linux)	Android, iOS, Web	Android, iOS, macOS, Windows
Производительность	Высокая	Средняя	Средняя
Разработка UI	Относительно простая, благодаря богатству виджетов	Средняя	Средняя
Размер приложения	Маленький	Средний	Средний
Скорость разработки	Высокая	Средняя	Средняя
Инструменты и IDE	Flutter Dev Tools, Android Studio, Visual Studio Code	Expo, React Native CLI	Visual Studio, Visual Studio Code

Учитывая все факторы, Flutter выделяется среди других фреймворков благодаря своей высокой производительности, скорости разработки и возможности создания красивых интерфейсов. Выбор в его пользу позволит создать качественное и современное приложение с минимальными затратами времени и ресурсов.

В качестве операционной системы была выбрана платформа Android, так как она является открытой, более распространенной и гибкой.

Для разработки была выбрана среда Android Studio, которая предоставляет полный набор инструментов для создания, тестирования и отладки приложений.

Для доступа к данным о погоде необходимо выбрать подходящий API (Application Programming Interface). API служит для связи между приложением и источником информации, предоставляя структурированный доступ к данным о погоде. Существует множество API, которые предлагают разные наборы данных, точность прогнозов и условия использования и т.д. Поэтому в далее представлен обзор, сравнение и выбор наиболее популярных и известных API, стабильно используемых в мобильных приложениях для прогнозирования погоды.

OpenWeatherMap – это популярный сервис с множеством возможностей использования метеорологических данных. Он подходит как для простых проектов, так и для сложных систем. «Данные о погоде для любой даты и времени за 46 с лишним лет в историческом архиве» [4].

AccuWeather – это коммерческий сервис с детализированными прогнозами на 90 дней для веб-сайтов, телевидения, радио и мобильных приложений. «Результаты поиска по городам всегда будут отображаться в порядке убывания значимости, при этом самые важные города будут перечислены в начале списка» [5].

Weatherbit.io – это простая и мощная платформа на базе машинного обучения. Данный API гарантирует высокую скорость обновления и большое время безотказной работы, которые можно проверить на официальном сайте в разделе состояния. «Источниками данных являются станции METAR из системы сбора NOAA, метеорологические радары, спутники, система мезомасштабного анализа в режиме реального времени» [6].

Чтобы определить наиболее подходящий API, сравним в таблице 3 данные варианты, опираясь на несколько важных критериев.

Таблица 3 – Сравнительный анализ API

Характеристика	OpenWeatherMap	AccuWeather	Weatherbit.io
Точность прогнозов	Средняя	Высокая	Средняя
Набор данных	Широкий	Широкий	Широкий
Исторические данные	Платно	Доступны	Платно
Гео покрытие	Глобальное	Глобальное	Глобальное
Стоимость	Бесплатно/Платно	Платно	Платно
Документация	Хорошая	Средняя	Средняя
Простота интеграции	Простая	Средняя	Средняя

Проведенный анализ позволил выявить, что OpenWeatherMap представляет собой наиболее оптимальное решение для разработки мобильного приложения. Данный сервис предоставляет все необходимые данные и легко интегрируется. Также OpenWeatherMap предлагает бесплатный тарифный план с ограничениями по частоте запросов, что вполне достаточно для разработки приложения.

2.2 Разработка диаграммы вариантов использования мобильного приложения

Диаграмма вариантов использования представляет собой удобный инструмент для демонстрации функциональных требований в виде схемы. На таких схемах используются различные графические элементы, линии связи и границы системы, чтобы легко обозначить роли пользователей и их действия. Этот инструмент позволяет более глубоко понять требования, что, в свою очередь, способствует его успешной разработке. Кроме того, диаграмма вариантов помогает оценить соответствие приложения потребностям пользователей. На рисунке 4 показана диаграмма вариантов использования

мобильного приложения для прогноза погоды, разработанная с учетом основных функций приложения.

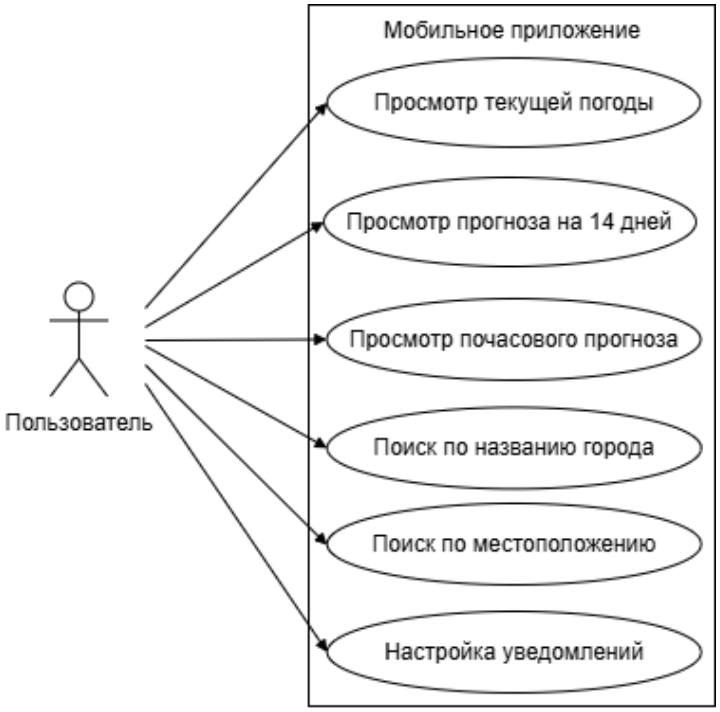


Рисунок 4 – Диаграмма вариантов использования

2.3 Выбор архитектуры мобильного приложения

Выбор подходящей архитектуры является важным этапом при создании масштабируемого и тестируемого приложения. Архитектура определяет то, как устроено приложение, как распределены его обязанности между компонентами и как они взаимодействуют друг с другом. Существуют три наиболее распространённых архитектурных паттерна: MVC (Model-View-Controller), MVP (Model-View-Presenter) и MVVM (Model-View-View Model).

Для разработки мобильного приложения был выбран архитектурный шаблон MVVM [12] в связи упрощением тестирования за счет модульности, уменьшением объема шаблонного кода для обработки обновления данных, удобством реализации во Flutter с применением дополнительных полезных библиотек и пакетов для маршрутизации и работы с асинхронными

операциями, а также поддержкой масштабируемости для добавления новых функций и экранов в будущем.

На рисунке 5 изображена схема взаимодействия модулей паттерна MVVM. Этот шаблон разделяет приложения на три отдельных слоя:

Модель (Model) – этот слой отвечает за логику и данные, которые составляют основу приложения. Он может включать в себя операции с данными, их хранение и управление состоянием;

Модель представления (View Model) – слой соединения представления с моделью, обеспечивая обмен данными и предоставление логики;

Представление (View) – этот компонент отображает данные в виде пользовательского интерфейса. Именно с ним взаимодействует пользователь.

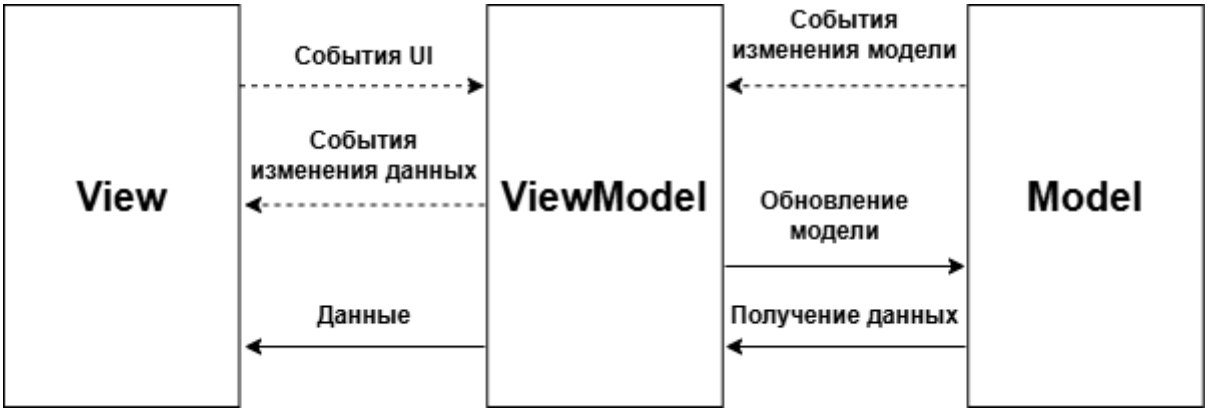


Рисунок 5 – Схема взаимодействия модулей архитектурного паттерна MVVM

Представленная диаграмма на рисунке 6 построена на основе схемы архитектурного паттерна MVVM. Она позволяет лучше понять структуру и функциональность системы.

В блоке представления (View) расположены экраны для взаимодействия с пользователем:

– главный экран отображает основные параметры погоды и взаимодействует с главным контроллером;

– экран прогноза погоды на 14 дней представляет прогноз в виде списка карточек дней и информативный блок о погоде. Взаимодействует с контроллером дней и главным контроллером;

– экран поиска городов предоставляет интерфейс пользователя для ввода и отображения результатов. Он связан с главным контроллером для работы с сервисом поиска городов;

– экран загрузки отвечает за предварительную загрузку данных, проверку разрешений и доступа к геолокации. Текущий экран функционирует с соответствующим сервисом инициализации;

– экран настроек необходим для отображения и настройки пунктов уведомлений. Данные передаются из сервиса уведомлений через главный контроллер.

Блок модели представления (View Model) включает в себя:

– главный контроллер погоды отвечает за управление данными, полученными из слоя Model, их обработку и предоставление экранам. Также он обрабатывает пользовательский ввод и обновляет данные;

– контроллер дней служит для форматирования передаваемых данных и распределения изображений погодных условий;

– сервис инициализации является небольшим навигационным компонентом, который проверяет наличие данных о местоположении, используя главный контроллер и принимает решение для перехода на главный экран или экран поиска городов;

Блок модели (Model) содержит следующие основные данные и функциональные сервисы:

– данные для поиска городов;

– данные о погоде;

– сервис API;

– сервис местоположения;

– сервис поиска городов;

– сервис уведомлений;

– сервис кэширования; этот сервис предназначен для обеспечения доступности данных в случае отсутствия соединения с интернетом, а

также для ускорения загрузки данных при повторных запусках приложения.

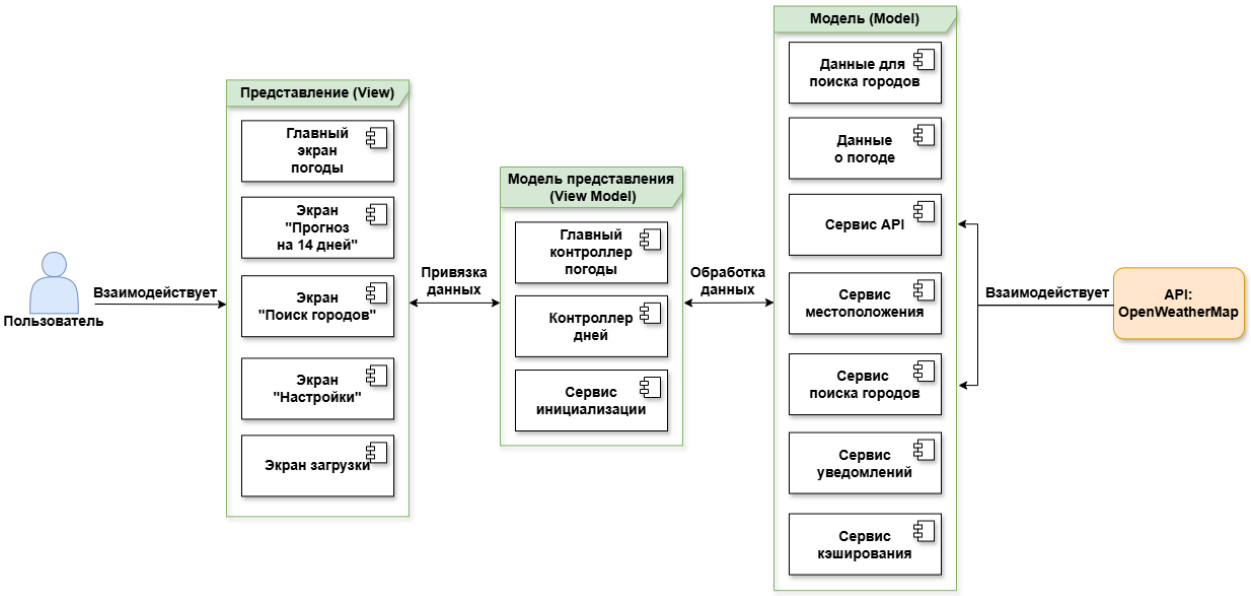


Рисунок 6 – Диаграмма компонентов архитектуры мобильного приложения

2.4 Разработка пользовательского интерфейса мобильного приложения

Разработка пользовательского интерфейса (UI) мобильного приложения «Прогноз погоды» должна основываться на принципах, которые помогут пользователю легко взаимодействовать с приложением. Разберем эти принципы и требования к ним.

Принцип интуитивно понятного и доступного интерфейса: UI должен быть достаточно простым, чтобы пользователи могли легко находить нужную информацию и выполнять свои задачи. Важно наличие оптимального количества элементов и узнаваемых графических и текстовых обозначений, чтобы пользователи не путались.

Принцип структурности интерфейса: вся информация должна быть представлена логично и удобно. Пользователь должен легко переходить от одной задачи к другой без лишних действий. Элементы и кнопки должны быть расположены в базовых местах для быстрого ориентира пользователей.

Принцип привлекательности: UI должен быть привлекательным, сбалансированным и гармоничным. Все шрифты и элементы должны быть заметными. Необходимо использовать единую цветовую гамму, которая не будет отвлекать от текста и соответствовать темам разделов [14].

В соответствии с запланированными экранами при составлении архитектуры приложения были созданы прототип главного экрана и набор иконок при помощи популярного графического редактора Adobe Photoshop. Данные элементы изображены на рисунках 7 и 8. Следует отметить, что окончательный вид приложения может отличаться от первоначального замысла.

Рисунок 7 – Прототип интерфейса главного экрана

Рисунок 8 – Набор иконок

Выводы по разделу 2

В данном разделе был выбран фреймворк Flutter, OpenWeatherMap API, Android Studio и платформа Android для разработки мобильного приложения.

Была создана диаграмма вариантов использования. Выбран шаблон MVVM, который позволяет разделить логику, данные и пользовательский интерфейс. Для наглядной демонстрации структуры приложения была спроектирована и описана диаграмма компонентов архитектуры.

Также был разработан прототип главного экрана с помощью графического инструмента Adobe Photoshop. Это позволило определить предполагаемое расположение элементов.

3 Реализация мобильного приложения

3.1 Разработка блока Model

В файле данных о погоде содержится код, который описывает структуру данных, необходимую для получения и обработки информации. Этот код является основой для шифрования сведений о текущих погодных условиях и прогнозах, а также для удобного представления данных пользователю.

Данные легко преобразуются из формата JSON в объекты Dart и обратно с помощью библиотеки convert [15]. Это упрощает работу с OpenWeatherMap API и делает взаимодействие с данными более гибким благодаря динамическим типам.

Основной класс data model служит основой для предоставления базовых данных. Он содержит информацию о местоположении, текущих погодных условиях, прогнозе на несколько дней и других необходимых параметрах.

Класс days содержит погодные условия на каждый день.

Класс hours хранит информацию на конкретный час.

Класс stations содержит информацию о метеостанциях, таких как VIAR и OPLA, что позволяет получать данные о качестве и расположении метеостанций.

Класс viar представляет метеостанцию с такими параметрами, как расстояние, координаты и качество данных. На рисунке 9 продемонстрирована реализация данного класса.

Класс current conditions содержит текущих погодных условий, включая температуру, влажность, скорость ветра и другие параметры.

```dart
class VIAR {
  dynamic distance;
  dynamic latitude;
  dynamic longitude;
  dynamic useCount;
  dynamic id;
  dynamic name;
  dynamic quality;
  dynamic contribution;

  VIAR(
      {this.distance,
        this.latitude,
        this.longitude,
        this.useCount,
        this.id,
        this.name,
        this.quality,
        this.contribution});

  VIAR.fromJson(Map<String, dynamic> json) {
    distance = json['distance'];
    latitude = json['latitude'];
    longitude = json['longitude'];
    useCount = json['useCount'];
    id = json['id'];
    name = json['name'];
    quality = json['quality'];
    contribution = json['contribution'];
  }

  Map<String, dynamic> toJson() {
    final Map<String, dynamic> data = <String, dynamic>{};
    data['distance'] = distance;
    data['latitude'] = latitude;
    data['longitude'] = longitude;
    data['useCount'] = useCount;
    data['id'] = id;
    data['name'] = name;
    data['quality'] = quality;
    data['contribution'] = contribution;
    return data;
  }
}
```

Рисунок 9 – Класс VIAR

Файл с данными для поиска городов содержит названия, координаты, страну и регион. Эти данные получаются в результате поиска и преобразуются в JSON-формате, получаемыми из API или хранящимися локально. Модель данных содержит фабричный метод fromJson для создания объекта с обработкой отсутствующих значений и конвертацией типов. Чтобы сохранить, передать и превратить объект в представление используется метод toJson.

```dart
import 'dart:convert';

// Преобразует JSON-строку в список объектов SearchCityData, возвращает список городов
List<SearchCityData> searchCityDataFromJson(String str) =>
    List<SearchCityData>.from(
        json.decode(str).map((x) => SearchCityData.fromJson(x)));

// Преобразует список объектов SearchCityData в JSON-строку, возвращает JSON-представление данных
String searchCityDataToJson(List<SearchCityData> data) =>
    json.encode(List<dynamic>.from(data.map((x) => x.toJson())));

// Модель данных для представления информации о городе
class SearchCityData {
  String name;               // Основное название города
  double lat;               // Широта местоположения
  double lon;               // Долгота местоположения
  String country;           // Страна расположения
  String state;             // Регион/область расположения

  // Конструктор модели данных города
  SearchCityData({
    required this.name,      // Обязательное название города
    required this.lat,      // Обязательная широта
    required this.lon,      // Обязательная долгота
    required this.country,  // Обязательное название страны
    required this.state,    // Обязательное название региона
  });

  // Фабричный метод для создания объекта из JSON, возвращает объект с данными города
  factory SearchCityData.fromJson(Map<String, dynamic> json) {
    return SearchCityData(
      name: json["name"] ?? "",  // Значение по умолчанию, если name отсутствует
      lat: json["lat"]?.toDouble() ?? 0.0,  // Конвертация в double с значением по умолчанию
      lon: json["lon"]?.toDouble() ?? 0.0,  // Конвертация в double с значением по умолчанию
      country: json["country"] ?? "",       // Значение по умолчанию для страны
      state: json["state"] ?? "",           // Значение по умолчанию для региона
    );
  }

  // Преобразует объект в JSON-представление, возвращает данные города
  Map<String, dynamic> toJson() => {
    "name": name,
    "lat": lat,
    "lon": lon,
    "country": country,
    "state": state,
  };
}
```

Рисунок 10 – Код «search_city_data.dart»

Чтобы интегрировать OpenWeatherMap нужен API-ключ. Для этого была пройдена регистрация на официальном сайте openweathermap.org [4]. В разделе управления автоматически был сгенерирован API-ключ, который расположен в файле «constants.dart» для безопасности. Сервис отправляет HTTP GET-запрос к API и в случае успешного выполнения запроса он расшифровывает JSON-ответ и возвращает его. Если возникла ошибка при запросе данных или отсутствия интернет-соединения, генерируется исключение с сообщением об ошибке. Метод readResponse служит для правильной обработки кодировки UTF-8. Для управления строками и сообщениями в приложении используется отдельный файл «AppStrings.dart» с константами для различных текстовых элементов.

Рисунок 11 отображает цельный код сервиса API. В коде использовались следующие дополнительные необходимые библиотеки и пакеты:

– «async» для работы с асинхронными операциями [16];

– «io» для работы с сетью и обработкой исключений, связанных с интернет-соединением [17];

– «convert» для преобразований и кодировки;

– «http» для выполнения запросов [18].

```dart
import 'dart:async';
import 'dart:convert';
import 'dart:io';
import 'package:weather/AppStrings.dart';
import 'package:http/http.dart' as http;

abstract class BaseApiServices {
  Future<dynamic> getApi(String url);
}

class ApiServices extends BaseApiServices {
  @override
  Future getApi(String url) async {
    var jsonData;
    try {
      var response = await http.get(Uri.parse(url));

      if (response.statusCode == HttpStatus.ok) {
        var responseBody = await readResponse(response);
        jsonData = jsonDecode(responseBody);
      } else {
        throw Exception('${AppStrings.errorWhileCommunication} ${response.statusCode}');
      }
    } on SocketException {
      throw Exception(AppStrings.internetConnectionError);
    }
    return jsonData;
  }

  Future<String> readResponse(http.Response response) async {
    return utf8.decode(response.bodyBytes);
  }
}
```

Рисунок 11 – Код сервиса API

Для реализации сервиса геолокации (местоположения) были задействованы пакеты geocoding и geolocator. На рисунке 12 отображена полная реализация этого сервиса, включающая в себя несколько таких методов, как: checkPermission для проверки разрешений на доступ к геолокации, requestPermission для запроса разрешений на доступ, getAdressFromPosition для получения адреса на основе координат широты и долготы, getLocation для получения текущего местоположения устройства с таймаутом в 10 секунд для обеспечения надежности и отзывчивости, getAddressFromCoordinate для преобразования координат в адрес.

```dart
import 'package:geocoding/geocoding.dart';
import 'package:geolocator/geolocator.dart';

class LocationService {
  Future<Position> getLocation() async {
    try {
      // Настройки запроса местоположения
      final LocationSettings locationSettings = LocationSettings(
        accuracy: LocationAccuracy.high,
        timeLimit: Duration(seconds: 10),
      );
      // Получение текущей позиции
      Position position = await Geolocator.getCurrentPosition(
        locationSettings: locationSettings,
      );
      return position;
    } catch (e) {
      rethrow;
    }
  }

/// Преобразует координаты из объекта Position в читаемый адрес
  Future<String> getAddressFromPosition(Position position) async {
    try {
      // Геокодирование координат в адрес
      List<Placemark> placemarks = await placemarkFromCoordinates(
          position.latitude, position.longitude);

      Placemark place = placemarks[0]; // Используем первый результат геокодирования

      return "${place.locality}, ${place.subLocality}"; // Форматируем адрес из компонентов
    } catch (e) {
      return "none";
    }
  }

// Преобразует координаты в читаемый адрес
  Future<String> getAddressFromCoordinate(double latitude, double longitude) async {
    try {
      List<Placemark> placemarks = await placemarkFromCoordinates(
          latitude, longitude);

      Placemark place = placemarks[0];

      return "${place.locality}, ${place.subLocality}";
    } catch (e) {
      return "none";
    }
  }

// Проверяет текущие разрешения на доступ к геоданным
// [Возвращает]: true если разрешение есть (always или whileInUse)
  Future<bool> checkPermission() async {
    LocationPermission permission = await Geolocator.checkPermission();
    return permission == LocationPermission.always ||
        permission == LocationPermission.whileInUse;
  }

// Запрашивает разрешение на доступ к геоданным
// [Возвращает]: true если пользователь дал разрешение
  Future<bool> requestPermission() async {
    LocationPermission permission = await Geolocator.requestPermission();
    return permission == LocationPermission.always ||
        permission == LocationPermission.whileInUse;
  }
}
```

Рисунок 12 – Код сервиса местоположения

В файле «search_city_service.dart» реализована функция, которая позволяет пользователю искать города по названию и получать список возможных вариантов с использованием OpenWeatherMap API. На рисунке 13 полностью показан код этого сервиса.

Алгоритм работы данной функции:

– формирование URL-адреса API на основе названия города и ключа API;

– отправка GET-запроса к API с использованием библиотеки HTTP.

– проверка успешности выполнения запроса;

– в случае успешного выполнения запроса ответ в формате JSON преобразуется в список объектов;

– в случае неудачного выполнения запроса генерируется исключение с сообщением об ошибке;

– метод возвращает список найденных городов, представленных в виде объектов файла «search_city_data.dart».

```dart
import 'dart:convert';
import 'package:weather/AppStrings.dart';
import 'package:weather/model/search_city_data.dart';
import 'package:weather/constants.dart';
import 'package:http/http.dart' as http;

class SearchCityService {
  Future<List<SearchCityData>> searchCity(String cityName) async {
    final String apiUrl =
        'http://api.openweathermap.org/geo/1.0/direct?q=$cityName&limit=5&appid=$WEATHER_API_KEY&lang=ru';

    final response = await http.get(Uri.parse(apiUrl));
    if (response.statusCode == 200) {
      final List<dynamic> jsonResponse = json.decode(response.body);
      List<SearchCityData> searchResults = jsonResponse
          .map((data) => SearchCityData.fromJson(data))
          .toList();
      return searchResults;
    } else {
      throw Exception(AppStrings.failedLoadSearchResults);
    }
  }
}
```

Рисунок 13 – Код сервиса для поиска городов

Сервис уведомлений реализован с использованием следующих библиотек: «Flutter local notifications» для отображения локальных уведомлений, «Work Manager» для выполнения периодических задач в фоновом режиме, «Shared Preferences» для сохранения предпочтений пользователя и предоставление настраиваемых уведомлений.

В процессе инициализации регистрируется периодическая задача, которая выполняется каждый час при наличии подключения к интернету. Метод updateWeatherData получает погодные данные из главного контроллера и метод checkWeatherConditions проверяет их на соответствие настройкам уведомлений. После этого обработчик фоновых задач callbackDispatcher вызывает метод для проверки погодных условий. В случае совпадения условий формируется сообщение и уведомление отображается. Для того, чтобы предотвратить отправку одного и того же типа уведомления более трех

раз в сутки, был создан счетчик и метод, который проверяет прошло ли 24 часа с последней проверки. На рисунке 14 отображена реализация метода проверки уведомлений и счетчика. Вывод уведомления показан на рисунке 15.

Таким образом данный сервис предоставляет надежную систему оповещений о погоде, которая работает даже при закрытом приложении, не нагружает систему и легко расширяется для новых типов уведомлений.

```dart
// Проверка возможности показа уведомления
static Future<bool> _canShowNotification(String notificationType) async {
  final prefs = await SharedPreferences.getInstance();
  final now = DateTime.now().toUtc();
  final notificationKey = '${notificationType}_count';
  final lastCheckKey = '${notificationType}_last_check';

  final lastCheck = prefs.getInt(lastCheckKey) ?? 0;
  final lastCheckDate = DateTime.fromMillisecondsSinceEpoch(lastCheck, isUtc: true);

  if (now.difference(lastCheckDate).inHours >= 24) {
    await prefs.setInt(notificationKey, 0);
    await prefs.setInt(lastCheckKey, now.millisecondsSinceEpoch);
    return true;
  }

  final count = prefs.getInt(notificationKey) ?? 0;
  return count < _maxNotificationsPerDay;
}

// Счетчик уведомлений
static Future<void> _incrementNotificationCount(String notificationType) async {
  final prefs = await SharedPreferences.getInstance();
  final notificationKey = '${notificationType}_count';
  final lastCheckKey = '${notificationType}_last_check';

  final count = prefs.getInt(notificationKey) ?? 0;
  await prefs.setInt(notificationKey, count + 1);
  await prefs.setInt(lastCheckKey, DateTime.now().toUtc().millisecondsSinceEpoch);
}
```

Рисунок 14 – Метод проверки уведомлений и метод счетчика

Рисунок 15 – Вывод уведомления

На рисунке 16 представлен полный код службы кэширования, реализованной с использованием пакета Shared Preferences. Эта служба предназначена для хранения данных в локальном хранилище устройства в формате пар «ключ-значение». Локальное хранилище представляет собой файл, который находится в приватной директории приложения операционной системы устройства. Сервис предоставляет методы для сохранения и извлечения данных о погоде в формате JSON-строк, чтобы обеспечить быстрый доступ к актуальным последним прогнозам при отсутствии подключения к интернету. Метод cacheWeatherData принимает объект DataModel под ключом weatherCacheKey, что позволяет кэшировать основные данные о погоде. Метод cacheCity кэширует информацию о последнем выбранном городе. Дополнительно реализована автоматическая очистка в случае обнаружения поврежденных данных, что гарантирует надежную работу приложения.

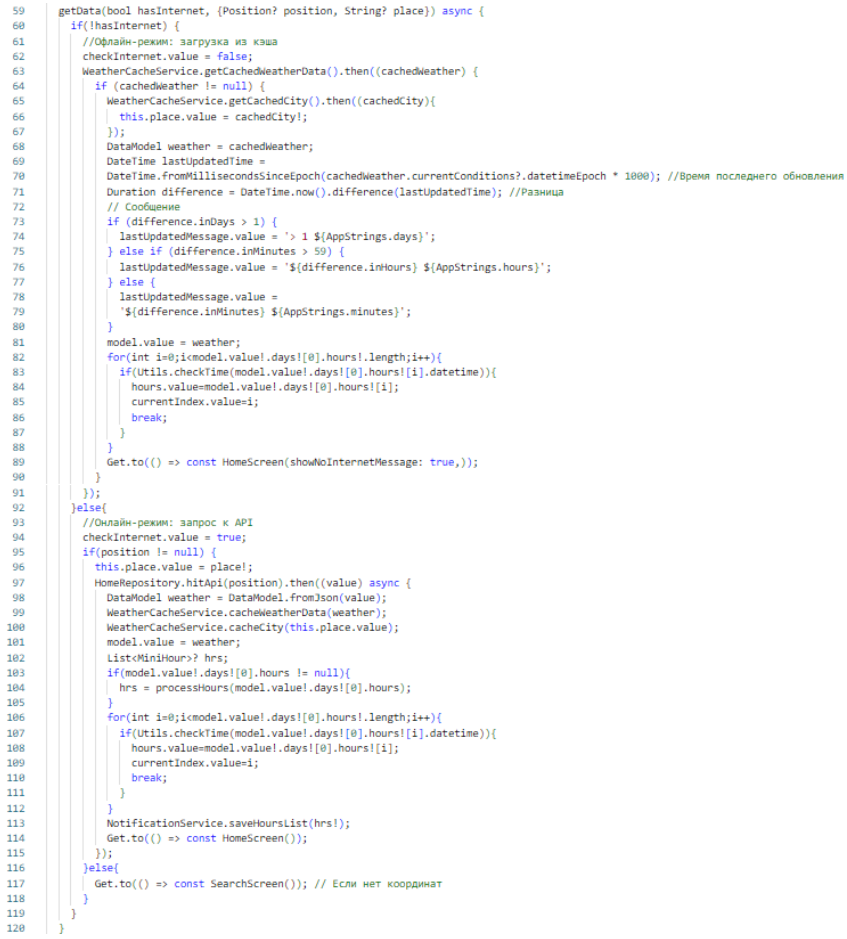
```dart
import 'dart:convert';
import 'package:shared_preferences/shared_preferences.dart';
import 'package:weather/model/data_model.dart';

class WeatherCacheService {
  // Ключ для кэширования
  static const _weatherCacheKey = 'cached_weather_data';
  static const _cityCacheKey = 'cached_city_name';

  // Сохранение погодных данных в кэш
  static Future<void> cacheWeatherData(DataModel weatherData) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString(
      _weatherCacheKey,
      jsonEncode(weatherData.toJson()),
    );
  }

  // Получение кешированных погодных данных
  static Future<DataModel?> getCachedWeatherData() async {
    final prefs = await SharedPreferences.getInstance();
    final cachedData = prefs.getString(_weatherCacheKey);

    if (cachedData != null) {
      try {
        return DataModel.fromJson(jsonDecode(cachedData));
      } catch (e) {
        await prefs.remove(_weatherCacheKey); // Удаляем битые данные
        return null;
      }
    }
    return null;
  }

  // Очистка кэша
  static Future<void> clearCache() async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.remove(_weatherCacheKey);
  }

  static Future<void> cacheCity(String cityName) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString(_cityCacheKey, cityName);
  }

  // Получить кешированный город
  static Future<String?> getCachedCity() async {
    final prefs = await SharedPreferences.getInstance();
    return prefs.getString(_cityCacheKey);
  }

  // Очистить кэш города
  static Future<void> clearCityCache() async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.remove(_cityCacheKey);
  }
}
```

Рисунок 16 – Код сервиса кэширования

3.2 Разработка блока View Model

Основной метод главного контроллера, отвечающий за получение данных о погоде, называется getData. На рисунке 17 показана часть метода. Он проверяет наличие интернет-соединения. Если оно отсутствует, то метод пытается загрузить данные из кэша. Также контроллер проверяет, когда данные были обновлены и вычисляет разницу между текущем временем и временем последнего обновления. Если интернет доступен и известно местоположение, вызывается метод для получения данных о погоде из API. Если интернет есть, но местоположение не известно, происходит перенаправление на экран поиска. После получения данных контроллер обновляет состояние и передает на главный экран.

Для обеспечения плавности пользовательского интерфейса в контроллере реализована система анимированного переключения между временными слотами прогноза. Дополнительно механизм использует таймеры для управления состоянием анимации, что позволяет достичь визуально приятного эффекта перехода без излишней нагрузки на систему.



```
getData(bool hasInternet, {Position? position, String? place}) async {
  if(!hasInternet) {
    //Офлайн-режим: загрузка из кэша
    checkInternet.value = false;
    WeatherCacheService.getCachedWeatherData().then((cachedWeather) {
      if (cachedWeather != null) {
        WeatherCacheService.getCachedCity().then((cachedCity){
          this.place.value = cachedCity!;
        });
        DataModel weather = cachedWeather;
        DateTime lastUpdatedTime =
        DateTime.fromMillisecondsSinceEpoch(cachedWeather.currentConditions?.datetimeEpoch * 1000); //Время последнего обновления
        Duration difference = DateTime.now().difference(lastUpdatedTime); //Разница
        // Сообщение
        if (difference.inDays > 1) {
          lastUpdatedMessage.value = '> 1 ${AppStrings.days}';
        } else if (difference.inMinutes > 59) {
          lastUpdatedMessage.value = '${difference.inHours} ${AppStrings.hours}';
        } else {
          lastUpdatedMessage.value =
          '${difference.inMinutes} ${AppStrings.minutes}';
        }
        model.value = weather;
        for(int i=0;i<model.value!.days![0].hours!.length;i++){
          if(Utils.checkTime(model.value!.days![0].hours![i].datetime)){
            hours.value=model.value!.days![0].hours![i];
            currentIndex.value=i;
            break;
          }
        }
        Get.to(() => const HomeScreen(showNoInternetMessage: true,));
      }
    });
  }else{
    //Онлайн-режим: запрос к API
    checkInternet.value = true;
    if(position != null) {
      this.place.value = place!;
      HomeRepository.hitApi(position).then((value) async {
        DataModel weather = DataModel.fromJson(value);
        WeatherCacheService.cacheWeatherData(weather);
        WeatherCacheService.cacheCity(this.place.value);
        model.value = weather;
        List<MiniHour>? hrs;
        if(model.value!.days![0].hours != null){
          hrs = processHours(model.value!.days![0].hours);
        }
        for(int i=0;i<model.value!.days![0].hours!.length;i++){
          if(Utils.checkTime(model.value!.days![0].hours![i].datetime)){
            hours.value=model.value!.days![0].hours![i];
            currentIndex.value=i;
            break;
          }
        }
        NotificationService.saveHoursList(hrs!);
        Get.to(() => const HomeScreen());
      });
    }else{
      Get.to(() => const SearchScreen()); // Если нет координат
    }
  }
}
```

Рисунок 17 – Код метода getData

Контроллер дней обеспечивает детализированное управление многодневным прогнозом. Он взаимодействует с главным контроллером и отвечает за хранение и обновление информации о выбранном дне, выбор соответствующего изображения погоды для отображения. При разработке были использованы реактивные переменные. Например, currentDay = 0.obs это реактивная переменная типа RxInt, которая хранит индекс выбранного дня в списке прогнозов. Начальное значение переменной равно 0, что соответствует первому дню по умолчанию. Использование obs (сокращение от «observable» – наблюдаемый) делает эту переменную наблюдаемой, то есть любые изменения её значения автоматически вызывают обновление интерфейса и уведомлений без необходимости полной перезагрузки данных. Метод setDay обеспечивает плавное переключение между днями, синхронизируя состояние с другими компонентами системы. Функционал преобразования дат включает несколько форматов отображения: полное представление даты, название

месяца, число месяца. Для температурных показателей реализовано форматирование с добавлением символа градуса и округлением до целых значений. Реализация учитывает требования к производительности, минимизируя количество операций при обновлении данных. Фрагмент кода контроллера представлен на рисунке 18.

```dart
class DaysController extends GetxController {
  /// Индекс текущего выбранного дня
  RxInt currentDay = 0.obs;

  /// Ссылка на главный контроллер погоды
  final homeController = Get.put(HomeController());

  /// Текущий день с погодными данными
  late Rx<Days> day;

  /// Конструктор инициализирует данные первого дня
  DaysController() {
    day = homeController.model.value!.days![0].obs;
  }

  /// Устанавливает выбранный день по индексу
  void setDay(int index) {
    day.value = homeController.model.value!.days![index];
    currentDay.value = index;
  }
```

Рисунок 18 – Фрагмент кода контроллера дней

Для того чтобы приложение могло перейти от стартовой загрузочной страницы к основному контенту, разработана специализированная служба инициализации. В случае, если есть данные о местоположении, этот сервис проверяет и обрабатывает их, используя главный контроллер. Если же данных нет, сервис перенаправляет пользователя на экран поиска города с помощью метода get to. Этот сервис отделен, чтобы избежать дублирование. На рисунке 19 представлен код работы данного сервиса.

```dart
import 'package:geolocator/geolocator.dart';
import 'package:get/get.dart';
import 'package:weather/view/search/search_screen.dart';
import 'package:weather/view_model/controllers/home_controller.dart';

class SplashServices{
  static void getApiData(bool hasInternet, {Position? position, String? place}){
    if(place != "")
    {
        final controller=Get.put(HomeController());
        controller.getData(hasInternet, position: position, place: place);
    }else{
        Get.to(()=>SearchScreen());
    }
  }
}
```

Рисунок 19 – Код сервиса инициализации

3.3 Разработка блока View

Главная страница мобильного приложения — это первое место, с которым взаимодействует пользователь. Она представлена на рисунке 20.

Благодаря пакету «Material» [25] удалось значительно сократить время разработки, так как он предлагает множество готовых элементов и стилей, которые помогают создавать плавные анимации и единый интерфейс.

Структура главного экрана организована в файле «home_screen.dart», который отвечает за взаимодействие со всеми компонентами приложения. Если у пользователя нет интернет-соединения, на экране отображается сообщение: «Отсутствует соединение с интернетом. Последнее обновление:», полученное от главного контроллера.

Виджет «location.dart» показывает название города, дату и день недели. При нажатии на этот виджет пользователь может перейти к экрану поиска городов или к экрану настроек. Это реализовано с помощью виджета Gesture Detector, который отслеживает нажатия на элементы, например, на изображения.

Основная карточка «info_card.dart» содержит информацию о текущей погоде, ощущаемой температуре, описании погоды и времени. Для плавного перехода к подробной информации мы используем класс Hero из библиотеки Dart, что делает анимацию более привлекательной. Часть реализации этой карточки показана на рисунке 21.

В файле «container_list.dart» содержатся небольшие контейнеры, которые отображают данные о облачности, скорости ветра и влажности. Их реализация представлена на рисунке 22.

Горизонтальный список, реализованный в файле «hours_list.dart» позволяет пользователю просматривать информацию о погоде на каждый час.

Рисунок 20 – Главный экран мобильного приложения

```
class InfoCard extends StatelessWidget {
  InfoCard({super.key});
 final controller=Get.put(HomeController());
 @override
 Widget build(BuildContext context) {
   var size=MediaQuery.sizeOf(context);
   return SizedBox(
     height: 240,
     width: size.width,
     child: Stack(
       alignment: Alignment.bottomCenter,
       children: [
         Container(
           height: 180,
           margin: const EdgeInsets.only(bottom: 30),
           width: size.width,
           decoration: BoxDecoration(
               borderRadius: BorderRadius.circular(20),
               gradient: LinearGradient(
                   begin: Alignment.topLeft,
                   end: Alignment.bottomRight,
                   colors: [
                     Colors.blue.withValues(alpha: .7),
                     Colors.blue,
                     Colors.blue,
                   ])),
         ),
         Positioned(
             top: 0,
             left: 10,
             child: Obx(()=>Image.asset(
               controller.getImage(controller.currentIndex.value),
               height: 150,
               width: 150,
               fit: BoxFit.fill,
             ))),
         Positioned(
           bottom: 50,
           left: 20,
           child: Column(
```

Рисунок 21 – Часть кода виджета основной карточки в файле «info_card.dart»

```dart
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'package:weather/AppStrings.dart';
import 'package:weather/res/images/image_assets.dart';
import 'package:weather/view_model/controllers/home_controller.dart';

class ContainerList extends StatelessWidget {
  ContainerList({super.key});
  final controller=Get.put(HomeController());
  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.symmetric(horizontal: 30),
      child: Row(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children: [
          Obx(() =>  SmallContainer(text: '${controller.getCloudOver()}%', image: ImageAssets.heavyRain, label: AppStrings.cloudiness,), ),
          Obx(() => SmallContainer(text : '${controller.getWindSpeed()} ${ AppStrings.speed}',image : ImageAssets.wind, label: AppStrings.wind,),),
          Obx(() =>  SmallContainer(text :'${controller.getHumidity()}%', image :ImageAssets.sun, label: AppStrings.humidity,),),
        ],
      ),
    );
  }
}
```

Рисунок 22 – Код «container_list.dart»

Экран прогноза погоды на ближайшие две недели взаимодействует с главным контроллером и контроллером дней. Рисунок 22 содержит снимок готового интерфейса этого экрана. Для удобства реализации экран был разделен на несколько частей:

– «app_bar.dart»: виджет кнопки возвращения на главный экран с помощью метода «navigator.pop»;

– «center_card.dart»: виджет с подробной информацией о погоде на выбранный день;

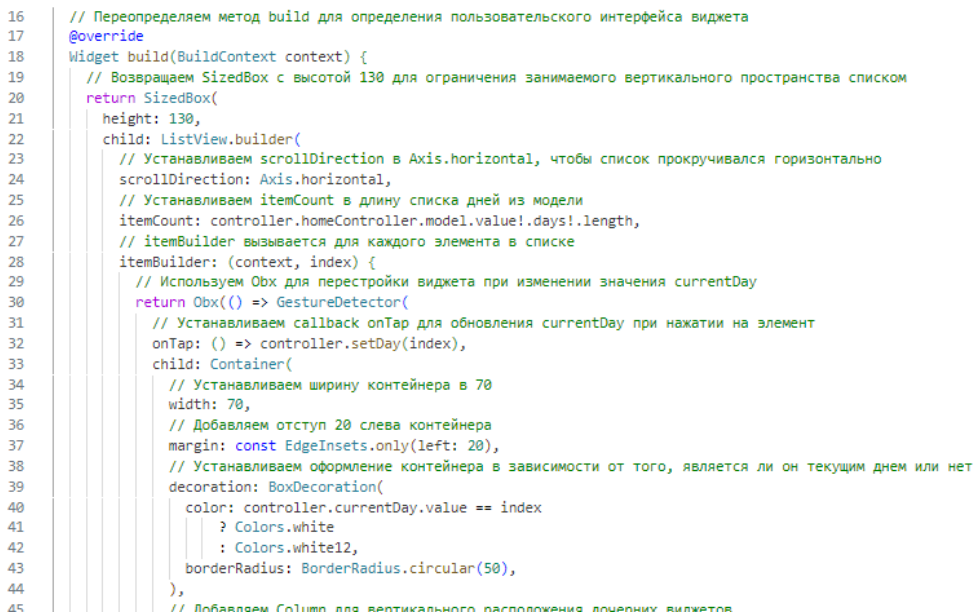
– «days_list.dart»: виджет, отображающий горизонтальный список дней для выбора. На рисунке 23 показан отрывок скриншота кода из этого файла с методами для отображения дня и месяца из контроллера, а на рисунке 24 метод построения части интерфейса;

– «small_container.dart»: виджет с дополнительной информацией о погоде (облачность, скорость ветра, влажность) в виде горизонтального списка.

Рисунок 22 – Экран «Прогноз погоды на 14 дней»

```
57  // Отображаем название месяца с помощью метода getMonth из контроллера
58  Text(
59    controller.getMonth(index),
60    style: TextStyle(
61      color: controller.currentDay.value == index
62          ? Colors.purple
63          : Colors.white,
64      fontSize: 30,
65      height: 0,
66      fontWeight: FontWeight.bold,
67    ),
68  ),
69  // Отображаем день месяца с помощью метода getMonthDay из контроллера
70  Text(
71    controller.getMonthDay(index),
72    style: TextStyle(
73      color: controller.currentDay.value == index
74          ? Colors.purple
75          : Colors.white,
76      fontSize: 15,
77      height: 0,
78      fontWeight: FontWeight.bold,
79    ),
```

Рисунок 23 – Методы для отображения дня и месяца из контроллера



```
// Переопределяем метод build для определения пользовательского интерфейса виджета
@override
Widget build(BuildContext context) {
  // Возвращаем SizedBox с высотой 130 для ограничения занимаемого вертикального пространства списком
  return SizedBox(
    height: 130,
    child: ListView.builder(
      // Устанавливаем scrollDirection в Axis.horizontal, чтобы список прокручивался горизонтально
      scrollDirection: Axis.horizontal,
      // Устанавливаем itemCount в длину списка дней из модели
      itemCount: controller.homeController.model.value!.days!.length,
      // itemBuilder вызывается для каждого элемента в списке
      itemBuilder: (context, index) {
        // Используем Obx для перестройки виджета при изменении значения currentDay
        return Obx(() => GestureDetector(
          // Устанавливаем callback onTap для обновления currentDay при нажатии на элемент
          onTap: () => controller.setDay(index),
          child: Container(
            // Устанавливаем ширину контейнера в 70
            width: 70,
            // Добавляем отступ 20 слева контейнера
            margin: const EdgeInsets.only(left: 20),
            // Устанавливаем оформление контейнера в зависимости от того, является ли он текущим днем или нет
            decoration: BoxDecoration(
              color: controller.currentDay.value == index
                  ? Colors.white
                  : Colors.white12,
              borderRadius: BorderRadius.circular(50),
            ),
            // Добавляем Column для вертикального расположения дочерних виджетов
```

Рисунок 24 – Метод build

Рисунок 25 представляет собой готовый экран настроек. При запуске страницы начальные настройки загружаются из сервиса уведомлений при помощи метода initState. Изменения настроек сохраняются асинхронно нажатием на иконку справа сверху страницы при помощи методов loadSettings и saveNotificationsSettings. Реализация этих методов представлена на рисунке 26. Переключатели работают с использованием виджета SwitchListTile для комбинирования его со списком. Страница представляет собой список из пяти условных уведомлений, таких как:

- напомнить, если температура станет ниже -5;
- напомнить, если температура станет ниже 0;
- напомнить, если температура станет выше 25;
- напомнить, если температура станет выше 30;
- оповестить о начале дождя.

Рисунок 25 – Экран «Настройки»

```dart
// Загрузка настроек
Future<void> _loadSettings() async {
  _notifications = await NotificationService.loadNotifications() ?? {
    'temp_below_5': false,
    'temp_below_0': false,
    'temp_above_25': false,
    'temp_above_30': false,
    'rain_alert': false,
  };
  setState(() {});
}

// Сохранение настроек
Future<void> _saveNotificationSettings() async {
  await NotificationService.saveNotifications(_notifications);
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(content: Text(AppStrings.notificationsSaved)),
  );
}
```

Рисунок 26 – Методы загрузки и сохранения настроек

Код экрана поиска городов реализует функциональность ввода названия города и получения результатов поиска с использованием нескольких компонентов. При открытии экрана создается объект уже существующего сервиса «search_sity_service.dart» и главного контроллера. В методе build создается основной интерфейс с заголовком и текстовым полем, а также добавляется виджет для отображения результатов поиска. На рисунке 27 представлен метод searchCity, при нажатии на кнопку которого происходит следующее: если текстовое поле пусто, показывается сообщение об ошибке; если введено название города, вызывается сервис поиска; если поиск успешен,

результаты обновляются и отображаются на экране. Ко всему прочему, предусмотрена проверка на ввод на русском языке с использованием регулярного выражения. Метод buildSearchResults отвечает за отображение списка найденных городов. Если поиск не был выполнен, отображается пустой центр. Для каждого найденного города формируется карточка с названием и дополнительной информацией о стране или регионе. При выборе элемента из списка вызывается метод получения данных с использованием координат. Весь функционал обеспечивает простую и динамичную форму. Рисунок 28 отображает интерфейс описанного выше экрана при вводе поиска города Новосибирска.

```dart
void _searchCity() async {
  String cityName = _searchController.text.trim();
  if (cityName.isEmpty) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text(AppStrings.searchEmptyError)),
    );
    return;
  }

  try {
    List<SearchCityData> results = await _searchCityService.searchCity(cityName);
    //Обновялем список результатов поиска
    setState(() {
      _searchResults.clear();
      _searchResults.addAll(results);
      _isSearchPerformed = true;
    });
  } catch (e) {
    //Ошибка при поиске
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text(AppStrings.errorSearchingCity)),
    );
  }
}

// Проверяем, введен ли текст на русском (хотя бы одна кириллическая буква)
bool _isInputRussian(String text) {
  final russianRegex = RegExp(r'[а-яА-ЯёЁ]');
  return russianRegex.hasMatch(text);
}
```

Рисунок 27 – Метод searchCity

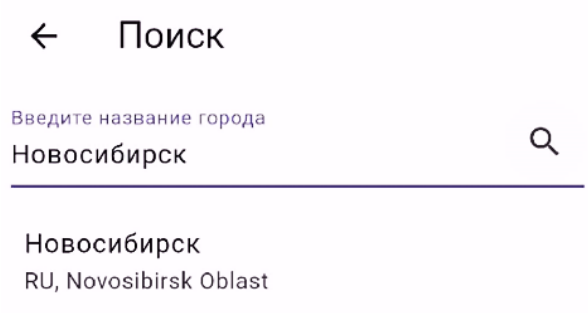


Рисунок 28 – Экран «Поиск города»

Экран загрузки обеспечивает плавный старт приложения, используя сервис местоположения, сервис инициализации и библиотеку geolocator.

Экран представлен на рисунке 29. В коде реализации данного экрана содержатся следующие ключевые этапы и компоненты:

– проверка наличия подключения к интернету с помощью библиотеки dart:io и метода checkInternetConnection;

– проверка разрешения на использование геолокации с помощью метода checkLocationPermission;

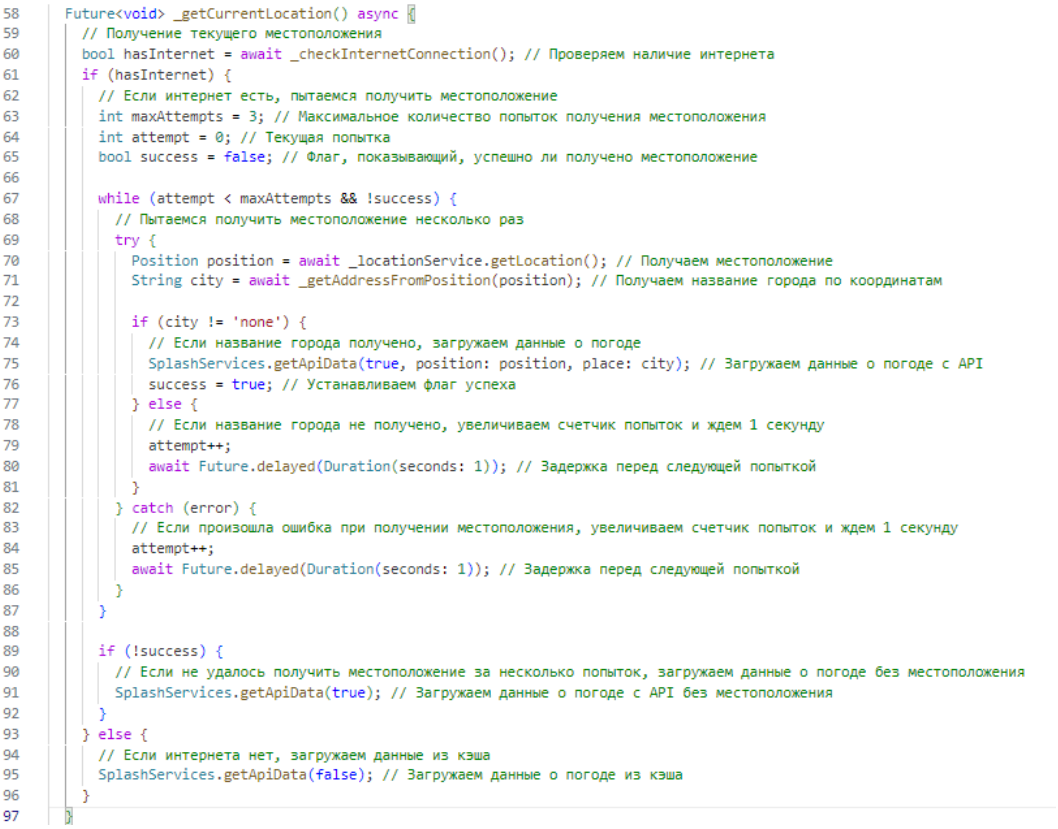
– запрос разрешения на доступ к местоположению в случае его отсутствия с помощью метода requestLocationPermission;

– метод getCurrenLocation позволяет получить текущее местоположение пользователя, он продемонстрирован на рисунке 30;

– метод getAdressFromPosition осуществляет обратное геокодирование для определения названия города на основе полученных координат;

– загрузка данных о погоде с использованием сервиса инициализации из кэша.

Рисунок 29 – Запрос разрешения на доступ к данным о местоположении устройства экрана загрузки



```dart
Future<void> _getCurrentLocation() async {
  // Получение текущего местоположения
  bool hasInternet = await _checkInternetConnection(); // Проверяем наличие интернета
  if (hasInternet) {
    // Если интернет есть, пытаемся получить местоположение
    int maxAttempts = 3; // Максимальное количество попыток получения местоположения
    int attempt = 0; // Текущая попытка
    bool success = false; // Флаг, показывающий, успешно ли получено местоположение

    while (attempt < maxAttempts && !success) {
      // Пытаемся получить местоположение несколько раз
      try {
        Position position = await _locationService.getLocation(); // Получаем местоположение
        String city = await _getAddressFromPosition(position); // Получаем название города по координатам

        if (city != 'none') {
          // Если название города получено, загружаем данные о погоде
          SplashServices.getApiData(true, position: position, place: city); // Загружаем данные о погоде с API
          success = true; // Устанавливаем флаг успеха
        } else {
          // Если название города не получено, увеличиваем счетчик попыток и ждем 1 секунду
          attempt++;
          await Future.delayed(Duration(seconds: 1)); // Задержка перед следующей попыткой
        }
      } catch (error) {
        // Если произошла ошибка при получении местоположения, увеличиваем счетчик попыток и ждем 1 секунду
        attempt++;
        await Future.delayed(Duration(seconds: 1)); // Задержка перед следующей попыткой
      }
    }

    if (!success) {
      // Если не удалось получить местоположение за несколько попыток, загружаем данные о погоде без местоположения
      SplashServices.getApiData(true); // Загружаем данные о погоде с API без местоположения
    }
  } else {
    // Если интернета нет, загружаем данные из кэша
    SplashServices.getApiData(false); // Загружаем данные о погоде из кэша
  }
}
```

Рисунок 30 – Метод получения текущего местоположение пользователя

3.4 Тестирование мобильного приложения

Функциональное тестирование является критически важным этапом разработки мобильного приложения прогноза погоды, обеспечивая корректную работу всех функциональных компонентов и высокое качество конечного продукта [19].

Тест-кейсы представляют собой набор конкретных шагов и условий, используемых для проверки каждого функционального требования. Они включают в себя предусловия, действия и ожидаемые результаты, что позволяет систематически и последовательно проверять все аспекты работы приложения [20].

В таблице 4 представлены тест-кейсы для тестирования основных функций мобильного приложения.

Таблица 4 – Тестирование основных функций приложения

№	Название тест-кейса	Предусловия	Шаги теста	Ожидаемый результат
1	Отображение текущей погоды	Приложение запущено	1. Выбрать город. 2. Открыть главный экран.	Отображается текущая температура, показаны параметры, присутствует иконка текущего состояния погоды.
2	Отображение прогноза на 14 дней	Приложение запущено	1. Выбрать город. 2. Открыть главный экран. 3. Перейти на вкладку «2 недели».	Отображается прогноз на 14 дней вперед, отображены иконки погодных условий.
3	Почасовой прогноз	Приложение запущено	1. Выбрать город. 2. Открыть главный экран.	Отображается температура по часам на текущие сутки, прокрутка работает.
4	Поиск по названию города	Приложение запущено, есть доступ к интернету	1. Нажать на название города. 2. Ввести название города. 3. Нажать кнопку поиска. 4. Выбрать город из списка.	В списке появился введенный город. Отображается погода для выбранного города.
5	Автоматическое определение местоположения	Приложение запущено	1. Разрешить доступ на использование геолокации. 2. Открыть главный экран.	Отображается погода для текущего местоположения.
6	Настройка уведомлений	Приложение запущено, разрешены push-уведомления	1. Нажать на иконку настроек. 2. Активировать пункт «Напомнить, если температура станет выше 25». 3. Сохранить настройки.	При достижении температуры выше 25 градусов приходит push-уведомление. Настройки сохраняются после перезапуска приложения.
7	Работа в офлайн-режиме	Приложение запущено, отсутствует интернет-соединение	1. Выбрать город. 2. Открыть главный экран.	Отображаются кэшированные данные о погоде, статус соединения и время последнего обновления.

Все тест-кейсы были успешно пройдены, что подтверждает соответствие приложения заявленным функциональным требованиям.

Интеграционное тестирование представляет собой процесс проверки корректности взаимодействия компонентов системы [24]. В рамках разработанного теста было проверено взаимодействие контроллеров, сервисов и пользовательского интерфейса. А именно полный сценарий поиска и смены города, включая работу геолокационных сервисов, обновление состояния и навигацию между экранами. Тест был реализован с использованием пакетов flutter test [21] и flutter integration test [23], которые позволяют имитировать реальные действие пользователя и проверяют поведение системы.

Описание работы интеграционного теста:

– инициализация окружения: устанавливается тестовый режим и сбрасываются зависимости;
– регистрация зависимостей необходимых сервисов и контроллеров;
– запуск приложения с начальным городом Москва;
– автоматический переход с экрана загрузки на главный экран;
– проверка корректности отображения начального города;
– переход на экран поиска городов;
– ввод нового города Самара в поле поиска;
– выбор города из списка результатов;
– возврат на главный экран с проверкой обновления города в контроллере.

Чтобы обеспечить надежность интеграционного тестирования, были реализованы специальные механизмы ожидания. Например, метод pumpUntilSplashCompletes, представленный на рисунке 31, гарантирует завершение анимации экрана загрузки. Расширение PumpUntil с таймаутом и логированием обеспечивает стабильность тестов при асинхронных операциях. Дополнительно была реализована настройка индивидуальных таймаутов на разных этапах теста и явное указание продолжительности анимаций.

```dart
// Хелпер для ожидания завершения SplashScreen
Future<void> _pumpUntilSplashCompletes(WidgetTester tester) async {
  final endTime = DateTime.now().add(const Duration(seconds: 30));

  // Ожидаем пока SplashScreen не выполнит свою работу
  while (find.byType(SplashScreen).evaluate().isNotEmpty) {
    if (DateTime.now().isAfter(endTime)) {
      throw Exception('SplashScreen не завершил работу за 30 секунд');
    }
    await tester.pump(const Duration(milliseconds: 100));
  }
}
```

Рисунок 31 – Метод pumpUntilSplashCompletes

Результаты тестирования, представленные на рисунке 32, подтвердили хорошую работоспособность на всех уровнях приложения.

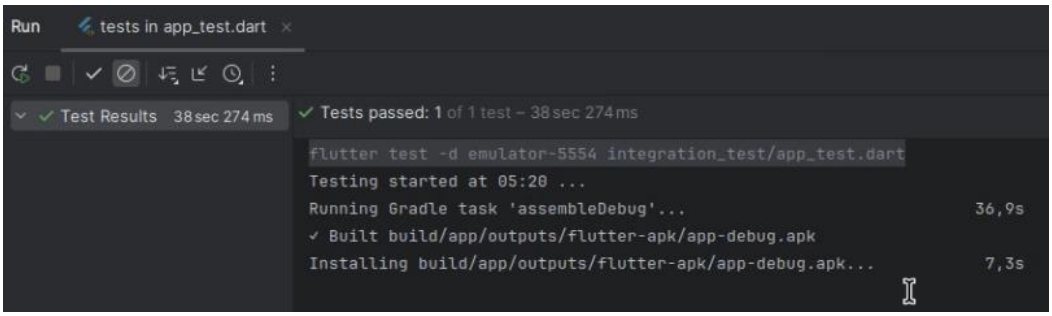


Рисунок 32 – Результаты интеграционного теста

Вывод по разделу 3

В третьем разделе подробно рассмотрена реализация мобильного приложения. Проведено функциональное и интеграционное тестирование, результаты которого подтвердили успешную работоспособность приложения.

Заключение

В рамках выпускной квалификационной работы было разработано мобильное приложение прогноза погоды.

В ходе работы были рассмотрены функциональные и архитектурные особенности, сформированы требования к мобильному приложению по методике FURPS+, проведен анализ существующих аналогов. По результатам анализа сделан вывод о целесообразности собственной разработки.

Особое внимание было уделено выбору инструментов разработки. Были выбраны следующие инструменты: фреймворк Flutter, язык программирования Dart, среда разработки Android Studio и API OpenWeatherMap. Была создана диаграмма вариантов использования, разработан прототип и создана диаграмма компонентов архитектуры приложения с использованием шаблона MVVM.

Разработано мобильное приложение для прогноза погоды, предназначенное для операционной системы Android.

В ходе функционального и интеграционного тестирования было установлено, что приложение работает стабильно и полностью соответствует заявленным требованиям.

Практическая значимость данной работы заключается в разработке мобильного приложения прогноза погоды, которое может отображать данные без подключения к интернету и отправлять условные уведомления.

Разработанное программное решение обладает потенциалом для практического применения. Оно может быть использовано как самостоятельный коммерческий продукт или стать основой для дальнейшего развития и расширения функциональных возможностей.

Таким образом, можно сделать вывод, что поставленные задачи и цель были достигнуты. Выполненная работа показала, как современные технологии могут быть использованы для создания полезных и востребованных приложений.

Список используемой литературы и используемых источников

1. Официальный сайт Gismeteo: [Электронный ресурс]. URL: https://www.gismeteo.ru/info/ (дата обращения: 15.11.2024)
2. Yahoo Weather: Official Documentation [Электронный ресурс]. URL: https://developer.yahoo.com/weather/ (дата обращения: 15.11.2024)
3. IBM The Weather Channel: Data Sources [Электронный ресурс]. URL: https://weather.com/science/weather-explainers (дата обращения: 15.11.2024)
4. OpenWeatherMap API Documentation [Электронный ресурс]. URL: https://openweathermap.org/api (дата обращения: 13.01.2025)
5. AccuWeather API documentation [Электронный ресурс]. – URL: https://developer.accuweather.com/ (дата обращения: 13.01.2025)
6. Weatherbit Enterprise Solutions [Электронный ресурс]. URL: https://www.weatherbit.io/api (дата обращения: 13.01.2025)
7. Соммервилл И., Инженерия программного обеспечения. 9-е изд. — М.: Вильямс, 2016. — С. 112.
8. React Native Documentation [Электронный ресурс]. URL: https://reactnative.dev (дата обращения: 25.11.2024)
9. React Native [Электронный ресурс] // Блог SkillFactory. — URL: https://blog.skillfactory.ru/glossary/react-native/ (дата обращения: 25.11.2024)
10. Сравнительный анализ React Native, Xamarin и Flutter для мобильной разработки [Электронный ресурс] // Хабр. — 2017. — URL: https://habr.com/ru/articles/343098/ (дата обращения: 26.11.2024)
11. Flutter Documentation [Электронный ресурс]. URL: https://flutter.dev (дата обращения: 27.11.2024)
12. Microsoft MVVM Documentation [Электронный ресурс]. URL: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm (дата обращения: 20.11.2024)

13. Shared Preferences Package [Электронный ресурс]. URL: https://pub.dev/packages/shared_preferences (дата обращения: 02.12.2024)

14. Основные принципы создания пользовательских интерфейсов (ПИ) [Электронный ресурс]. URL: https://ncrdo.ru/center/blog/osnovnye-printsipy-sozdaniya-polzovatelskikh-interfeysov-pi/ (дата обращения: 25.11.2024)

15. Dart convert library [Электронный ресурс]. URL: https://api.flutter.dev/flutter/dart-convert/ (дата обращения: 13.01.2025)

16. Dart async library [Электронный ресурс] // URL: https://dart-dev.web.app/libraries/dart-async (дата обращения: 13.01.2025)

17. Dart I/O library: dart:io [Электронный ресурс]. URL: https://dart.dev/libraries/dart-io (дата обращения: 15.11.2024)

18. Fetch data from the internet [Электронный ресурс]. URL: https://dart.dev/tutorials/server/fetch-data (дата обращения: 15.11.2024)

19. Functional testing // Wikipedia. URL: https://en.wikipedia.org/wiki/Functional_testing (дата обращения: 16.04.2025).

20. Тест-кейс в тестировании программного обеспечения [Электронный ресурс] // URL: https://www.sravni.ru/kursy/info/test-kejs-v-testirovanii/ (дата обращения: 16.04.2025)

21. Flutter Test package documentation [Электронный ресурс] // Flutter Official Documentation. – URL: https://api.flutter-io.cn/flutter/flutter_test (дата обращения: 19.04.2025)

22. Характеристики качества FURPS+ [Электронный ресурс] // LiveJournal. – 2023. – URL: https://m-i-kuznetsov.livejournal.com/220415.html (дата обращения: 12.10.2024)

23. Integration testing [Электронный ресурс] // Flutter documentation. – URL: https://flutter-website-filiph-staging.web.app/testing/integration-tests (дата обращения: 20.04.2025)

24. Unit-тесты во Flutter [Электронный ресурс] // Habr. – 2023. – URL: https://habr.com/ru/companies/friflex/articles/666578/ (дата обращения: 15.04.2025)

25. Material library - Flutter API docs [Электронный ресурс] : официальная документация / URL: https://api.flutter.dev/flutter/material/ (дата обращения: 12.02.2025)