

Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина «Объектно-ориентированное программирование»

«К защите допустить»
Руководитель курсового проекта
ассистент кафедры
_____ А.Г. Бокун
____.____.2024

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему:
«Приложение для обмена изображениями»

БГУИР КП 6-05-0612-02 07 ПЗ

Выполнила студент группы 353502
ЗГИРСКАЯ Дарья Денисовна

(подпись студента)

Курсовой проект представлен на
проверку _____.____.2025

(подпись студента)

Минск 2025

СОДЕРЖАНИЕ

| | |
|--|----|
| Введение..... | 4 |
| 1 Обзор существующих аналогов..... | 5 |
| 1.1 Обзор рынка социальных сетей | 5 |
| 1.2 Анализ аналогов и конкурентов..... | 6 |
| 2.1 Функциональные требования..... | 12 |
| 2.2 Нефункциональные требования..... | 13 |
| 2.3 Требования к пользовательскому опыту..... | 15 |
| 3 Теоретические основы объектно-ориентированного проектирования | 17 |
| 3.1 Объектно-ориентированное программирование..... | 17 |
| 3.2 MVVM архитектура | 20 |
| 4 Разработка, тестирование и анализ результатов работы приложения: применение принципов ооп на практике | 22 |
| 4.1 Используемые технологии и инструменты разработки | 22 |
| 4.2 Архитектура приложения | 24 |
| 4.3 Реализация логики работы приложения | 27 |
| 4.5 Тестирование логики приложения..... | 32 |
| Заключение | 37 |

ВВЕДЕНИЕ

В современном мире социальные сети играют большую роль в нашей повседневной жизни. Среди них особой популярностью пользуются такие приложения, как Instagram, TikTok, Pinterest и другие, позволяющие обмениваться изображениями и видео о своей жизни и увлечениях, в развлекательных и образовательных целях, а также для обмена идеями и вдохновения.

Разрабатываемое приложение представляет собой подобие Instagram: пользователь может подписываться на других пользователей, чтобы следить за их обновлениями; также пользователь может лайкать пост, если он ему понравился; и загружать собственные изображения и истории.

Использование MVVM архитектуры (Model-View-ViewModel) при разработке проекта обеспечивает гибкость, масштабируемость и удобство поддержки кода, что особенно важно для такого рода приложений, где возможны частые обновления и добавление нового контента.

Целью данного проекта является создание мобильного приложения под Android на Kotlin с использованием Jetpack Compose, сочетающей механики подписки, лайков и хранения и обмена изображениями.

Для достижения этой цели были поставлены следующие задачи:

- 1 Разработать ядро приложения на основе MVVM архитектуры, обеспечив разделение логики, интерфейса и данных.
- 2 Реализовать базовый функционал: загрузку изображений, авторизацию и регистрацию пользователя, отображение профиля.
- 3 Добавить функционал подписок и возможность их просматривать, а также отображать список подписчиков.
- 4 Добавить «лайки» на посты.
- 5 Реализовать интуитивно понятный интерфейс с использованием Jetpack Compose.

Проект разрабатывается на языке Kotlin с применением следующих технологии Jetpack Compose, позволяющей создавать интерфейс на основе специальных Composable функций.

3 MVVM архитектура – это архитектурный паттерн, который разделяет приложение на три основных компонента для удобства разработки и поддержки (Model, View, ViewModel).

1 ОБЗОР СУЩЕСТВУЮЩИХ АНАЛОГОВ

1.1 Обзор рынка социальных сетей

Современный рынок социальных сетей представляет собой динамично развивающуюся отрасль, которая оказывает значительное влияние на коммуникацию, развлечения, бизнес и маркетинг. По исследованию [1], в 2023 году число пользователей социальных сетей в мире превысило 4.9 млрд человек, что составляет около 60% населения Земли.

Рассмотрим ключевых игроков рынка:

1 Instagram (Meta) – одна из самых популярных платформ для обмена визуальным контентом (фото, видео, Stories). Активно используется блогерами и бизнесом.

2 TikTok (ByteDance) – лидер среди short-video платформ с алгоритмами персонализации контента.

3 Pinterest – социальная сеть для поиска идей (дизайн, мода, кулинария) с акцентом на визуальный контент.

4 Facebook (Meta) – крупнейшая соцсеть с широким функционалом (лента, группы, маркетплейс).

5 Twitter (X) – платформа для микроблогинга и обсуждения актуальных тем.

В период 2023-2024 годов в социальных сетях стали просматриваться следующие тренды:

- видеоконтент (Reels, Shorts) – остается главным драйвером роста;
- монетизация – расширение возможностей для создателей контента (донаты, подписки, партнерские программы);
- AR-фильтры и VR – интеграция технологий дополненной реальности (например, маски в Instagram);
- алгоритмические ленты – персонализация контента на основе ИИ;
- эфемерный контент (Stories, disappearing messages) – рост популярности временного контента;

На сегодняшний день рынок социальных сетей продолжает расти за счет расширения аудитории в развивающихся странах, интеграции e-commerce (покупки через соцсети), развития метавселенных и Web3-технологий (децентрализованные соцсети).

Данный обзор подтверждает актуальность разработки социальных приложений, ориентированных на визуальный контент и взаимодействие пользователей.

1.2 Анализ аналогов и конкурентов

Современный рынок социальных сетей [2] демонстрирует четкую специализацию платформ по типам контента и целевой аудитории. Наиболее близкими аналогами разрабатываемого приложения являются Instagram [3], TikTok и Pinterest, в то время как Facebook, VK и Twitter занимают смежные ниши, предлагая иные модели взаимодействия [4].

Instagram (рисунок 1.2.1), принадлежащий Meta, остается безусловным лидером среди платформ для обмена визуальным контентом. Его ключевое преимущество - идеально сбалансированный интерфейс для публикации фотографий и видео, дополненный функциями Stories и Reels. Однако в последние годы платформа столкнулась с проблемой перенасыщения рекламой и снижением органического охвата, что вызывает недовольство части пользовательской базы, особенно среди творческой аудитории и микроблогеров.

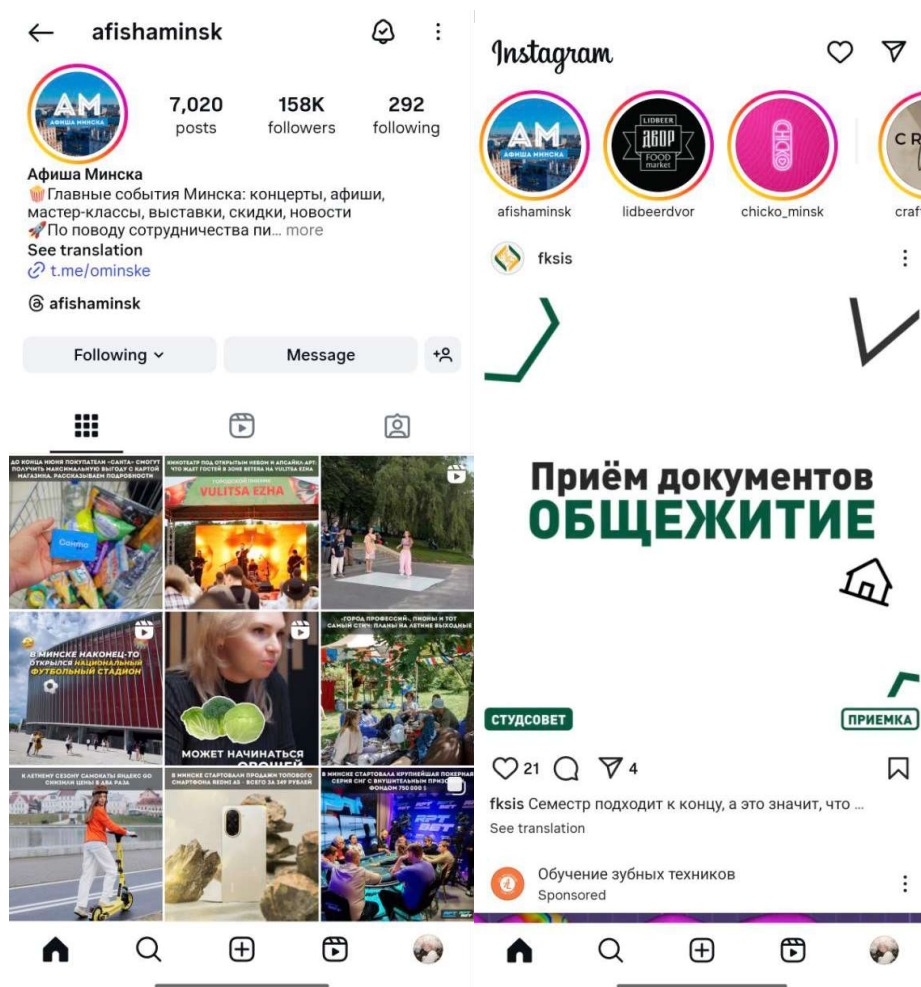


Рисунок 1.2.1 – Instagram

TikTok (рисунок 1.2.2), революционизировавший рынок коротких видео, создал абсолютно новую парадигму потребления контента через вертикальный скроллинг и сверхточные алгоритмы рекомендаций.

Хотя платформа демонстрирует феноменальные показатели вовлеченности, ее формат существенно отличается от классических фото-ориентированных соцсетей, оставляя пространство для более специализированных решений.

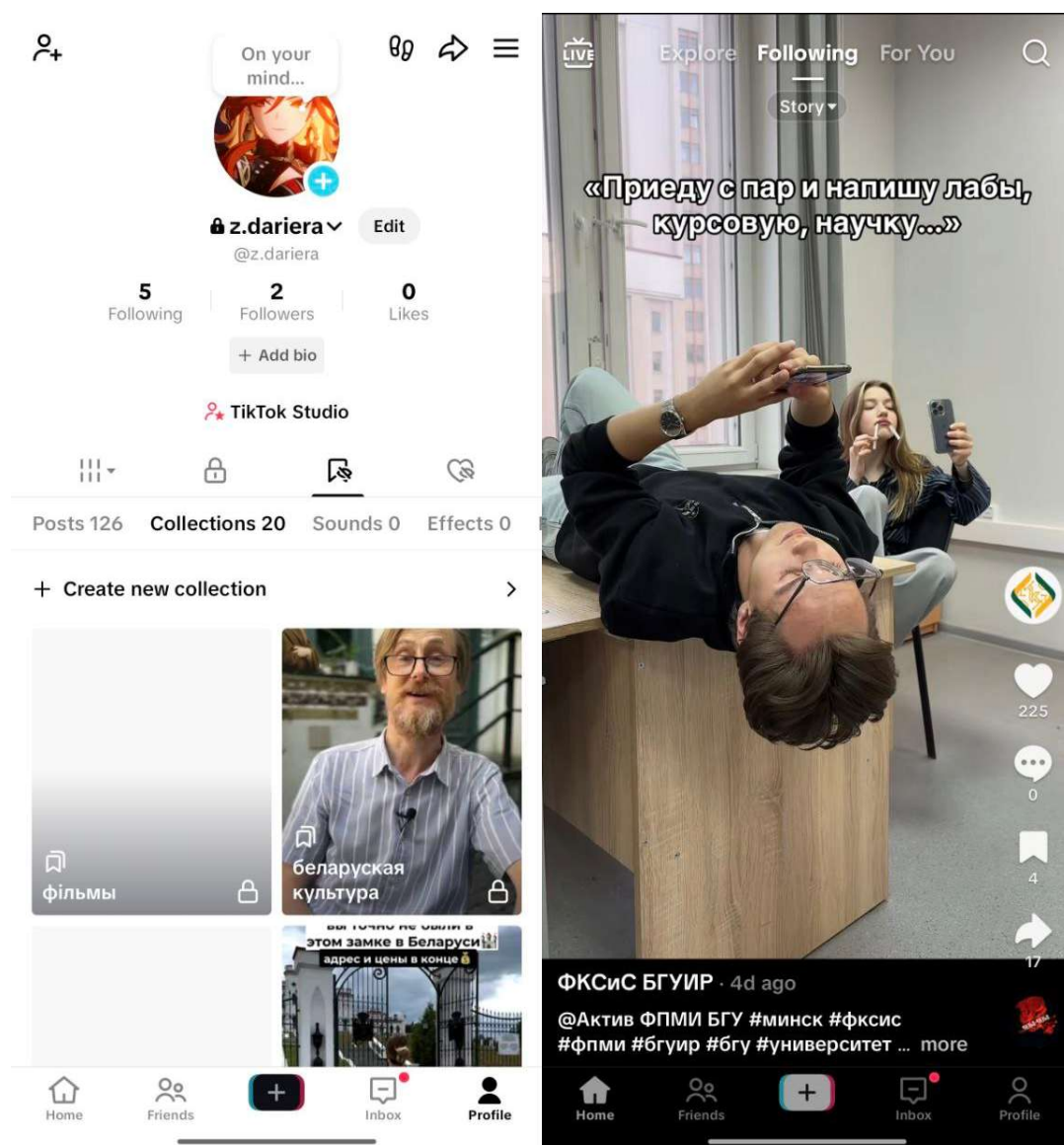


Рисунок 1.2.2 – TikTok

Pinterest (рисунок 1.2.3) занимает уникальную нишу "визуального поисковика", сочетая элементы социальной сети с инструментами для сохранения и систематизации контента. Его аудитория преимущественно женская, ориентированная на поиск вдохновения в дизайне, моде и кулинарии.

Однако слабая социальная составляющая (ограниченные возможности для комментирования и обсуждений) делает Pinterest скорее дополнением, чем конкурентом классическим соцсетям.

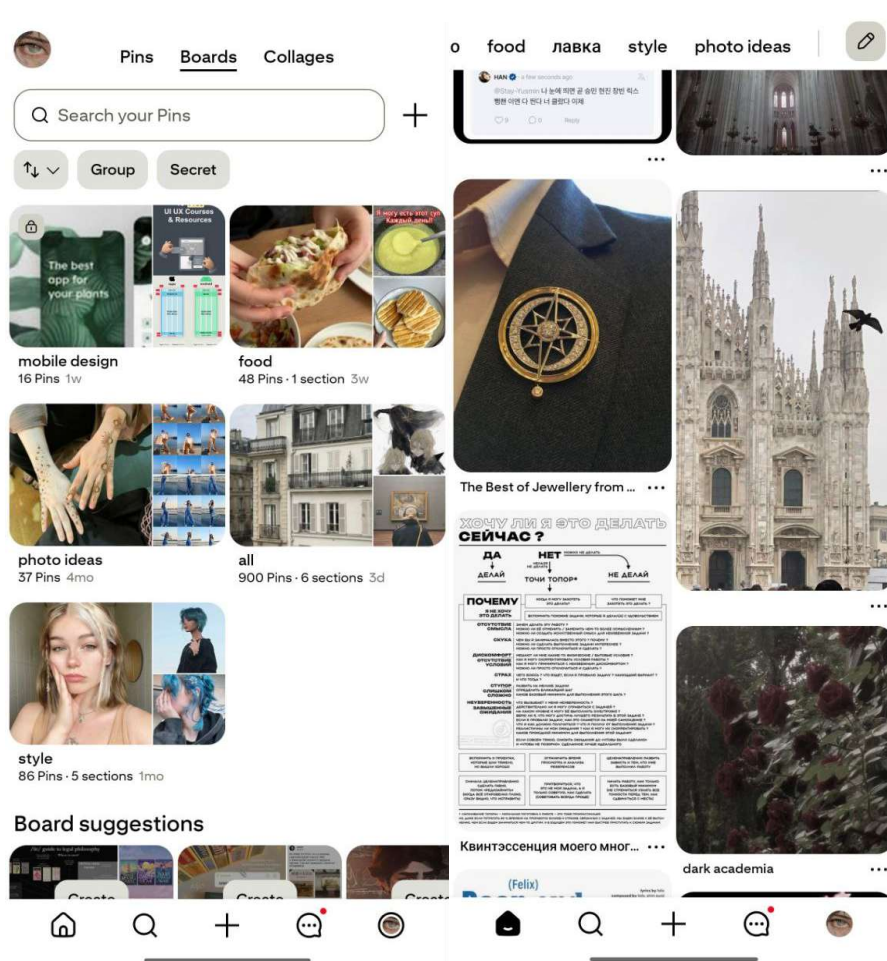


Рисунок 1.2.3 – Pinterest

Facebook и VK (рисунок 1.2.4) представляют собой наиболее яркие примеры универсальных социальных экосистем, которые за годы развития превратились из простых сетей для общения в сложные многофункциональные платформы. Эта трансформация привела к созданию уникальных цифровых пространств, сочетающих в себе элементы социального взаимодействия, медиапотребления и бизнес-инструментария.

Однако их главная слабость – попытки быть "все для всех", что приводит к перегруженности интерфейса и размыванию целевой аудитории. Попытка совместить в одной платформе столь разнородные функции неизбежно приводит к компромиссам в пользовательском опыте. В отличие от более специализированных сетей (например, LinkedIn для профессионалов

или TikTok для развлечений), Facebook и VK вынуждены балансировать между потребностями совершенно разных групп пользователей

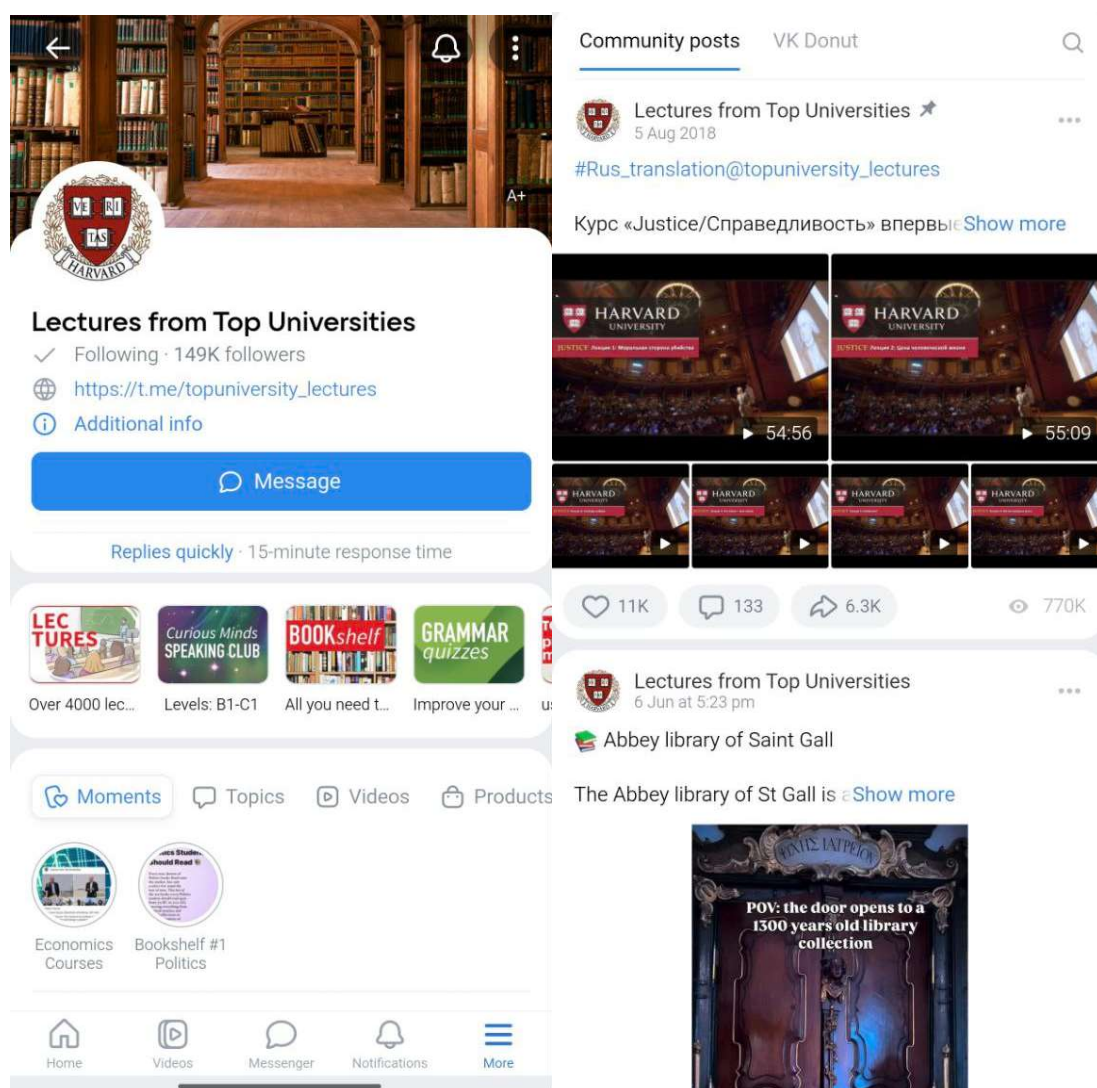


Рисунок 1.2.4 – VK

Twitter (X), несмотря на недавние изменения, остается в первую очередь тексто-ориентированной платформой для оперативного обмена информацией.

Проведенный анализ выявляет несколько перспективных направлений для разрабатываемого приложения:

- создание более сфокусированного и менее коммерциализированного пространства для обмена визуальным контентом по сравнению с Instagram;
- развитие социальных функций, которых не хватает Pinterest;
- предложение упрощенного и интуитивного интерфейса в противовес перегруженному функционалом Facebook и VK.

Данные аналоги демонстрируют устойчивый интерес аудитории к социальным сетям с медиаконтентом и персонализацией пространства.

Особую ценность может представлять сочетание эстетики раннего Instagram с современными форматами типа Stories, но без избыточной автоматизации контента, характерной для TikTok. Это позволит занять нишу среди пользователей, ценящих визуальное самовыражение, но уставших от агрессивных алгоритмов и коммерциализации существующих платформ.

1.2 Отличительные особенности проекта

Разрабатываемая социальная сеть представляет собой современное мобильное приложение, ориентированное на удобный обмен визуальным контентом в формате фотографий и коротких историй. В отличие от перегруженных функционалом универсальных платформ, проект делает ставку на простоту, интуитивность и комфортное взаимодействие пользователей с контентом.

Ключевые особенности интерфейса и функционала:

1 Основной приложением становится персональный профиль пользователя, включающий аватар, публикации, список подписчиков и подписок. Такой подход обеспечивает прозрачность социальных связей и создает условия для формирования сообщества вокруг контента. В отличие от многих современных аналогов, где профиль перегружен второстепенными функциями, здесь акцент делается именно на визуальной составляющей и социальном взаимодействии.

2 Система публикаций реализована через два основных формата: классические посты и временные истории. Пользователи могут делиться фотографиями в основной ленте, где другие участники смогут оценивать контент через механизм лайков. В отличие от алгоритмических лент крупных платформ, здесь предполагается хронологическая подача контента от подписанных аккаунтов, что делает взаимодействие более предсказуемым и осмысленным.

3 Функция историй, оформленная в виде кружков в верхней части ленты, позволяет делиться моментами в формате 24 часа. Этот популярный формат, впервые внедренный Snapchat и популяризированный Instagram, в данном проекте получает минималистичную реализацию без избыточных фильтров и эффектов, что соответствует общей концепции простоты и удобства.

5 Механизм поиска реализован через отдельный раздел, позволяющий находить других пользователей по ключевым запросам. В отличие от сложных поисковых систем крупных платформ, здесь используется упрощенный

алгоритм, делающий процесс обнаружения контента и людей более прямым и прозрачным.

6 Система авторизации через электронную почту обеспечивает простой старт для новых пользователей, избегая сложных процедур верификации, характерных для некоторых современных платформ. Такой подход снижает барьер входа, сохраняя при этом достаточный уровень безопасности и идентификации.

Отличия от существующих аналогов:

- минималистичный дизайн – в отличие от перегруженных интерфейсов Facebook или VK, здесь используется чистый, воздушный дизайн с акцентом на контент;

- хронологическая лента – в противовес алгоритмическим лентам Instagram и TikTok, где порядок постов определяется сложными алгоритмами;

- сфокусированный функционал – отсутствие второстепенных функций (маркетплейсов, игр, сервисов знакомств), которые отвлекают от основного назначения;

- упрощенная система взаимодействий – базовые действия (лайки, подписки) без избыточных вариантов реакций;

- прямая навигация – четкое разделение на основные разделы (лента, поиск, создание, профиль) без скрытых меню и сложных переходов;

2 ТРЕБОВАНИЯ К ПРОЕКТУ

2.1 Функциональные требования

Функциональные требования описывают конкретные задачи, которые должна выполнять реализовываемое приложение. Разрабатываемая социальная платформа предоставляет пользователям комплексный, но при этом интуитивно понятный набор возможностей для взаимодействия с контентом и другими участниками сети. Ядро функционала строится вокруг трех ключевых аспектов: создания и публикации контента, социального взаимодействия и персонализации профиля.

Для начала рассмотрим систему публикаций и управления контентом. Благодаря ей пользователи получают возможность создавать два основных типа контента. Для долговременных публикаций предусмотрен механизм постов – пользователи могут загружать фотографии, дополняя их при необходимости текстовыми описаниями. Временный контент размещается через систему историй, которые автоматически удаляются через 24 часа после публикации. Особое внимание уделено процессу загрузки медиа – интерфейс предусматривает предпросмотр, базовое кадрирование и возможность повторной загрузки при неудачных попытках.

Для социального взаимодействия платформа реализует полный цикл социальных связей. Пользователи могут подписываться на интересные аккаунты, формируя персональную ленту контента. Механизм подписок двусторонний – каждый участник видит как список своих подписчиков, так и перечень аккаунтов, на которые подписан сам. Система лайков позволяет мгновенно реагировать на понравившийся контент. В отличие от многих современных аналогов, лента формируется строго в хронологическом порядке публикаций от подписанных аккаунтов, без алгоритмического вмешательства.

Далее необходимо реализовать персонализацию и управление профилем. Это включает в себя то, что каждый пользователь получает полноценный профиль с аватаром, который служит его визуальным идентификатором в системе. Профиль отображает статистику активности (количество публикаций, подписчиков и подписок), а также сетку опубликованных постов. Реализована возможность редактирования базовой информации о себе. Система предусматривает простой механизм переключения между режимом просмотра собственного профиля и его отображением для других пользователей.

Также архитектура приложения построена вокруг четырех основных разделов, доступных через нижнюю панель навигации: лента подписок, раздел

поиска, экран создания контента и личный профиль. Поисковая система позволяет находить других пользователей по ключевым словам, с отображением результатов в режиме реального времени. В верхней части ленты реализован горизонтальный скролл с кружками историй от подписанных аккаунтов.

Процесс регистрации и входа в систему реализован через электронную почту, что обеспечивает баланс между удобством и безопасностью. Новые пользователи проходят стандартную процедуру верификации через email, после чего получают возможность настроить свой профиль. Система предусматривает восстановление доступа через привязанный email-адрес.

Технические аспекты реализации заключаются в том, что все функциональные требования реализуются с соблюдением принципов MVVM-архитектуры, что обеспечивает четкое разделение между:

- бизнес-логикой (обработка данных, работа с API);
- presentation-слоем (отображение интерфейса);
- domain-логикой (правила взаимодействия).

Помимо этого, особое внимание уделено обработке пользовательских действий – все интерфейсные элементы обеспечивают мгновенную визуальную обратную связь при взаимодействии (анимация лайков, индикаторы загрузки, плавные переходы между экранами).

2.2 Нефункциональные требования

Нефункциональные требования определяют ключевые характеристики качества, надежности и удобства использования разрабатываемой социальной платформы. Они охватывают аспекты производительности, безопасности, масштабируемости, пользовательского опыта и совместимости, формируя основу для стабильной и комфортной работы приложения.

Для начала проанализируем производительность и отзывчивость приложения.

Приложение должно обеспечивать плавную работу на большинстве современных Android-устройств, включая модели со средними техническими характеристиками. Время загрузки ленты новостей не должно превышать 1-2 секунд при стабильном интернет-соединении, а переход между экранами (профиль, лента, поиск) должен происходить мгновенно, без видимой задержки. Загрузка изображений и историй оптимизируется через кэширование – уже просмотренные медиафайлы сохраняются локально, чтобы минимизировать трафик и ускорить повторный доступ. Прокрутка ленты

должна быть плавной, без фризлов или подтормаживаний, даже при большом количестве контента.

Что касается надежности и отказоустойчивости, социальная сеть должна работать стабильно в стандартных условиях использования, без неожиданных завершений или зависаний. В случае потери соединения или серверных ошибок пользователь получает понятное уведомление с возможностью повторить действие. Все данные (посты, лайки, подписки) автоматически синхронизируются с сервером, а критическая информация (настройки профиля, список подписок) дополнительно сохраняется локально на устройстве. Это исключает потерю данных при переустановке приложения или временном отсутствии интернета.

Помимо этого, приложение должно гарантировать защиту пользовательских данных на всех этапах работы. Авторизация по электронной почте включает обязательную верификацию, а передача данных осуществляется только через зашифрованные соединения (HTTPS). Пароли и персональная информация хранятся в безопасном формате, с использованием хэширования и современных методов криптографии. Доступ к галерее устройства и камере запрашивается только при необходимости загрузки контента, с возможностью отзыва разрешений в настройках.

Для комфорта пользователя интерфейс должен строиться на принципах минимализма и интуитивности.

Все ключевые элементы (кнопка лайка, добавления поста, перехода в профиль) имеют четкие визуальные подсказки и мгновенную обратную связь при нажатии.

Анимации используются умеренно – для плавных переходов между экранами и визуального подтверждения действий (например, анимация сердца при лайке).

Шрифты, размеры кнопок и отступы адаптированы под разные диагонали экранов, включая компактные смартфоны.

Приложение поддерживает Android 8.1 и выше, охватывая более 93% активных устройств на рынке. Адаптивный интерфейс корректно отображается на экранах с разным соотношением сторон (18:9, 19.5:9, 20:9). Тестирование проводится на широком спектре устройств, включая модели с минимальными характеристиками (3 ГБ ОЗУ, 32 ГБ памяти).

Архитектура проекта позволяет легко добавлять новые функции без переработки основного кода. Например:

- ведение верификации профилей;
- добавление новых типов контента (видео, опросы);
- интеграция с мессенджером.

В разрабатываемой социальной сети серверная часть полностью построена на облачной платформе Firebase, что обеспечивает автоматическую масштабируемость и высокую доступность без необходимости управления собственной инфраструктурой. Такой подход позволяет сосредоточиться на разработке клиентской части, делегируя вопросы серверного масштабирования и производительности Google Cloud.

Firebase Authentication, Firestore и Storage автоматически масштабируются в зависимости от нагрузки, обеспечивая стабильную работу при росте пользовательской базы. Firestore использует распределенную архитектуру Google Cloud, что гарантирует:

- низкую задержку запросов (менее 200 мс для большинства операций);
- автоматическое шардирование данных при увеличении объема;
- поддержку до 1 миллиона одновременных подключений.

Помимо этого, приложение минимизирует нагрузку на батарею устройства. Фоновые процессы (синхронизация данных, уведомления) объединяются в пакеты и выполняются редко при неактивном использовании. Загрузка медиа (особенно историй) происходит только при подключении к Wi-Fi, если это указано в настройках.

Таким образом, эти требования формируют основу для создания социальной сети, которая сочетает высокую производительность с безопасностью и удобством. Акцент на стабильность работы, продуманный UX и адаптивность к разным устройствам позволит приложению конкурировать с существующими платформами, предлагая пользователям более предсказуемую и комфортную среду для общения.

2.3 Требования к пользовательскому опыту

Требования к пользовательскому опыту определяют, каким должно быть взаимодействие пользователя с социальной платформой на уровне ощущений, удобства, эмоциональной вовлечённости и общей удовлетворённости. Учитывая, что разрабатываемая платформа стремится предоставить интуитивно понятный и комфортный набор возможностей, основной акцент делается на простоту использования, доступность и удовольствие от процесса взаимодействия.

Одним из ключевых требований является интуитивно понятный и минималистичный интерфейс, который позволяет пользователю с первого взгляда понять логику работы приложения. Все ключевые элементы, такие как кнопки для лайков, добавления постов, перехода в профиль, должны быть чётко различимы и логично расположены. Пользователь всегда должен

понимать, что от него требуется на каждом этапе взаимодействия, будь то создание нового поста, просмотр ленты или поиск других пользователей. Важным элементом является мгновенная визуальная обратная связь: кнопки должны реагировать на нажатия, а выполненные действия должны сопровождаться анимациями, подтверждающими результат – например, анимация сердца при лайке или индикаторы загрузки при публикации контента.

Эстетическая составляющая также играет большую роль. Интерфейс должен быть визуально приятным и современным, с использованием сбалансированных цветовых схем и плавной типографики. Умеренное использование анимаций, таких как плавные переходы между экранами, служит не только для эстетики, но и для улучшения восприятия, делая взаимодействие более динамичным и менее резким.

Особое внимание уделяется бесперебойной и плавной работе приложения. Лента новостей должна загружаться быстро, а переходы между экранами (профиль, лента, поиск) должны происходить мгновенно, без видимой задержки. Прокрутка ленты должна быть абсолютно плавной, без каких-либо фризов или подтормаживаний, даже при интенсивном использовании и большом количестве контента. Это обеспечивает ощущение отзывчивости и надёжности, минимизируя фрустрацию пользователя.

Наконец, доступность и адаптивность обеспечивают комфортный опыт для максимально широкой аудитории. Интерфейс должен корректно отображаться на различных диагоналях экранов и разрешениях, адаптируясь под особенности устройств. Шрифты, размеры кнопок и отступы должны быть оптимальными для чтения и взаимодействия на разных устройствах, включая компактные смартфоны. Это гарантирует, что каждый пользователь, независимо от используемого устройства, получит единообразный и приятный опыт взаимодействия с платформой.

Таким образом, вышеописанные требования направлены на создание дружелюбной, предсказуемой и комфортной среды для общения и взаимодействия, обеспечивающей пользователя стабильным, интуитивно понятным и приятным опытом использования социальной платформы.

3 ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

3.1 Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) [5] – это не просто набор инструментов, а скорее философия создания программ. Представьте, что вы строите сложную модель железной дороги. ООП – это ваш подход к проектированию: вместо того чтобы лепить всё из одного куска, вы создаёте отдельные, взаимодействующие части: поезда, станции, стрелки, каждый из которых имеет свои уникальные особенности и действия. Этот способ мышления помогает стандартизировать написание кода, что приводит к меньшему количеству ошибок, ускорению разработки и делает код понятным и легко поддерживаемым для всей команды.

Основная идея ООП заключается в том, что все программы состоят из объектов. Каждый такой объект — это как самостоятельная "единица" внутри вашей программы, обладающая своими данными (характеристиками) и способностью выполнять определённые действия.

В мире ООП мы оперируем четырьмя ключевыми понятиями:

1 Объект – это конкретная, осязаемая сущность в вашей программе. Возьмём пример из приложения для онлайн-обучения: «Конкретный студент Иван Петров» - это объект. Или «Онлайн-курс по Kotlin для начинающих» - тоже объект. Каждый из них имеет свои уникальные данные (например, имя студента, его успеваемость, название курса, количество уроков) и может выполнять определённые действия.

2 Класс – это шаблон, по которому создаются объекты. Если объект – это уже построенный дом, то класс – это его архитектурный проект. Например, мы можем создать класс Студент, который описывает общую структуру любого студента: у него есть имя, фамилия, список пройденных курсов. На основе этого класса мы можем создавать конкретные объекты: студентИван, студентМария и так далее.

3 Классы могут наследовать свойства и поведение друг от друга. Например, у нас есть общий класс ПользовательПлатформы. От него могут наследоваться классы Студент и Преподаватель. Класс Преподаватель унаследует базовые свойства ПользователяПлатформы (например, логин, пароль), но добавит свои уникальные свойства (например, предметыПреподавания) и методы.

4 Метод – это функция или действие, которое может выполнять объект, или способ взаимодействия с ним. В примере с классом Студент, метод

записатьсяНаКурс(курс: Курс) позволит студенту зарегистрироваться на новый курс, а метод получитьОценку(урок: Урок) вернет его оценку за конкретный урок.

5 Атрибут (или Свойство) – это характеристика или данные, описывающие объект. Для объекта «Конкретный студент Иван Петров» атрибутами будут имя («Иван»), фамилия («Петров»), среднийБалл (например, 4.5). В классе Студент мы объявляем, что у каждого студента будут такие атрибуты, а при создании конкретного объекта-студента мы заполняем их соответствующими значениями.

Объектно-ориентированное программирование основывается на четырёх столпах, которые обеспечивают его эффективность и гибкость:

1 Инкапсуляция – это как «капсула», которая упаковывает данные и методы, работающие с этими данными, в единый блок (класс), а затем скрывает внутренние детали реализации от внешнего мира. Представьте себе современный автомобиль: вы взаимодействуете с ним через руль, педали, кнопки на панели. Вам не нужно знать, как именно работает двигатель или трансмиссия, чтобы управлять машиной. Интерфейс автомобиля (руль, педали) – это его публичное API, а все сложные механизмы внутри – инкапсулированы.

Правильная инкапсуляция имеет множество преимуществ:

- компоненты, чьи внутренние детали скрыты, могут быть легко использованы в разных частях программы или даже в других проектах, так как их внутренние изменения не влияют на внешнее взаимодействие;
- разработчики могут работать над разными модулями независимо, зная, что их изменения не «сломают» другие части системы, если они не меняют публичный интерфейс;
- когда что-то идёт не так, легче найти источник проблемы, потому что код разбит на независимые, хорошо изолированные части.

В Kotlin [6] инкапсуляция реализуется с помощью модификаторов доступа и свойств:

- private: доступен только внутри того же класса;
- public: доступен из любого места (по умолчанию);
- protected: доступен внутри класса и в его подклассах;
- internal: доступен внутри одного модуля (например, одного проекта Android-приложения).

Kotlin поощряет инкапсуляцию, делая свойства приватными по умолчанию или используя val/var для автоматического создания геттеров и сеттеров, при этом позволяя настроить их видимость. Например, вы можете

создать свойство, доступное только для чтения, не предоставляя публичный сеттер.

2 Наследование – это мощный принцип, который позволяет создавать иерархии классов, где дочерние классы (подклассы) перенимают характеристики и поведение родительских классов (базовых классов). Представьте себе классификацию животных: есть общий класс Животное, у которого есть базовые свойства (например, количествоЛап, типПитания) и действия (есть(), спать()). От него могут наследоваться Млекопитающее, Птица, Рыба. А от Млекопитающего, в свою очередь, может наследоваться Собака или Кошка, добавляя свои уникальные черты.

В Kotlin наследование обозначается с помощью ключевого слова `open`. Базовый класс должен быть помечен как `open`, чтобы от него можно было наследоваться.

Это позволяет повторно использовать уже написанный код и создавать специализированные версии, не дублируя общую логику. Дочерние классы могут не только использовать, но и переопределять поведение унаследованных методов (если они помечены как `open` в базовом классе и `override` в дочернем). Kotlin, в отличие от Java, по умолчанию делает классы `final`, что означает, что их нельзя наследовать, если явно не указано `open`. Это стимулирует более осознанное использование наследования. Классы в Kotlin могут наследовать только от одного другого класса, но при этом могут реализовывать множество интерфейсов, что даёт большую гибкость.

3 Полиморфизм буквально означает «много форм». Этот принцип тесно связан с наследованием и позволяет обрабатывать объекты разных классов как экземпляры их общего родительского класса, при этом каждый объект будет вести себя в соответствии со своим реальным типом.

Вернёмся к примеру с животными. У нас есть функция `попроситьИздатьЗвук(животное: Animal)`. Если мы передадим ей объект Собака, она издаст «Гав!». Если передадим Кошка, она скажет «Мяу!». Хотя функция работает с общим типом `Animal`, поведение будет зависеть от конкретного типа объекта.

Более строго, полиморфизм позволяет вызывать метод через переменную родительского класса и получить поведение, которое соответствует реальному классу-потомку, на который ссылается эта переменная. Это делает код более гибким и расширяемым, позволяя добавлять новые типы, не изменяя существующий код, который работает с базовым классом.

4 Абстракция – это процесс выделения только самых важных и существенных характеристик объекта, скрывая при этом ненужные детали

реализации. Это как карта города: она показывает улицы, здания и важные ориентиры, но не отображает каждую трещину в асфальте или цвет каждой двери. Она предоставляет вам достаточно информации, чтобы ориентироваться, но не перегружает ненужными подробностями.

В программировании абстракция помогает упростить сложные системы, представляя их в более управляемом виде. В Kotlin абстракция реализуется через абстрактные классы и интерфейсы:

1 Абстрактный класс (`abstract class`) служит шаблоном для других классов и не может быть создан напрямую (нельзя создать объект `abstract class`). Он может содержать как реализованные методы, так и абстрактные методы (помеченные `abstract`), которые не имеют реализации и обязаны быть переопределены в дочерних классах. Это позволяет задать общую структуру поведения, не вдаваясь в конкретные детали реализации.

2 Интерфейс (`interface`) полностью абстрактен. Он описывает только сигнатуры методов и свойств, но не содержит их реализации (хотя в Kotlin 1.2+ интерфейсы могут содержать реализации по умолчанию). Класс, реализующий интерфейс, должен самостоятельно описать всё поведение, определённое в интерфейсе. Интерфейсы используются для определения контрактов поведения.

Главная цель абстракции – упростить работу со сложными системами, разбив их на независимые, логические блоки с чётко определёнными обязанностями. Это снижает связанность кода (зависимость одних частей от других), делает его легче для изменения, расширения и тестирования.

3.2 MVVM архитектура

Архитектура MVVM [7] (Model-View-ViewModel) представляет собой мощный и широко используемый архитектурный паттерн, особенно актуальный для разработки приложений с графическим пользовательским интерфейсом. Он не является совершенно новым изобретением, но представляет собой усовершенствованную версию предыдущих паттернов, таких как MVC (Model-View-Controller) и MVP (Model-View-Presenter), стремясь решить их присущие сложности в области разделения ответственности и тестируемости.

Основная идея MVVM заключается в создании чёткого разделения между тремя ключевыми компонентами: пользовательским интерфейсом (View), который отвечает за отображение информации; бизнес-логикой приложения (Model), управляющей данными и правилами; и слоем, который связывает их (ViewModel), предоставляя данные для отображения и

обрабатывая пользовательские взаимодействия. ViewModel выступает в роли интеллектуального «посредника», который получает данные от Model, преобразует их в удобный для View формат и передаёт действия пользователя обратно в Model. Это тщательное разделение позволяет создавать приложения, которые являются не только гибкими и легко тестируемыми, но и значительно упрощают их сопровождение и дальнейшее развитие.

Взаимодействие между компонентами MVVM происходит в однонаправленном потоке. View наблюдает за изменениями данных, предоставляемых ViewModel. ViewModel, в свою очередь, запрашивает данные у Model. Когда Model предоставляет данные, ViewModel обрабатывает их и обновляет свои наблюдаемые поля, на которые подписана View, что приводит к автоматическому обновлению пользовательского интерфейса. При этом пользовательские действия, инициированные во View, передаются в ViewModel, которая затем обрабатывает их, возможно, инициируя операции в Model, и соответствующим образом обновляет данные для View.

Такой подход обеспечивает строгую направленность зависимостей: View зависит от ViewModel, а ViewModel зависит от Model. Сама Model остаётся полностью независимой, что гарантирует, что изменения в UI или логике представления не нарушат базовую бизнес-логику.

Преимущества архитектуры MVVM многочисленны. Она способствует чёткому разделению ответственности, что делает код более управляемым и понятным. Её высокая тестируемость позволяет изолированно проверять логику каждого компонента, особенно ViewModel, без необходимости запускать UI. Удобство сопровождения и расширения значительно возрастает благодаря модульной структуре. В Android, способность ViewModel сохранять состояние UI при изменениях конфигурации является неоценимым преимуществом. Наконец, MVVM прекрасно интегрируется с реактивными фреймворками, упрощая связывание данных и создавая динамичные и отзывчивые пользовательские интерфейсы. MVVM фактически стал стандартом в современной Android-разработке, предлагая надёжный и масштабируемый способ создания сложных приложений.

4 РАЗРАБОТКА, ТЕСТИРОВАНИЕ И АНАЛИЗ РЕЗУЛЬТАТОВ РАБОТЫ ПРИЛОЖЕНИЯ: ПРИМЕНЕНИЕ ПРИНЦИПОВ ООП НА ПРАКТИКЕ

4.1 Используемые технологии и инструменты разработки

Для реализации разработанного приложения были выбраны ведущие нативные технологии для Android-разработки: Android Studio в качестве основной интегрированной среды разработки (IDE) и Jetpack Compose для построения пользовательского интерфейса. Эти инструменты обеспечивают высокую эффективность, стабильность и глубокую интеграцию, что позволяет создавать современные, производительные и масштабируемые Android-приложения.

Jetpack Compose (рисунок 4.1.1) — это современный декларативный UI-фреймворк от Google, предназначенный для создания нативных пользовательских интерфейсов на платформе Android [8]. В отличие от традиционного императивного подхода с XML-разметкой, Compose позволяет описывать UI с помощью кода на Kotlin [9]. Это значительно упрощает процесс разработки, делая его более интуитивным и эффективным.

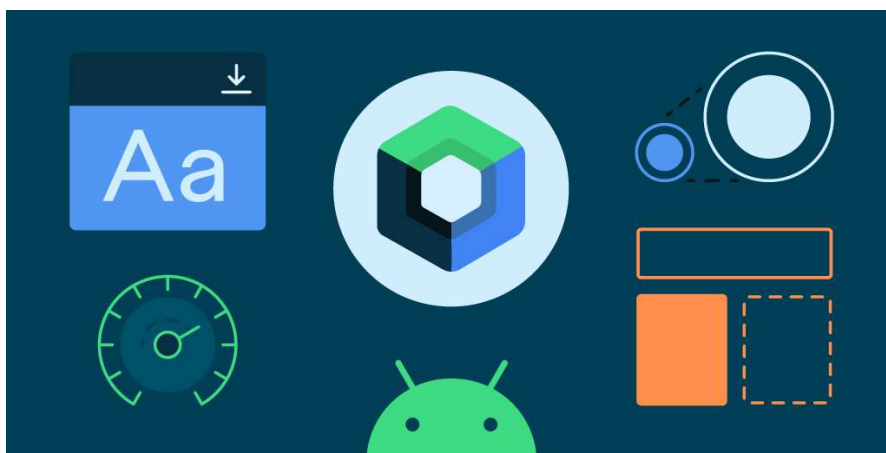


Рисунок 4.1.1 – Jetpack Compose

Одним из ключевых преимуществ Compose является его декларативный подход, который означает, что разработчик описывает, как UI должен выглядеть в данный момент, а фреймворк сам обновляет его при изменении состояния. Это значительно сокращает количество шаблонного кода, устраняет распространённые ошибки, связанные с управлением состоянием UI, и упрощает создание динамичных и отзывчивых интерфейсов. Compose также предлагает мощную систему композиции, позволяющую создавать

сложные UI из простых, многократно используемых компонентов, известных как «компоаблы» (composables) [10]. Он предоставляет обширную библиотеку готовых UI-элементов, а также гибкие возможности для создания кастомных компонентов. Jetpack Compose тесно интегрирован с другими библиотеками Android Jetpack, такими как ViewModel и LiveData/StateFlow, что облегчает реализацию архитектурных паттернов, в частности MVVM. Это обеспечивает доступ к нативным возможностям устройства, таким как камера, сенсоры, файловая система и другие платформенные API, что позволяет создавать полноценные и функциональные приложения.

Android Studio (рисунок 4.1.2) — это официальная интегрированная среда разработки (IDE) для платформы Android, разработанная Google на базе IntelliJ IDEA. Она является комплексным решением для всего цикла разработки мобильных приложений, начиная от написания кода и отладки, и заканчивая тестированием, профилированием и развертыванием.

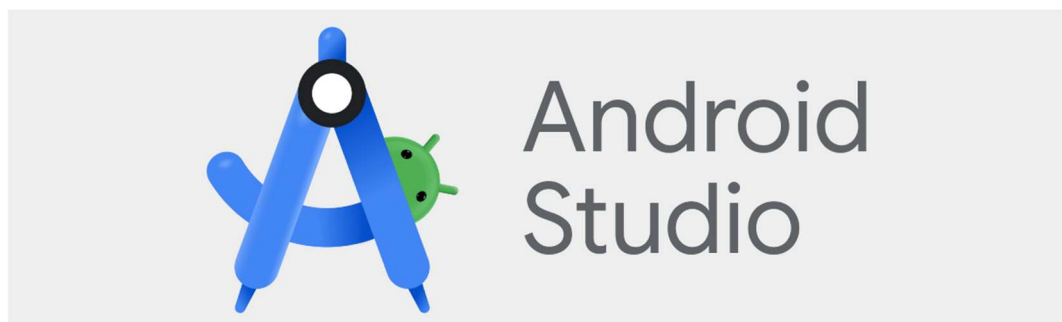


Рисунок 4.1.2 – Android Studio

Android Studio предоставляет разработчику полный набор инструментов, необходимых для эффективной работы. Это включает в себя мощный редактор кода с интеллектуальным автодополнением и анализом, встроенные инструменты отладки для поиска и устранения ошибок, а также обширные возможности для тестирования приложений на различных устройствах и версиях Android. Особенно ценными являются встроенные эмуляторы Android, которые позволяют запускать и тестировать приложение на виртуальных устройствах без необходимости использовать физические девайсы. Интеграция с системой контроля версий Git упрощает командную работу, а средства профилирования производительности помогают оптимизировать приложение. Android Studio полностью поддерживает Jetpack Compose, предоставляя функции предпросмотра композитов в реальном времени, что значительно ускоряет и упрощает разработку UI.

4.2 Архитектура приложения

Проект разрабатываемого приложения имеет структуру, отражённую на рисунке 4.2.1.

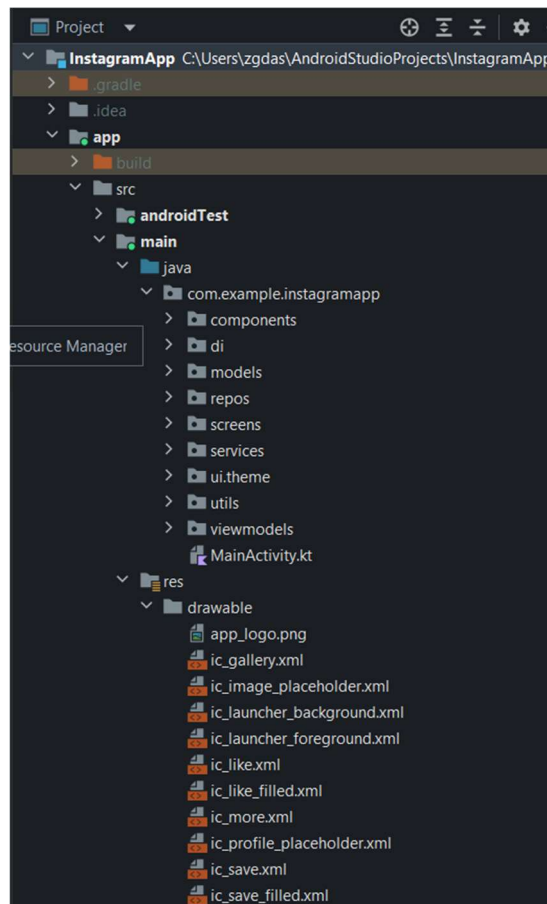


Рисунок 4.2.1 – Структура приложения для обмена изображениями

Архитектура разработанного приложения строго следует принципам MVVM (Model-View-ViewModel), что проявляется в тщательном разделении кода на логические компоненты и строгой направленности зависимостей. Такой подход не только обеспечивает предсказуемость и порядок в кодовой базе, но и значительно упрощает тестирование, расширение функционала и дальнейшее сопровождение проекта.

Основная логика приложения находится в модуле `app/src/main/java/com.example.instagramapp`, который содержит следующие ключевые компоненты, отражающие слои MVVM:

1 Модель (Model) – это слои `models` и `repos`. Каталог `models` содержит сущности данных, то есть классы, описывающие структуру информации, с которой оперирует приложение. Это определения для User (пользователь),

Post (публикация), Comment (комментарий) или Story (история). Эти классы служат «скелетом» для данных, являясь основой бизнес-логики.

Каталог repos (сокращение от repositories, репозитории) реализует механизмы доступа к данным и управления ими. Здесь находятся классы, которые абстрагируют работу с различными источниками данных. Например, UserRepository будет отвечать за получение, сохранение и обновление информации о пользователях, не раскрывая этих деталей верхним слоям. Именно здесь происходит управление данными, обеспечивая их целостность и доступность.

2 Слой Представления (View) сосредоточен в каталоге screens, а также частично в MainActivity.kt.

Каталог screens содержит реализации пользовательского интерфейса для каждого экрана приложения, написанные с использованием Jetpack Compose. Здесь находятся «компосаблы» (composables), которые определяют внешний вид и расположение элементов UI, таких как лента публикаций, профиль пользователя, экран создания поста или поисковая страница.

MainActivity.kt является основной точкой входа для UI, управляя навигацией между экранами, а также инициализируя и отображая корневые элементы пользовательского интерфейса. Роль View в MVVM заключается в пассивном отображении данных и передаче пользовательских действий (например, нажатий кнопок или ввода текста) в ViewModel. View не содержит сложной бизнес-логики и напрямую не взаимодействует с Model.

3 Модель Представления (ViewModel) – это каталог viewmodels. Каталог viewmodels содержит классы ViewModel, которые служат посредниками между слоями View и Model.

Каждый ViewModel (например, HomeViewModel, ProfileViewModel, PostsViewModel) отвечает за управление состоянием конкретного экрана или части UI. ViewModel запрашивает необходимые данные у репозитория (из repos), преобразует их в удобный для отображения формат (например, форматирует даты, обрабатывает изображения) и предоставляет эти данные View через наблюдаемые объекты (StateFlow).

При получении пользовательских действий от View (например, нажатие «лайк», отправка комментария), ViewModel обрабатывает их, инициируя соответствующие операции в репозиториях. Важно, что ViewModel не имеет прямой ссылки на View, что позволяет легко тестировать её логику в изоляции от UI и обеспечивает её устойчивость к изменениям конфигурации (например, повороту экрана).

Помимо основных компонентов MVVM, структура включает и другие важные каталоги:

- components – переиспользуемые UI-компоненты (например, кастомные кнопки, карточки публикаций), которые могут использоваться на разных экранах, способствуя модульности UI;

- di (Dependency Injection) – здесь находится настройка для системы внедрения зависимостей (с использованием Hilt), которая облегчает управление зависимостями между различными слоями и классами, делая код более гибким и тестируемым;

- services - соержжит общую логику, не привязанную к конкретному экрану или сущности репозитория (сервисы для работы с FirebaseStorage).

- ui.theme – определяет общую стилистику приложения, такую как цвета, типографика и формы, что обеспечивает единообразие дизайна.

- utils – содержит вспомогательные классы и функции общего назначения (например, навигация и форматирование времени).

4 Каталог res (resources) содержит все необходимые ресурсы приложения, такие как изображения (drawable), иконки (ic_*.xml для векторных ассетов), которые используются слоем UI для отображения элементов.

5 Отдельный модуль test [unitTest] классами CommentRepositoryTest, PostRepositoryTest, ProfileRepositoryTest, StoryRepositoryTest, UserRepositoryTest демонстрирует приверженность принципам тестирования. Эти тесты проверяют корректность работы репозитория, гарантируя, что бизнес-логика и взаимодействие с данными функционируют правильно, независимо от UI (рисунок 4.2.2). Это критически важно для обеспечения стабильности и надёжности приложения при его дальнейшем развитии.

```
@OptIn(ExperimentalCoroutinesApi::class)
class UserRepositoryTest {

    private lateinit var firebaseAuth: FirebaseAuth
    private lateinit var userRepository: UserRepository

    private val testDispatcher = StandardTestDispatcher()

    @Before
    fun setUp() {
        Dispatchers.setMain(testDispatcher)
        firebaseAuth = mockk(relaxed = true)
        userRepository = UserRepository(firebaseAuth)
    }

    @After
    fun tearDown() {
        Dispatchers.resetMain()
    }
}
```

Рисунок 4.2.2 – UserRepositoryTest

Кроме этого, отдельный модуль `serialization-` с `JsonSerializer`, `XmlSerializer` и `SerializationManager.kt` выносит функциональность сериализации/десериализации данных в отдельную, переиспользуемую библиотеку.

Таким образом, структура проекта `InstagramApp` чётко отражает архитектуру MVVM, обеспечивая логическое разделение слоев, инверсию зависимостей и высокую тестируемость. Это закладывает прочный фундамент для создания масштабируемого, поддерживаемого и надёжного Android-приложения.

4.3 Реализация логики работы приложения

Логика работы разработанного приложения для социальной платформы строится на фундаментальных принципах архитектуры MVVM, обеспечивая чёткое разделение ответственности и предсказуемость потока данных. Это позволяет каждой части системы выполнять свои специализированные задачи, взаимодействуя с другими компонентами через определённые контракты, что значительно упрощает разработку и дальнейшую поддержку.

Взаимодействие начинается, когда пользователь открывает приложение и попадает на стартовый экран, например, `SignInScreen.kt`. Этот экран, являясь частью слоя Представления (`screens`), отображает элементы пользовательского интерфейса, такие как поля для ввода email и пароля, а также кнопки для входа или регистрации (рисунок 4.3.1). `SignInScreen` не содержит бизнес-логики; его единственная задача – отобразить UI и передать пользовательские действия.

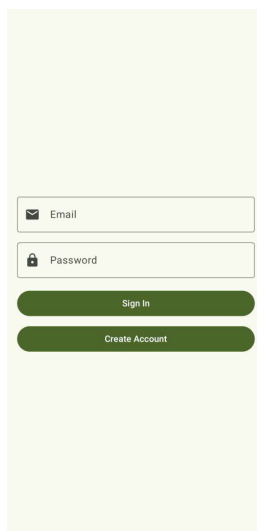


Рисунок 4.3.1 – `SignInScreen`

Когда пользователь вводит свои учетные данные и нажимает кнопку «Войти», SignInScreen передает эти данные соответствующей ViewModel, в данном случае AuthViewModel.kt из каталога viewmodels. AuthViewModel является мозгом этого экрана. Она получает введенные данные, инициирует процесс аутентификации, обращаясь к слою Модели (models и repos). Конкретно, AuthViewModel вызывает метод signIn у UserRepository (из repos), который является частью слоя Модели и отвечает за взаимодействие с сервисом аутентификации, таким как Firebase. UserRepository, в свою очередь, использует UserRemoteDataSource для непосредственного выполнения запроса к серверу.

После того как UserRepository получает ответ от сервера (успешная аутентификация или ошибка), он передает этот результат обратно в AuthViewModel. AuthViewModel обрабатывает полученный результат: если вход успешен, она может обновить внутреннее состояние, указывающее на авторизованного пользователя, и инициировать навигацию к ProfileScreen.kt (рисунок 4.3.2).

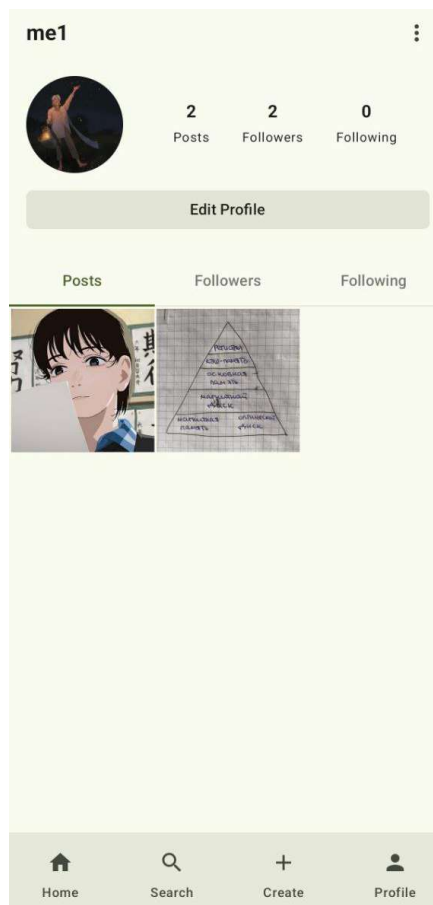


Рисунок 4.3.2 – ProfileScreen

Если произошла ошибка, AuthViewModel обновляет своё состояние, которое затем наблюдается SignInScreen, и отображает сообщение об ошибке пользователю («Неверный логин или пароль», рисунок 4.3.3).

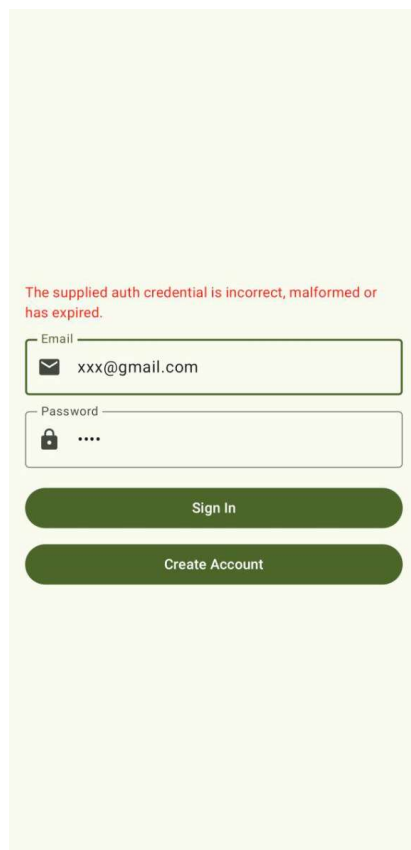


Рисунок 4.3.3 – Экран неверно введенного пароля

Аналогичная логика применима и к другим функциям приложения. Например, когда пользователь просматривает HomeScreen.kt (ленту публикаций), HomeViewModel.kt загружает данные о постах и историях. HomeViewModel обращается к PostRepository и StoryRepository (из repos), которые, в свою очередь, взаимодействуют с соответствующими источниками данных для получения списка публикаций и историй. Эти данные, представленные в виде сущностей (Post, Story из models), передаются обратно в HomeViewModel. HomeViewModel преобразует их в удобный для отображения формат и предоставляет HomeScreen через наблюдаемые данные (StateFlow<List<Post>>). HomeScreen затем автоматически обновляет свои PostItem.kt и StoryCircle.kt (из components), отображая актуальный контент (рисунок 4.3.4).

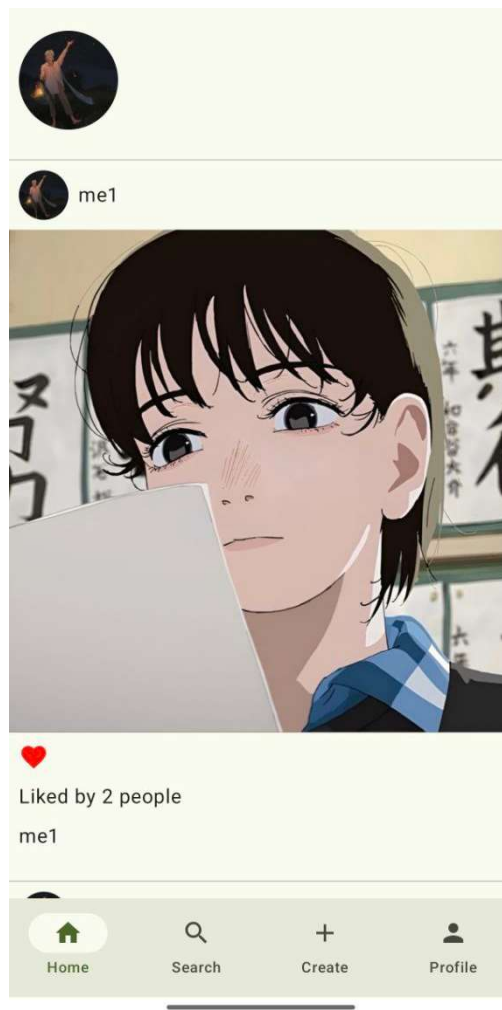


Рисунок 4.3.4 – HomeScreen

Когда пользователь нажимает «лайк» на `PostItem.kt`, это событие передается в `HomeViewModel`. `HomeViewModel` вызывает соответствующий метод `likePost` в `PostRepository`. `PostRepository` отправляет запрос на сервер, обновляет статус лайка и возвращает результат. `HomeViewModel` получает обновленные данные, которые снова отправляются в `HomeScreen`, заставляя `PostItem` обновить визуальное состояние лайка (рисунок 4.3.5).



Рисунок 4.3.5 – Обновление лайка поста

Таким образом, каждый компонент в архитектуре MVVM выполняет свою строго определённую роль:

1 Экраны (screens) (HomeScreen.kt, ProfileScreen.kt и другие) отвечают исключительно за отображение информации и сбор пользовательского ввода. Они «слушают» ViewModel и обновляются в соответствии с её состоянием.

2 ViewModel (viewmodels) (HomeViewModel.kt, ProfileViewModel.kt и другие) является центром логики представления. Она собирает и преобразует данные для View, а также обрабатывает взаимодействия с пользователем, делегируя более сложную бизнес-логику Модели. ViewModel не имеет прямых ссылок на View, что обеспечивает её независимость.

3 Модель (models, repos, services) – это ядровая бизнес-логика и источники данных. models определяет структуру данных, repos абстрагирует работу с данными (будь то Firebase через FirebaseStorageService или локальное хранилище), а services (FirebaseStorageService.kt) предоставляют низкоуровневые API для взаимодействия с внешними службами. Этот слой не зависит ни от View, ни от ViewModel.

4 Модули утилит (utils) и компонентов (components) предоставляют переиспользуемые элементы и вспомогательные функции, которые используются различными слоями приложения, поддерживая модульность и чистоту кода.

5 Процесс регистрации, создания постов (CreatePostsAndStories.kt), поиска (SearchScreen.kt) и управления профилем (ProfileScreen.kt) следует тем же принципам: каждый экран взаимодействует со своей ViewModel (например, CreatePostsAndStoriesViewModel, SearchViewModel, ProfileViewModel), которая, в свою очередь, обращается к соответствующим репозиториям (PostRepository, StoryRepository, UserRepository) для выполнения необходимых операций.

Такая строгая организация логики работы, с однонаправленным потоком данных и чётким разделением ответственности, делает приложение легко расширяемым. Например, добавление новой функции, такой как видео-посты, потребует изменений в models (новая сущность VideoPost), repos (методы для работы с видео), viewmodels (логика для отображения и загрузки видео) и screens (UI для видео). Однако эти изменения будут локализованы в соответствующих слоях, не затрагивая другие части системы, что является ключевым преимуществом выбранной архитектуры.


4.5 Тестирование логики приложения

Тестирование логики приложения является абсолютно критичным этапом в процессе разработки любого программного обеспечения, особенно когда речь идет о социальной платформе, где корректность обработки пользовательских данных, взаимодействие с API и сохранение состояния напрямую влияют на качество сервиса и доверие пользователей. Для приложения, подобного Instagram, жизненно важно убедиться в правильности всех механизмов, от базовых операций, таких как регистрация или создание публикации, до более сложных сценариев, включающих обработку изображений, управление подписками и синхронизацию данных.

Одной из важнейших задач при тестировании логики является проверка корректности всех операций, связанных с пользователями и их контентом. Это включает в себя валидацию процесса аутентификации: необходимо убедиться, что пользователь может успешно зарегистрироваться, войти в систему и выйти из нее, а также что все механизмы восстановления пароля и сброса данных работают без сбоев. Тесты должны охватывать как стандартные случаи, так и граничные ситуации, например, попытки входа с неверными учетными данными или регистрацию с уже существующим именем пользователя.

Особое внимание уделяется проверке механизма работы с публикациями и историями (рисунок 4.5.1). Это включает в себя тестирование процесса создания нового поста: приложение должно корректно обрабатывать загрузку изображений (взаимодействие с FirebaseStorageService из вашего модуля services), сохранять текстовые описания и метаданные (через PostRepository в repos). Важно убедиться, что публикации корректно отображаются в ленте (HomeScreen), что их можно редактировать, удалять, а

также что система лайков и комментариев функционирует безупречно. Тесты должны подтверждать, что при добавлении лайка (HomeViewModel вызывает likePost в PostRepository) эти действия правильно отражаются в базе данных и мгновенно обновляются для других пользователей.

A screenshot of an IDE showing the code for a test class named PostRepositoryTest. The code is written in Kotlin and includes annotations for coroutines and Firebase. It defines several private lateinit var properties for Firestore collections and a PostRepository, and a private val testDispatcher.

```
@OptIn(ExperimentalCoroutinesApi::class)
class PostRepositoryTest {

    private lateinit var firestore: FirebaseFirestore
    private lateinit var collectionReference: CollectionReference
    private lateinit var documentReference: DocumentReference
    private lateinit var postRepository: PostRepository

    private lateinit var postsCollection: CollectionReference
    private lateinit var likesCollection: CollectionReference
    private lateinit var commentsCollection: CollectionReference
    private lateinit var bookmarksCollection: CollectionReference

    private val testDispatcher = StandardTestDispatcher()
}
```

Рисунок 4.5.1 – Корректный результат тестирования постов

Аналогично проверяется логика работы с историями (рисунок 4.5.2), их просмотр (StoryViewerScreen.kt) и загрузка (CreatePostsAndStories.kt).

A screenshot of an IDE showing the code for a test class named StoryRepositoryTest. The code is written in Kotlin and includes annotations for coroutines and Firebase. It defines several private lateinit var properties for Firestore collections and a StoryRepository, and a private val testDispatcher.

```
@OptIn(ExperimentalCoroutinesApi::class)
class StoryRepositoryTest {

    private lateinit var firestore: FirebaseFirestore
    private lateinit var storiesCollection: CollectionReference
    private lateinit var documentReference: DocumentReference
    private lateinit var storyRepository: StoryRepository

    private val testDispatcher = StandardTestDispatcher()
}
```

Рисунок 4.5.2 – Корректный результат тестирования историй

Кроме функциональных аспектов, тестирование включает проверку сериализации и десериализации данных, что особенно важно для сохранения и загрузки состояния приложения, а также для обмена данными с бэкендом. Модуль serialization-library с JsonSerializer и XmlSerializer играет здесь ключевую роль. Тесты для этих компонентов гарантируют, что объекты, такие как User и Post, правильно преобразуются в текстовый формат (например, JSON) для передачи по сети или сохранения, и что из этого формата они корректно восстанавливаются без потери информации или ошибок. Такая проверка обеспечивает надежность работы с данными, что является основой стабильности всего приложения и сохранения пользовательского контента.

Все упомянутые аспекты логики приложения тестируются с помощью автоматизированных юнит-тестов, которые располагаются в вашем модуле `test [unitTest]`. Наличие классов вроде `PostRepositoryTest`, `ProfileRepositoryTest`, `StoryRepositoryTest` и `UserRepositoryTest` свидетельствует о систематическом подходе к проверке каждого компонента слоя Модели. Эти тесты позволяют регулярно и быстро проверять правильность работы различных частей кода при каждом изменении, гарантируя, что новые функции не нарушают существующий функционал.

Помимо автоматизированных тестов, ручное тестирование также проводится с целью проверки пользовательского опыта и визуальной корректности работы всех элементов интерфейса. В ходе ручного тестирования тщательно проверяются основные сценарии взаимодействия, такие как создание поста, просмотр ленты, взаимодействие с профилем, а также граничные случаи и нестандартные ситуации (например, попытка загрузки слишком большого изображения, потеря интернет-соединения во время загрузки). Особое внимание уделяется тому, чтобы интерфейс был интуитивно понятным, анимации плавными, а реакция приложения на действия пользователя — быстрой и предсказуемой. Ручное тестирование также включает оценку производительности приложения на различных устройствах и версиях Android, чтобы убедиться в его стабильной работе в реальных условиях и совместимости.

Комплексная проверка всех аспектов логики приложения, сочетающая автоматизированные и ручные методы, позволила обеспечить стабильную, корректную и высококачественную работу социальной платформы, соответствующую ожиданиям пользователей.

4.6 Анализ результатов работы

Анализ результатов разработки и тестирования показал, что созданное приложение для социальной платформы полностью соответствует основным требованиям по функциональности, стабильности и удобству использования. Все ключевые аспекты, предусмотренные в рамках проекта, успешно реализованы и прошли проверку, не выявив критических ошибок, что подтверждает корректность выбранных архитектурных решений и эффективность их воплощения.

Основной функционал социальной платформы, связанный с управлением контентом, также функционирует корректно. Пользователи могут беспрепятственно создавать новые публикации и истории, загружать изображения, добавлять текстовые описания и взаимодействовать с контентом других пользователей. В ходе тестирования было подтверждено, что механики добавления лайков и управления подписками работают корректно, а изменения в данных мгновенно отображаются в пользовательском

интерфейсе. Это свидетельствует о правильной реализации взаимодействия между слоями View, ViewModel и Model, а также об эффективной синхронизации данных с бэкендом (Firebase).

Кроме того, были успешно проверены механизмы сохранения и загрузки данных с использованием разработанной библиотеки сериализации. Тесты подтвердили, что объекты данных (пользователи, публикации) корректно преобразуются для хранения и передачи, а затем безошибочно восстанавливаются. Это гарантирует сохранность пользовательских данных и стабильность работы приложения при любых сценариях.

В процессе комплексного тестирования, включавшего как автоматизированные юнит-тесты репозитория и ViewModel, так и ручные проверки пользовательского интерфейса, были выявлены и оперативно устранены небольшие недочёты. Эти доработки были связаны преимущественно с визуальным оформлением и улучшением пользовательского опыта, например, оптимизация загрузки изображений в ленте, улучшение анимаций переходов между экранами или уточнение формулировок в сообщениях об ошибках. Проведённое тестирование также подтвердило, что интерфейс приложения интуитивно понятен и не вызывает затруднений у пользователей, что достигается благодаря продуманной навигации и стандартным элементам дизайна.

В целом, достигнутые результаты свидетельствуют о том, что проект успешно выполнил поставленные цели по разработке функционального и стабильного приложения для социальной платформы. Все выявленные в процессе тестирования замечания были своевременно проанализированы и исправлены, что позволило создать высококачественный программный продукт. Разработанное приложение готово к дальнейшему использованию, а его модульная архитектура на основе MVVM обеспечивает прочную основу для будущего расширения функционала и адаптации к новым требованиям.

ЗАКЛЮЧЕНИЕ

В заключение следует отметить, что в ходе работы над проектом по разработке приложения для социальной платформы была проделана всесторонняя и комплексная работа, охватывающая этапы проектирования архитектуры, непосредственного программирования и тщательного тестирования. Созданное приложение полностью соответствует поставленным техническим требованиям, обеспечивая стабильную и надёжную работу всех ключевых функций, присущих современным социальным сервисам. Благодаря продуманной логике взаимодействия компонентов и корректной реализации алгоритмов, приложение демонстрирует предсказуемое и эффективное поведение во всех основных сценариях использования, что было многократно подтверждено результатами проведённых тестов. Это свидетельствует о высоком качестве реализации и внимании к деталям на всех этапах разработки.

В перспективе проект обладает значительным потенциалом для дальнейшего развития и улучшения. В приложение можно внедрить расширенные функции взаимодействия между пользователями, такие как личные сообщения, групповые чаты или видеозвонки, что значительно расширит его социальную составляющую. Также возможно добавление новых типов контента, например, коротких видеороликов или прямых трансляций, что сделает платформу более разнообразной и привлекательной. В дальнейшем, для масштабирования и обеспечения высокой доступности, рекомендуется рассмотреть переход на более мощные серверные решения и оптимизацию архитектуры бэкенда. Также рекомендуется продолжать работу по тестированию с привлечением реальных пользователей, что поможет собрать ценные отзывы и внести дополнительные улучшения, направленные на совершенствование пользовательского опыта. В целом, достигнутые результаты работы подтверждают успешность выполнения поставленных задач и создают прочную основу для дальнейшей эволюции проекта в рамках полноценной социальной платформы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Обзор рынка социальных сетей в 2024 году [Электронный ресурс]. Режим доступа: <https://www.statista.com/outlook/digital-markets/social-networks/worldwide>. Дата доступа: 15.03.2025.
- [2] Тенденции роста мобильных социальных приложений [Электронный ресурс]. Режим доступа: <https://www.appannie.com/en/go/state-of-mobile/2024>. Дата доступа: 25.03.2025.
- [3] Instagram: история успеха [Электронный ресурс]. Режим доступа: <https://www.businessinsider.com/history-of-instagram-2016-3>. Дата доступа: 05.04.2025.
- [4] Топ-10 социальных сетей по количеству активных пользователей [Электронный ресурс]. Режим доступа: <https://www.pewresearch.org/internet/2024/02/01/social-media-use-2024/>. Дата доступа: 10.04.2025.
- [5] Объектно-ориентированное программирование: базовые концепции [Электронный ресурс]. Режим доступа: <https://javarush.com/groups/posts/1913-оор-концепції>. Дата доступа: 15.04.2025.
- [6] Принципы ООП в Kotlin [Электронный ресурс]. Режим доступа: <https://kotlinlang.org/docs/object-oriented-programming.html>. Дата доступа: 25.04.2025.
- [7] MVVM (Model-View-ViewModel) в Android-разработке [Электронный ресурс]. Режим доступа: <https://developer.android.com/topic/libraries/architecture/viewmodel>. Дата доступа: 05.05.2025.
- [8] Что такое Jetpack Compose? [Электронный ресурс]. Режим доступа: <https://developer.android.com/jetpack/compose/documentation>. Дата доступа: 10.05.2025.
- [9] Kotlin как основной язык для Android-разработки [Электронный ресурс]. Режим доступа: <https://developer.android.com/kotlin>. Дата доступа: 10.05.2025.
- [10] Android Studio: официальное руководство [Электронный ресурс]. Режим доступа: <https://developer.android.com/studio/intro>. Дата доступа: 10.05.2025.