Министерство образования и науки Нижегородской области

Государственное бюджетное профессиональное образовательное учреждение

**«Нижегородский Губернский колледж»**

Методическая комиссия Информатика и вычислительная техника

Допущен к защите:

преподаватель

\_\_\_\_\_\_\_\_\_\_Ю. С. Мамшева

« » 2024 г.

РАЗРАБОТКА МОБИЛЬНОГО ПРИЛОЖЕНИЯ

**ПРОЕКТНАЯ РАБОТА**

по МДК 01.03 «Разработка мобильных приложений»

Студент: \_\_\_\_\_\_\_\_\_\_\_\_ Е.С.Колиниченко 27.11.2024

(подпись)

Специальность, группа: 09.02.07, 43П

Нижний Новгород

2024

СОДЕРЖАНИЕ

# ГЛАВА 1. ПОСТАНОВКА ЦЕЛЕЙ И ЗАДАЧ

1. **Техническое задание**
   1. **Общие сведения**
      1. Наименование системы

Полное наименование: Meow Mates

Краткое наименование: MM.

* + 1. Наименование организаций

Заказчик: Мамшева Юлия Сергеевна

Разработчик: Колиниченко Екатерина Сергеевна

* + 1. Плановые сроки начала и окончания работы

Начало работ: 2 октября 2024

Окончание работ: запланировано на момент выполнения всех стадий разработки, что включает 25 очных пар по 1,5 часа каждая. Таким образом, окончание работ будет определено по завершении всех запланированных занятий.

* + 1. Порядок оформления и предъявления заказчику результатов работ

Работы по созданию мобильного приложения Meow Mates сдаются Разработчиком поэтапно в соответствии с календарным планом. По окончании каждого этапа Разработчик предоставляет Заказчику соответствующую часть отчетных документов, ссылки на репозитории, а также демонстрирует результаты, разработанные в рамках конкретного этапа, которые прописаны и согласованы в документе с Заказчиком.

* 1. **Назначение и цели создания системы**
     1. Актуальность

В современном мире наблюдается рост интереса к домашним животным, особенно к кошкам. Владельцы все чаще ищут способы социализации своих питомцев и создания для них дружеских связей. Мобильное приложение Meow Mates предоставляет уникальную платформу, где владельцы могут зарегистрировать своих пушистиков и находить подходящих "партнеров" для их котиков. Это не только способствует улучшению эмоционального состояния животных, но и создает сообщество любителей кошек, где они могут обмениваться опытом и советами.

Таким образом, разработка Meow Mates актуальна как для владельцев кошек, так и для их питомцев, стремящихся к социальной активности.

* + 1. Цели и задачи

Цель: Создание мобильного приложения, предназначенного для помощи владельцам кошек в поиске подходящих "партнеров" для их питомцев.

Задачи, которые можно выделить в контексте данной цели:

1. Создание репозитория на СКВ
2. Разработка интерфейса приложения для удобной регистрации питомцев.
3. Создание системы профилей для кошек с возможностью добавления фотографий и описаний.
4. Реализация алгоритма поиска и сопоставления питомцев по интересам и характеристикам.
5. Внедрение системы сообщений для общения между владельцами.
6. Проектирование и реализация серверной части приложения.
7. Тестирование серверной части приложения.
8. Обеспечение безопасности данных пользователей.
9. Разработка документации по использованию приложения.
   1. **Характеристика объектов автоматизации**

Заказчиком является компания, специализирующаяся на разработке приложений для домашних животных.

Объекты автоматизации в приложении Meow Mates могут включать в себя следующие элементы:

1. Интерфейс пользователя.

2. Модули регистрации и поиска питомцев.

3. Система управления профилями пользователей.

* 1. **Требования к системе**
     1. Требования к системе в целом
        1. Требования к структуре и функционированию системы

Система Meow Mates должна быть централизованной, т.е. все данные должны располагаться в центральном хранилище, использующем Supabase.

Система должна быть построена на двух уровнях: владелец питомца и питомец (кот).

Источниками данных для системы должны быть:

* Supabase как облачная база данных для хранения информации о пользователях и их питомцах.

Система должна поддерживать:

* работу пользователей в режиме – 24 часов в день, 7 дней в неделю (24х7);
* выполнение своих функций – регистрацию пользователей, регистрацию питомцев, поиск пар;
* хранение данных, предоставление отчетности.

Для обеспечения высокой надежности функционирования системы как в целом, так и её отдельных компонентов должно обеспечиваться выполнение требований по диагностированию её состояния.

Диагностирование системы должно осуществляться с использованием встроенных средств мониторинга, предоставляемых Supabase.

* + - 1. Требования к численности и квалификации персонала системы и режиму его работы

В состав персонала, необходимого для обеспечения эксплуатации Meow Mates, необходимо выделение следующих ответственных лиц:

* Главный администратор – 1 человек.
* Технические специалисты для поддержки приложения.

К квалификации персонала, эксплуатирующего систему, предъявляются следующие требования:

* Конечный пользователь — знание основ работы с мобильными приложениями и умение пользоваться смартфоном.
  + - 1. Показатели назначения

В зависимости от различных вероятных условий система должна выполнять требования, приведенные в таблице 1.

Таблица 1 – Требования к сохранению работоспособности системы

|  |  |
| --- | --- |
| Вероятное условие | Требование |
| Нарушения в работе системы внешнего электроснабжения серверного оборудования продолжительностью до 15 мин. | Функционирование в полном объеме. |
| Нарушения в работе системы внешнего электроснабжения серверного оборудования продолжительностью до 5 ч. | Аварийное функционирование системы некоторое время, сохранение всех данных. |
| Выход из строя сервера подсистемы хранения данных. | Уведомление администратора о проблеме и автоматическое восстановление данных из резервной копии. |
| Уязвимость системы из-за устаревшего ПО. | Регулярное обновление программного обеспечения и библиотек приложений. |

* + - 1. Требования к надежности

Уровень надежности мобильного приложения Meow Mates должен достигаться согласованным применением организационных, организационно-технических мероприятий и программных средств.

Надежность должна обеспечиваться за счет:

* регулярного обновления программного обеспечения;
* резервного копирования данных в облачном хранилище (Supabase);
* мониторинга работы приложения и его компонентов;
* тестирования на устойчивость к сбоям и ошибкам.

Время устранения отказа должно быть следующим:

* при перерыве и выходе за установленные пределы параметров электропитания — не более 150 минут;
* при сбоях в работе приложения — не более 24 часов.

Система должна соответствовать следующим параметрам:

* среднее время восстановления 2 часов;
* время наработки на отказ 1 час.

Перечень аварийных ситуаций, по которым регламентируются требования к надежности

При работе системы возможны следующие аварийные ситуации, которые влияют на надежность работы системы:

* сбой в работе сервера Supabase;
* сбой в работе мобильного устройства пользователя;
* ошибки в коде приложения, не выявленные при тестировании;
* проблемы с сетью, влияющие на доступ к облачным данным.

Требования к надежности технических средств и программного обеспечения:

К надежности мобильного приложения предъявляются следующие требования:

* приложение должно быть разработано с учетом принципов устойчивости к сбоям;
* необходимо проводить регулярные обновления для устранения уязвимостей и повышения стабильности работы;
* приложение должно иметь встроенные механизмы для автоматического восстановления данных после сбоев.

Эти требования обеспечат высокую степень надежности и стабильности работы мобильного приложения Meow Mates, что является критически важным для удовлетворения потребностей пользователей.

* + - 1. Требования к эргономике и технической эстетике

Подсистема должна обеспечивать удобный для конечного пользователя интерфейс, отвечающий следующим требованиям.

В части внешнего оформления:

* интерфейсы подсистем должен быть типизированы;
* должно быть обеспечено наличие русскоязычного интерфейса пользователя;
* должен использоваться понятный шрифт;
* размер шрифта должен быть: 20 пт;
* цветовая палитра должна быть привлекательной и гармоничной.

В части диалога с пользователем:

* при возникновении ошибок в работе подсистемы должно выводиться сообщение с наименованием ошибки и с рекомендациями по её устранению на русском языке.
  + - 1. Требования к эксплуатации, техническому обслуживанию, ремонту и хранению компонентов системы

Условия эксплуатации, а также виды и периодичность обслуживания мобильного приложения Meow Mates должны соответствовать требованиям по эксплуатации, техническому обслуживанию и ремонту, изложенным в документации разработчика.

Приложение должно быть доступно для пользователей на мобильных устройствах с операционными системами Android.

Для обеспечения надежности работы приложения необходимо:

Регулярно обновлять приложение для устранения уязвимостей и добавления новых функций.

Проводить тестирование приложения на различных устройствах для обеспечения совместимости и стабильности работы.

Обеспечивать резервное копирование данных пользователей в облачном хранилище (Supabase) для предотвращения потери информации.

Состав, место и условия хранения данных пользователей определяются в соответствии с политикой конфиденциальности и правилами использования приложения.

Для обеспечения бесперебойной работы приложения необходимо следить за состоянием серверов Supabase, которые обеспечивают хранение данных. В случае возникновения сбоев или проблем с доступом к серверу, пользователи должны получать уведомления о статусе работы приложения через интерфейс.

Техническое обслуживание приложения включает в себя:

* Мониторинг производительности и отзывчивости приложения.
* Обновление библиотек и зависимостей, используемых в приложении.
* Проведение регулярных проверок безопасности для защиты данных пользователей от несанкционированного доступа.

Все действия по техническому обслуживанию и обновлению приложения должны документироваться для обеспечения прозрачности процессов и возможности анализа в случае возникновения проблем.

* + - 1. Требования к защите информации от несанкционированного доступа

Обеспечение информационной безопасности системы Meow Mates должно удовлетворять следующим требованиям:

Защита системы должна обеспечиваться комплексом программно-технических средств и поддерживающих их организационных мер.

Защита системы должна обеспечиваться на всех технологических этапах обработки информации и во всех режимах функционирования, включая ремонтные работы.

Программно-технические средства защиты не должны существенно ухудшать основные функциональные характеристики системы (надежность, быстродействие).

Разграничение прав доступа пользователей должно строиться по принципу "что не разрешено, то запрещено".

Средства антивирусной защиты должны быть установлены на всех рабочих местах пользователей и администраторов системы Meow Mates и должны обеспечивать:

* централизованное управление сканированием;
* централизованную автоматическую инсталляцию клиентского ПО;
* централизованное автоматическое обновление вирусных сигнатур;
* ведение журналов вирусной активности;
* администрирование всех антивирусных продуктов.
  + - 1. Требования к защите от влияния внешних воздействий

Применительно к программно-аппаратному окружению Системы предъявляются следующие требования к защите от влияния внешних воздействий.

Требования по стойкости, устойчивости и прочности к внешним воздействиям: Система должна иметь возможность функционирования при колебаниях напряжения электропитания в пределах от 155 до 265 В (220 ± 20 % - 30 %);

* + - 1. Требования по стандартизации и унификации

Разработка приложения должна осуществляться с использованием стандартных методологий функционального моделирования: IDEF0, DFD и информационного моделирования IE в рамках рекомендаций по стандартизации.

Для разработки пользовательских интерфейсов должны использоваться возможности Jetpack Compose для Kotlin.

* + - 1. Дополнительные требования

Приложение должно разрабатываться и эксплуатироваться на уже имеющемся у Заказчика аппаратно-техническом комплексе (смартфоны). Необходимо создать отдельные зоны разработки и тестирования приложения Meow Mates.

* + - 1. Требования безопасности

При внедрении, эксплуатации и обслуживании технических средств приложения должны выполняться меры электробезопасности в соответствии с действующими нормами безопасности.

Аппаратное обеспечение приложения должно соответствовать требованиям пожарной безопасности в производственных помещениях согласно актуальным стандартам.

Должно быть обеспечено соблюдение общих требований безопасности при обслуживании приложения в процессе эксплуатации.

* + 1. Требования к функциям, выполняемым системой

1. Авторизация и регистрация, выход из системы:

* Регистрация в системе с проверкой на корректность введенных данных;
* Авторизация в системе с хэшированием пароля
* Возможность выхода из системы

1. Регистрация питомцев в системе:

* Возможность добавления нового котика с указанием его характеристик (имя, порода, возраст, фото).
* Возможность редактирования информации о питомце.
* Возможность удаления питомца из профиля.

1. Просмотр и поиск котиков

* Возможность просмотра списка котиков поблизости.
* Возможность сортировки и фильтрации котиков по различным параметрам (возраст, порода, пол).
* Возможность просмотра профиля других котиков с их характеристиками.

1. Просмотр профиля владельца:

* Возможность редактирования информации о владельце (имя, контактные данные).
* Просмотр статистики взаимодействия с другими пользователями (например, количество добавленных питомцев, количество просмотров профиля).
  + 1. Требования к видам обеспечения
       1. Требования к информационному обеспечению

База данных должна обеспечивать надежное хранение информации о пользователях и их питомцах, а также статистику взаимодействия между ними. Интерфейс приложения должен быть интуитивно понятным и удобным для пользователей, что позволит им быстро освоить работу с системой.

* + - 1. Требования к аппаратному обеспечению

Мобильные устройства (смартфоны) с достаточной производительностью для работы приложения Meow Mates. Сервер для хранения данных в облаке (Supabase) и обеспечения доступа к системе через интернет.

* + - 1. Требования к программному обеспечению

Специализированное программное обеспечение для автоматизации процесса регистрации питомцев и их поиска, а также для управления данными пользователей и питомцев.

* + - 1. Требования к контролю, хранению, обновлению и восстановлению данных

Система должна протоколировать все события, связанные с изменением информационного наполнения, и иметь возможность восстанавливать свое состояние в случае сбоя, используя ранее запротоколированные изменения данных. Хранение исторических данных должно производиться не более чем за 5 (пять) предыдущих лет; по истечении данного срока данные должны переходить в архив.

* + - 1. Требования к методическим материалам

Подготовка подробных инструкций по работе с мобильным приложением для пользователей различного уровня подготовки, включая руководство по установке и настройке приложения Meow Mates.

Эти требования обеспечивают функциональность и надежность мобильного приложения Meow Mates, а также удобство использования для владельцев кошек при поиске подходящих "партнеров" для своих питомцев.

* 1. **Стадии разработки системы, ресурсы, роли**



Основные стадии разработки системы:

1. Проектирование целей и задач;
2. Разработка макета приложения;
3. Разработка серверной части приложения;
4. Тестирование серверной части приложения;
5. Разработка мобильного приложения;
6. Тестирование мобильного приложения;

В качестве среды разработки была выбрана Android Studio. Приложение разрабатывается на языке программирования Kotlin с использованием Supabase в роли хранилища данных.

Ресурсы:

* IDE: Android Studio
* Фреймворк: Jetpack Compose
* Паттерн: MVVM
* Язык: Kotlin
* База данных: Supabase
* Документация: Microsoft Word, Markdown

Ответственность в команде:

Разработчик Екатерина ответственна за все аспекты разработки, включая проектирование, программирование функционала, тестирование и документирование.

Распределение ролей представлено в таблице 2.

Таблица 2 – Распределение ролей участников

|  |  |  |
| --- | --- | --- |
| **Стадия** | **Время** | **Участники** |
| **Постановка целей и задач** | | |
| Разработка технического задания | 2 часа | Екатерина |
| Выбор паттерна проектирования | 30 минут | Екатерина |
| Проектирование требований к серверной части приложения | 30 минут | Екатерина |
| Проектирование требований к мобильному приложению | 30 минут | Екатерина |
| Выбор для реализации ПО и обоснование выбора | 30 минут | Екатерина |
| **Разработка макета приложения** | | |
| Разработка макета каждого экрана разрабатываемого мобильного приложения (выбор шрифтов, цветовой палитры) | 6 часов | Екатерина |
| **Разработка серверной части приложения** | | |
| Проектирование серверной части в Supabase | 2 часа | Екатерина |
| Реализация серверной части в мобильном приложении | 2 часа | Екатерина |
| **Тестирование серверной части приложения** | | |
| Разработка документации для тестирования сервера приложения (тест-план, тестовый набор, тест-кейс) | 3 часа | Екатерина |
| Спроектировать и реализовать 10 unit-test к серверу приложения | 1 час | Екатерина |
| **Разработка мобильного приложения** | | |
| Подключение шрифтов и настройка ранее выбранных цветов | 1 час | Екатерина |
| Верстка макета | 15 часов | Екатерина |
| Подключение библиотек из интернета | 1 час | Екатерина |
| Подключение к серверной части приложения | 5 часов | Екатерина |
| Разработка вспомогательных методов, и методов для обработки данных с сервера. | 4 часа | Екатерина |
| **Тестирование мобильного приложения** | | |
| Разработка тестовой документации (тест план, тестовый набор, тест-кейс) | 2 часа | Екатерина |
| Проектирование и реализация 5 unit-test и 5 ui-test к мобильному приложению | 2 часа | Екатерина |

* 1. **Порядок контроля и приёмки системы**
     + 1. Виды и объем испытаний системы

Система подвергается испытаниям следующих видов:

1. Предварительные испытания.

2. Опытная эксплуатация.

3. Приемочные испытания.

Состав, объем и методы предварительных испытаний системы определяются документом «Программа и методика испытаний», разрабатываемым на стадии «Рабочая документация».

Состав, объем и методы опытной эксплуатации системы определяются документом «Программа опытной эксплуатации», разрабатываемым на стадии «Ввод в действие».

Состав, объем и методы приемочных испытаний системы определяются документом «Программа и методика испытаний», разрабатываемым на стадии «Ввод в действие» с учетом результатов проведения предварительных испытаний и опытной эксплуатации.

* + - 1. Требования к приемке работ по стадиям

Участники испытаний: организации Заказчика и Разработчика.

Место проведения: на территории Заказчика.

Таблица 3 – Требования к приемке работ по стадиям

|  |  |  |
| --- | --- | --- |
| Стадия испытаний | Порядок согласования документации | Статус приемочной комиссии |
| Предварительные испытания | Проведение предварительных испытаний.  Фиксирование выявленных неполадок в Протоколе испытаний.  Устранение выявленных неполадок.  Проверка устранения выявленных неполадок.  Принятие решения о возможности передачи приложения в опытную эксплуатацию.  Составление и подписание Акта приемки приложения в опытную эксплуатацию. | Экспертная группа |
| Опытная эксплуатация | Проведение опытной эксплуатации.  Фиксирование выявленных неполадок в Протоколе испытаний.  Устранение выявленных неполадок.  Проверка устранения выявленных неполадок.  Принятие решения о готовности приложения к приемочным испытаниям.  Составление и подписание Акта о завершении опытной эксплуатации приложения. | Группа тестирования |
| Приемочные испытания | Проведение приемочных испытаний.  Фиксирование выявленных неполадок в Протоколе испытаний.  Устранение выявленных неполадок.  Проверка устранения выявленных неполадок.  Принятие решения о возможности передачи приложения в промышленную эксплуатацию.  Составление и подписание Акта о завершении приемочных испытаний и передаче приложения в промышленную эксплуатацию.  Оформление Акта завершения работ. | Приемочная комиссия |

* 1. **Требования к составу и содержанию работ по подготовке объекта автоматизации к вводу системы в действие**
     + 1. Технические мероприятия

1. Установка и настройка специализированного программного обеспечения Meow Mates.
2. Интеграция новой системы с уже существующими информационными системами, если это необходимо.
3. Обеспечение безопасности данных, включая регулярное резервное копирование информации в Supabase.
   * + 1. Организационные мероприятия
4. Определение ответственных лиц за внедрение, обучение и поддержку системы.
5. Разработка процедур работы с приложением, включая правила добавления питомцев, поиск и фильтрацию котиков.
6. Проведение обучающих семинаров и тренингов для пользователей по работе с приложением.
7. Мониторинг работы приложения, анализ его эффективности и внесение необходимых корректировок.
   * + 1. Дополнительные мероприятия
8. Проведение обучения пользователей по работе с приложением.
9. Планирование и организация регулярного технического обслуживания и поддержки приложения.
   1. **Требования к документированию**

Документирование MM-Simulator должно содержать:

1. Разработка плановой документации, описывающей применение и варианты использования приложения.
2. Разработка тестовой документации, содержащей тест-план, тестовую стратегию, тестовые сценарии и тестовые наборы.
3. Использование документации в репозитории в формате README файла, с ссылкой на более подробную документацию в формате Markdown.
   1. **Источники разработки**

Настоящее Техническое Задание разработано на основе следующих документов и информационных материалов:

* ГОСТ 34.602-89 «Техническое задание на создание автоматизированной системы».
* ГОСТ 12.4.131-83 «Система стандартов безопасности труда. Средства индивидуальной защиты. Общие требования к проверке».
* ГОСТ 12.1.004-91 «ССБТ. Пожарная безопасность. Общие требования».
* ГОСТ 24.701-86 «Надежность автоматизированных систем управления».

## Выбор ПО и паттерна проектирования

Для мобильного приложения «Meow Mates2 был выбран паттерн проектирования MVVM (Model-View-ViewModel). Этот паттерн позволяет эффективно разделять логику приложения и пользовательский интерфейс, что особенно важно для мобильных приложений, где удобство взаимодействия с пользователем имеет первостепенное значение.

Обоснование выбора MVVM:

1. Четкое разделение ответственности:

* Model отвечает за данные и бизнес-логику приложения (например, информация о питомцах, пользователях и их взаимодействиях).
* View представляет пользовательский интерфейс (UI), который отображает данные и реагирует на действия пользователя.
* ViewModel служит связующим звеном между Model и View, обеспечивая привязку данных и управление состоянием UI.

1. Упрощение тестирования: Паттерн MVVM позволяет легко тестировать бизнес-логику, так как ViewModel может быть протестирован независимо от View. Это упрощает процесс написания юнит-тестов.
2. Поддержка реактивного программирования: Использование LiveData и других реактивных компонентов в Android позволяет View автоматически обновляться при изменении данных в ViewModel, что улучшает пользовательский опыт.
3. Удобство разработки: MVVM хорошо интегрируется с библиотеками Android Jetpack, такими как LiveData и ViewModel, что упрощает разработку и уменьшает количество кода.

## ВЫБОР ДЛЯ РЕАЛИЗАЦИИ ПО

Для реализации мобильного приложения «Meow Mates» были выбраны следующие технологии:

**3.1. Язык программирования: Kotlin**

Обоснование выбора:

* Kotlin является официальным языком для разработки под Android, что обеспечивает хорошую поддержку со стороны Google.
* Он предлагает лаконичный синтаксис и современные возможности, такие как расширения функций и корутины для асинхронного программирования.
* Kotlin совместим с Java, что позволяет использовать существующие библиотеки и фреймворки.

**3.2. Среда разработки: Android Studio**

Обоснование выбора:

* Android Studio предоставляет мощные инструменты для разработки, отладки и тестирования приложений.
* Она включает в себя встроенные средства для работы с Gradle, что упрощает управление зависимостями и сборкой проекта.
* Поддерживает интеграцию с Git для управления версиями кода.

**3.3. Хранилище данных: Supabase**

Обоснование выбора:

* Supabase предоставляет облачное решение для хранения данных с поддержкой PostgreSQL, что обеспечивает надежность и масштабируемость.
* Он предлагает простую интеграцию с приложениями на Kotlin через RESTful API или библиотеку клиентского SDK.
* Supabase включает функции аутентификации и управления пользователями, что упрощает реализацию безопасности в приложении.

**3.4. UI-фреймворк: Jetpack Compose**

Обоснование выбора:

* Jetpack Compose позволяет создавать пользовательские интерфейсы декларативным способом, что упрощает разработку и поддержку UI.
* Он интегрируется с другими компонентами Jetpack, такими как LiveData и ViewModel, что делает его идеальным выбором для использования с паттерном MVVM.
* Compose обеспечивает высокую производительность и возможность кастомизации интерфейса без необходимости написания большого объема кода.

# ГЛАВА 2. РАЗРАБОТКА МАКЕТА

## Макет каждого экрана разрабатываемого приложения.

Макет был выполнен в Figma и представлен по ссылке: <https://www.figma.com/design/w2HXW4Iq7LtwhHI8sro678/Meow-mates?node-id=0-1&t=nz9VY0NWUA284gup-1> .

## Выбор цветовой палитры

Разработаны две цветовые темы: тёмная (Рис. 1 ниже) и светлая (Рис. 2 ниже).

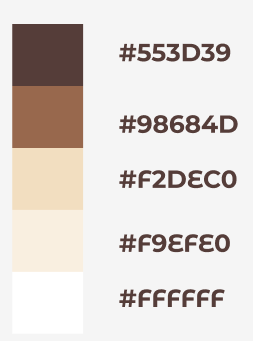


Рисунок 1. Цветовая схема светлой т емы приложения

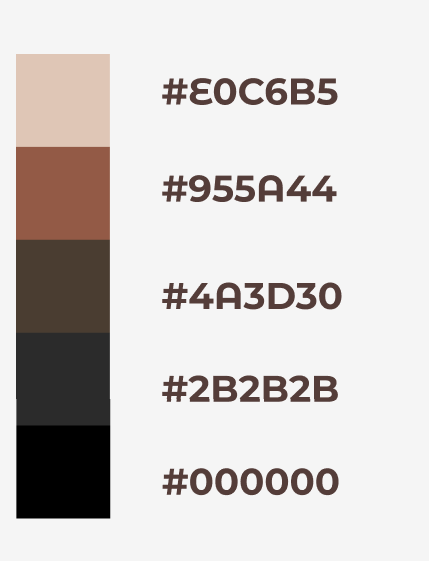


Рисунок 2. Цветовая схема темной темы приложения

## Выбор шрифта для заголовков и основного текста

Шрифт, используемый в макете приложения указан на Рисунке 3 ниже.



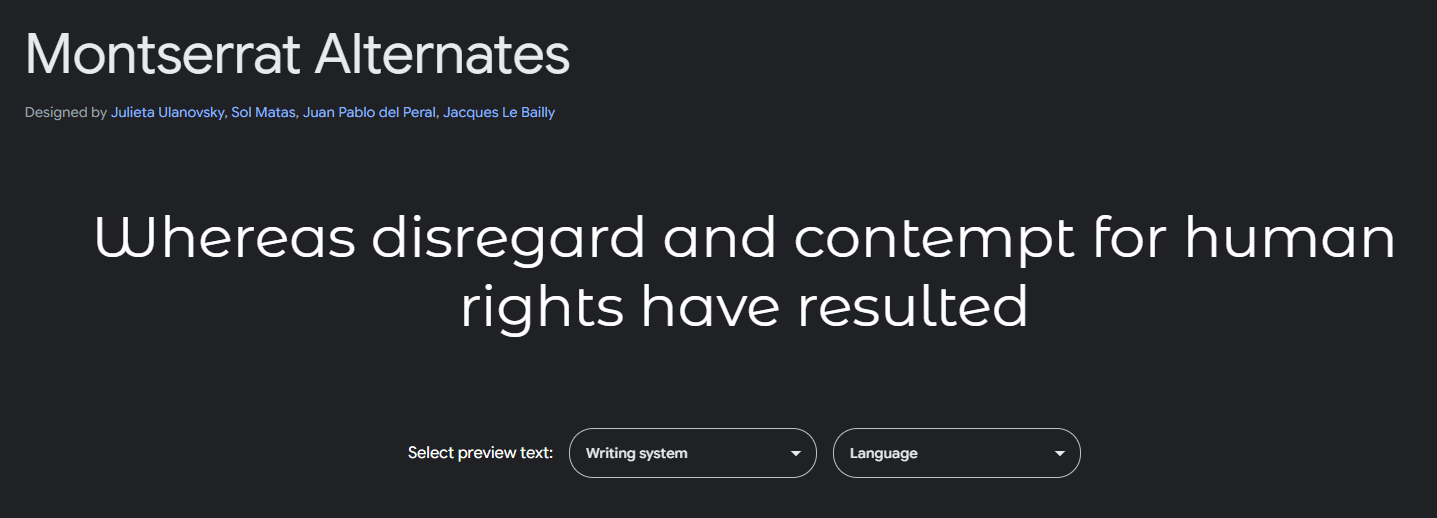


Рисунок 3. Шрифты приложения.

## Элементы формы

В на отжельном экране были вынесены все элементы приложения (иконки, картинки по умолчанию). Их можно увидеть на Рис. 5 ниже



Рисунок 5. Элементы формы.

## Разработка логотипа

Как такогого логитипа у приложения нет, но приложение активно использует название Meow Mates, написанное специальным шрифтом Cat Regular. Выглядит следующим образом:



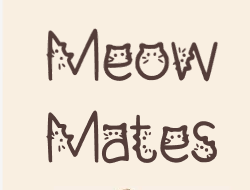


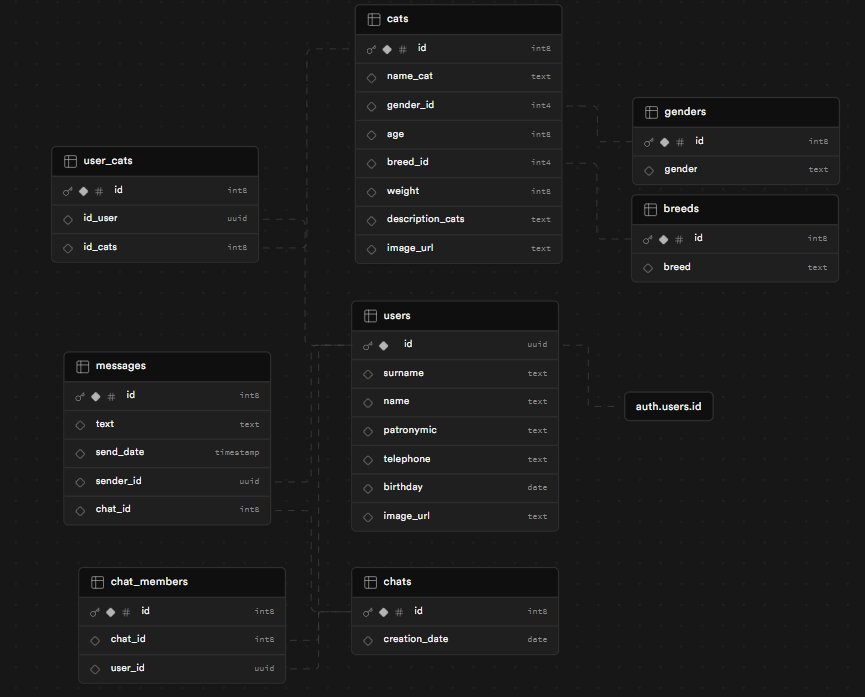
Рисунок 6. Логотип.

## Инструменты Figma.

Если перейти по [ссылке макета](https://www.figma.com/design/w2HXW4Iq7LtwhHI8sro678/Meow-mates?node-id=0-1&t=nz9VY0NWUA284gup-1), откроется главная страница макета с описанием стилей, того, как выглядит приложение в нескольких цветовых темах (светлая, темная). Описание того, какие шрифты необходимо использовать.

# ГЛАВА 3. РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ

Была разработана следующая схема базы данных:



А теперь, о каждой таблице поподробней:

* 1. **Таблица breeds:**

Данная таблица хранит информацию о породе кота (кот/кошка) и имеет следующие поля:

|  |  |  |
| --- | --- | --- |
| PK | Поле | Тип данных |
| PK | id | Int8 |
|  | breed | text |

* 1. **Таблица genders:**   
     Данная таблица хранит информацию о гендере кота (кот/кошка) и имеет следующие поля:

|  |  |  |
| --- | --- | --- |
| PK | Поле | Тип данных |
| PK | id | Int8 |
|  | gender | text |

* 1. **Таблица cats:**

Данная таблица хранит информацию о котах (имя, порода, возраст, вес, описание, картинку, пол кота, код породы) и имеет следующие поля:

|  |  |  |
| --- | --- | --- |
| PK | Поле | Тип данных |
| PK | id | Int8 |
|  | name\_cat | text |
| FK | gender\_id | Int8 |
|  | age | Int8 |
| FK | breed\_id | Int8 |
|  | wheight | Int8 |
|  | description\_cats | text |
|  | Image\_url | text |

Поле gender\_id ссылается на таблицу genders на поле id. Поле breed\_id ссылается на таблицу breeds на поле id.

* 1. **Таблица users:**

Данная таблица хранит информацию о хозяине кота (ФИО, телефон, дату рождения, ссылку на картинку) и имеет следующие поля:

|  |  |  |
| --- | --- | --- |
| PK | Поле | Тип данных |
| PK | id | uuid |
|  | surname | text |
|  | name | text |
|  | patronymic | text |
|  | telephone | text |
|  | birthday | text |
|  | Image\_url | text |

Поле id ссылается на встроенную схему авторизации в supabase (auth), на таблицу users на поле id. Благодаря этому соединению мы можем применять интегрированный механизм аутентификации от Supabase и одновременно сохранять дополнительные сведения о залогиненном пользователе.

* 1. **Таблица user\_cats:**

Данная таблица позволяет связать 2 таблицы связью многое-ко-многим. Она имеет следующие поля:

|  |  |  |
| --- | --- | --- |
| PK | Поле | Тип данных |
| PK | id | Int8 |
| FK | Id\_user | uuid |
| FK | Id\_cats | Int8 |

Поле id\_user ссылается на таблицу users на поле id. Поле id\_cats ссылается на таблицу cats на поле id. Эти отношения представляют собой связь «один ко многим» между таблицами пользователей и котов. У одного владельца может быть несколько питомцев, но один питомец не может принадлежать нескольким владельцам.

* 1. **Таблица chats:**

Данная таблица хранит информацию о чатах (номере и дате создания) имеет следующие поля:

|  |  |  |
| --- | --- | --- |
| PK | Поле | Тип данных |
| PK | id | Int8 |
|  | Creation\_date | date |

* 1. **Таблица chat\_members:**

Данная таблица имеет следующие поля:

|  |  |  |
| --- | --- | --- |
| PK | Поле | Тип данных |
| PK | id | Int8 |
|  | Chat\_id | Int8 |
|  | user\_id | Int8 |

Поле user\_id ссылается на таблицу users на поле id. Поле chat\_id ссылается на таблицу chats на поле id. Эти всвязи создают отношения «многие ко многим» между таблицами пользователей и чатов. В одном чате может быть несколько участников, как минимум двое, а один и тот же человек может быть участником нескольких чатов.

* 1. **Таблица messages:**

Данная таблица хранит информацию о сообщениях (сообщение, дату отправки, кто отправил, в каком чате) и имеет следующие поля:

|  |  |  |
| --- | --- | --- |
| PK | Поле | Тип данных |
| PK | id | uuid |
|  | text | text |
|  | Send\_date | timestamp |
|  | Sender\_id | uuid |
|  | Chat\_id | Int8 |

# ГЛАВА 4. ТЕСТИРОВАНИЕ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ

Документация тестирования серверной части представлена в [Приложении 3](#_Приложение_3).

# ГЛАВА 5. РАЗРАБОТКА МОБИЛЬНОГО ПРИЛОЖЕНИЯ

## Используемые библиотеки

* + - 1. Библиотеки для работы supabase.

|  |
| --- |
| val supabase\_version = "2.6.1" *implementation*(platform("io.github.jan-tennert.supabase:bom:$supabase\_version")) *implementation*("io.github.jan-tennert.supabase:postgrest-kt") *implementation*("io.github.jan-tennert.supabase:gotrue-kt") *implementation*("io.github.jan-tennert.supabase:realtime-kt") |

* + - 1. Библиотека Jetpack Compose для реализации навигации между экранами в вашем приложении.

|  |
| --- |
| val nav\_version = "2.7.7" *implementation*("androidx.navigation:navigation-compose:$nav\_version") |

* + - 1. Набор библиотек Ktor для создания HTTP-клиента. Используются различные модули для обработки контента и сериализации/десериализации JSON с помощью kotlinx.serialization.

|  |
| --- |
| val ktor\_version = "2.3.10" *implementation*("io.ktor:ktor-client-android:$ktor\_version") |

* + - 1. Библиотеки Hilt (DI фреймворк) для внедрения зависимостей.

|  |
| --- |
| val hilt\_android\_version = "2.48" *implementation*("com.google.dagger:hilt-android:$hilt\_android\_version") *kapt*("com.google.dagger:hilt-compiler:$hilt\_android\_version") *implementation*("androidx.hilt:hilt-navigation-compose:1.0.0") |

## Архитектура

Архитектура, выбранная для проекта, — Clean Architecture. Это решение было принято с учётом следующих преимуществ:

1. Разделение ответственности (масштабируемость, минимизация проблем с зависимостями).
2. Тестируемость.
3. Устойчивость к изменениям.

Основные принципы:

1. Принцип единственной ответственности. Каждый компонент (модуль, класс или функция) выполняет только одну задачу. Это позволяет чётко определить ответственность каждого компонента и снизить связанность кода.
2. Разделение на слои. Приложение должно быть разделено на слои, каждый из которых отвечает за свою область. Обычно выделяют следующие уровни: domain, models, views.

* Domain — бизнес-логика, изолированная от деталей реализации.
* Models — хранилище данных.
* Views — отвечает за отображение пользовательского интерфейса и обработку событий.

Архитектура проекта представлена на рисунке 15.

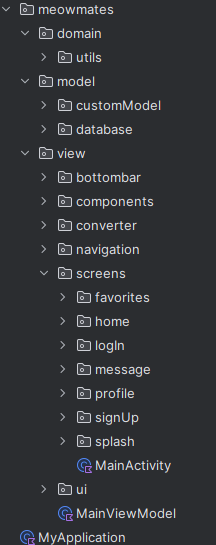


Рисунок 1. Архитектура проекта.

## Тема

После создания приложения на Jetpack Compose создаётся базовая тема Material Theme. Но данную тему не очень удобно использовать, поэтому была создана новая, custom тема MeowMatesTheme.

|  |
| --- |
| object MeowMatesTheme {  val colors: CollorPalete  @ReadOnlyComposable  @Composable  get() = *LocalColors*.current   val fonts: myType  @ReadOnlyComposable  @Composable  get() = *LocalFonts*.current }  @Composable fun MeowMatesTheme(  currentTheme: Int = ThemeMode.Dark,  fonts: myType = MeowMatesTheme.fonts,  content : @Composable () -> Unit ) {   val colors = when (currentTheme){  ThemeMode.Light -> *baseLightPalette* ThemeMode.Dark -> *baseDarkPalette* else -> if(isSystemInDarkTheme()){  *baseDarkPalette* } else {  *baseLightPalette* }  }  CompositionLocalProvider(  *LocalColors* provides colors,  *LocalFonts* provides fonts,  content = content  ) }  data object ThemeMode {  const val System = 0  const val Light = 1  const val Dark = 2 } |

В эту тему можно передать аргументом цветовую режим темы (светлая, тёмная).

Также для удобства был создан класс темы, чтобы задавать свои названия переменным с цветом.

|  |
| --- |
| data class CollorPalete(  val background: Color,  val title:Color,  val text: Color,  val inversionText: Color,  val activIcon:Color,  val passivIcon:Color,  val colorBar: Color,  val container: Color,  val button:Color ) |

На основе этого класса были добавлены две темы: темная и светлая.

|  |
| --- |
| /светлая тема - не перенесены цвета val *ColorBar2* = *Color*(0xFFFFFFFF) val *Background2* = *Color*(0xFFF9EFE0) val *Primary2* = *Color*(0xFFF2DEC0) val *Secondary2* = *Color*(0xFF98684D) val *Tertiary2* = *Color*(0xFF553D39)  val *baseLightPalette* = CollorPalete(  background = *Background2*,  title = *Tertiary2*,  text= *Tertiary2*,  inversionText = *Background2*,  activIcon = *Secondary2*,  passivIcon =*Tertiary2*,  colorBar = *ColorBar2*,  container = *Primary2*,  button = *Secondary2* ) //темная тема val *Primary1* = *Color*(0xFFE0C6B5) val *Secondary1* = *Color*(0xFF4A3D30) val *Background1* = *Color*(0xFF2B2B2B) val *ColorBar1* = *Color*(0xFF000000)  val *baseDarkPalette* = CollorPalete(  background = *Background1*,  title = *Primary1*,  text= *Primary1*,  inversionText = *Background1*,  activIcon = *Primary1*,  passivIcon = *Secondary1*,  colorBar = *ColorBar1*,  container = *Secondary1*,  button = *Primary1* ) val *LocalColors*= *staticCompositionLocalOf*<CollorPalete>**{** *baseDarkPalette* **}** |

Точно таким же способом создан класс для шрифтов, в котором добавлены шрифты и стили для текста

|  |
| --- |
| val *Montseratt* = *FontFamily*(  *Font*(R.font.*montseratt\_bold*, FontWeight.Bold),  *Font*(R.font.*montseratt\_regular*, FontWeight.Normal) ) val *CatFonts* = *FontFamily*(  *Font*(R.font.*cat*) )  data class myType(  val textWatermark: TextStyle,  val title:TextStyle,  val defaultText:TextStyle  )  val *type* = myType(  textWatermark = TextStyle(  fontFamily = *Montseratt*,  fontWeight = FontWeight.Bold,  fontSize = 16.*sp* ),  title = TextStyle(  fontFamily = *CatFonts* ),  defaultText = TextStyle(  fontFamily = *Montseratt*,  fontWeight = FontWeight.Normal,  fontSize = 16.*sp* )  ) val *LocalFonts* = *staticCompositionLocalOf* **{** *type* **}** |

В MainActicity установлена тема

|  |
| --- |
| MeowMatesTheme(  currentTheme = ThemeMode.Dark ) **{**  //Scaffold  **}** |

## Настройка Hilt.

В процессе разработки мобильного приложения, многие классы (например, viewModel) имеют зависимости, которые необходимо передавать при создании объекта. Зависимостей может быть достаточно много и для того, чтобы каждый раз для создания объекта с зависимостями не нужно было прописывать и сами зависимости будет использована библиотека Hilt, которая будет автоматически генерировать нужные зависимости (для этого нужно только прописать модуль для генерации)

Для того, что бы hilt работал корректно, нам необходимо добавить класс MyApplication, который наследуем от Application() и используем аннотацию @HiltAndroidApp. Таким образом, получаем следующий класс:

|  |
| --- |
| @HiltAndroidApp  class MyApplication: Application() { } |

Данный класс необходим для того, что бы подключить необходимые зависимости. Далее, необходимо подключить данный класс в файле AndroidManifest:

|  |
| --- |
| <application  android:name="com.example.MyApplication"  /// </application> |

Также, чтобы Hilt предоставлял зависимости другим классам Android, у MainActivity прописана аннотация @AndroidEntryPoint

|  |
| --- |
| @AndroidEntryPoint class MainActivity : ComponentActivity() {  ///  } |

## Подключение к supabase

В данном проекте используется Supabase — это база данных, бесплатный аналог похожего сервиса FireBase. Данный сервис позволяет начинать разработку приложений, не заботясь о необходимости поддержания серверной инфраструктуры. В Supabase можно работать с базами данных, которая создана на основе PostgreSQL, так же есть встроенная аутентификация, возможность создания так называемых «Корзин» для работы с изображениями, файлами у пользователей. Также, в Supabase есть функция синхронизации в режиме реального времени, с помощью веб-хуков.

Для того, чтобы использовать Supabase – необходимо подключить несколько библиотек, которые были подключены и описаны в пункте 5.1 данного отчета. Для того, чтобы было удобно работать с данными, были созданы следующие модели данных:

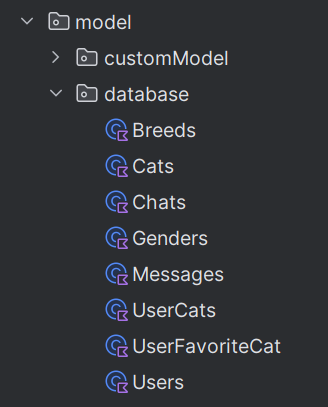


Рисунок 2. Модели таблиц БД

Для каждой таблицы базы данных были созданы аналогичные классы данных в приложении для того, чтобы при обращении к таблицам базы данных Supabase можно было удобно импортировать все в таблицу из базы данных. К примеру, возьмем таблицу с породами котов – в таблице в базе данных, там есть только 2 столбца: id и breed (порода). Аналогично была создана таблица в приложении. Также, добавлена аннотация @Serializable для того, чтобы указать что данный класс будет сериализировать данные.

|  |
| --- |
| @Serializable data class Breeds (  val id: Int =0,  val breed: String = "" ) |

По аналогии были созданы и все последующие классы.

Класс Cats, который хранит в себе информацию о котах:

|  |
| --- |
| @Serializable data class Cats (  var id: Int =0,  var name\_cat: String = "",  var gender\_id: Int =0,  var age: Int=0,  var breed\_id: Int=0,  var weight: Int = 0,  var description\_cats: String ="",  var image\_url: String ="" ) |

Класс Сhats, который хранит в себе информацию о чатах пользователей:

|  |
| --- |
| @Serializable data class Chats (  val id: Int =0,  val first\_user:String,  val second\_user:String ) |

Класс Genders, который хранит в себе информацию о гендере кота/кошки:

|  |
| --- |
| @Serializable data class Genders (  val id: Int =0,  val gender: String ="" ) |

Класс Messages, коорый хранит в себе информацию о сообщениях пользователей. А именно, сами сообщения, отправителя, ну и id чата:

|  |
| --- |
| @Serializable data class Messages (  val id: Int = 0,  val text: String ="",  val sender\_id: String ="",  val chat\_id: Int =0 ) |

Класс UserCats, хранит информацию, где какой кот принадлежит какому хозяину:

|  |
| --- |
| @Serializable data class UserCats (  val id: Int =0,  val id\_user: String="",  val id\_cats: Int=0 ) |

Класс UserFvoriteCat хранит информацию о том, какой кот понравился хозяину:

|  |
| --- |
| @Serializable data class UserFavoriteCat (  val id: Int=0,  val id\_user: String="",  val id\_cat: Int =0 ) |

Класс Users, хранит информацию о пользователях системы:

|  |
| --- |
| @Serializable data class Users (  var id: String = "",  var surname: String ="",  var name: String ="",  var patronymic:String="",  var birthday:String? = "",  var image\_url: String? ="",  var telephone: String? ="" ) |

## Основные экраны приложения

В приложении разработано несколько экранов, которые вы можете посмотреть на рисунке 3 ниже.

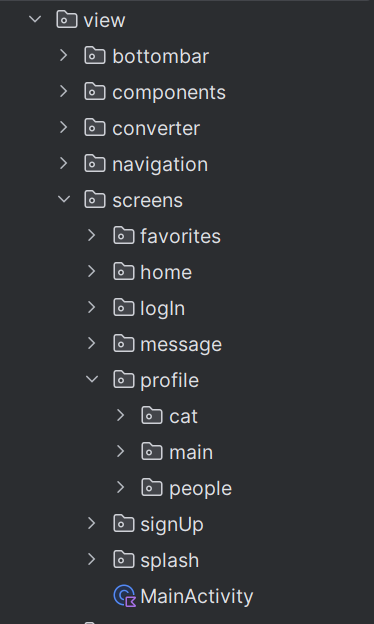


Рисунок 3. Папки приложения

Пожалуй, начнем с самого 1 экрана, который видит пользователь – Splash. Этот экран находится активным несколько секунд, необходим для того, что бы перейти на другой экран – авторизации. Код для того что бы перейти на следующий экран находится в отдельном файле SplashViewModel:

|  |
| --- |
| @HiltViewModel class SplashViewModel @Inject constructor(): ViewModel() {   fun launch(controller: NavHostController){  *viewModelScope*.*launch* **{** delay(2000L)  controller.navigate(NavigationRoutes.LOGIN) **{** popUpTo(NavigationRoutes.SPLASH)**{** inclusive = true;  **}  }  }** } } |



Рисунок 4. Splash-скрин

Следующий экран, это экран авторизации - LogIn. Необходим для того, чтобы зайти в приложение. В данном экране в логике приложения есть 1 функция, которая реализует вход в приложение, то есть переход на следующую страницу.

|  |
| --- |
| fun LogIn(controller: NavHostController, context: Context){  *viewModelScope*.*launch* **{** try{  val user = Constants.supabase.*auth*.signInWith(Email)**{** email = emailUser.value  password = passwordUser.value  **}** *currentUser* = Constants.supabase.*auth*.currentUserOrNull()?.id  Toast.makeText(context, "Авторизация прошла успешно", Toast.*LENGTH\_SHORT*).show()  Log.d("log in","Success")  Log.d("log in", *currentUser*.*toString*())  state = ResultStateSignIn.Success("Success")  controller.navigate(NavigationRoutes.HOME)**{** popUpTo(NavigationRoutes.LOGIN)**{** inclusive = true  **}  }** }  catch(e:Exception){  Log.d("log in",e.message.*toString*())  Toast.makeText(context, "Возникла ошибка входа: ${e.message.*toString*()}", Toast.*LENGTH\_SHORT*).show()  state = ResultStateSignIn.Error(e.message.*toString*())  //\_logInState.value = ResultLogIn.Error(e.message ?: "Unknown error")  }  **}** } |

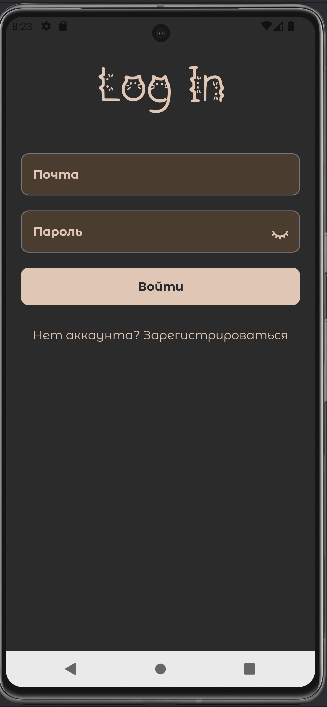


Рисунок 5. Экран авторизации

SignUp – экран для того, что бы пользователь мог зарегистрироваться. В данном экране есть только 1 функция, которая проверяет что бы данные не были пустыми, и выполняет переход от 1 экрана к другому, регистрируя пользователя в базу данных.

|  |
| --- |
| fun signUp(controller: NavHostController, context: Context){  if(surname.value!="" && name.value!=""  && patro.value!="" && telephone.value!=""  && emailUser.value!="" && passwordUser.value!="" && birthdate.value!=""  ) {  *viewModelScope*.*launch* **{** try {  Constants.supabase.*auth*.signUpWith(Email) **{** email = emailUser.value  password = passwordUser.value  **}** val user = Constants.supabase.*auth*.currentUserOrNull()  if(user != null) {  Log.d("sign up", "Был зарегистрирован следующий пользователь: ${user.id}")  Log.d("sign up", "Пользователь: ${Constants.supabase.*auth*.currentUserOrNull()!!.id}")   val newUser = Users(  id = user.id,  surname = surname.value,  name = name.value,  patronymic = patro.value,  birthday = birthdate.value,  image\_url = "",  telephone = telephone.value  )  var result = Constants.supabase.*from*("users").insert(newUser)  *currentUser* = Constants.supabase.*auth*.currentUserOrNull()?.id  Log.d("sign up","Success")  Log.d("sign up", *currentUser*.*toString*())  state = ResultSignUp.Success("Success")  Toast.makeText(context, "Регистрация прошла успешно", Toast.*LENGTH\_SHORT*).show()  controller.navigate(NavigationRoutes.HOME)**{** popUpTo(NavigationRoutes.SIGNUP)**{** inclusive = true  **}  }** }  } catch (e: Exception) {  Log.d("sign up", "ERROR: ${e.message.*toString*()}")  Toast.makeText(context, "Произошла ошибка: ${e.message.*toString*()}", Toast.*LENGTH\_SHORT*).show()  state = ResultSignUp.Error(e.message.*toString*())  }  **}** }  else{  //Toast.makeText(context, "Не все поля заполненны", Toast.LENGTH\_SHORT).show()  Log.d("sign up", "Не все поля заполненны")  Toast.makeText(context, "Не все поля заполненны", Toast.*LENGTH\_SHORT*).show()  }  } |

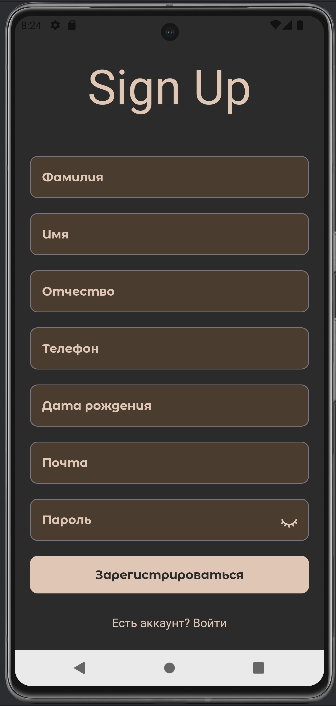


Рисунок 6. Экран регистрации пользователя

Home – главный экран приложения. Где есть список всех котов, в виде карточек, у которым мы можем поставить лайк. На этом экране, после того как мы перешли на данный экран – загружаются данные при помощи функции GetComponent(), которая в свою очередь запускает еще 3 функции: GetFavoriteCat() – которая загружает «лайкнутых котов», GetBreeds() – загружает попроды котов, GetCats() – воссоединяет все данные воедино и записывает все это в итоговый лист котов типа CustomCat – который хранит в себе данные о котах с породой в текстовом формате, и хранит в себе информацию о том, лайкнут ли он или нет текущим пользователем.

|  |
| --- |
| @HiltViewModel class HomeViewModel @Inject constructor():ViewModel() {  var catsList = *mutableStateOf*<List<CustomCats>>(*emptyList*())  var breedsList = *mutableStateOf*<List<Breeds>>(*emptyList*())  var favoriteCatList = *mutableStateOf*<List<UserFavoriteCat>>(*emptyList*())   init{  GetComponent()  }  fun GetComponent(){  GetFavoriteCat()  GetBreeds()  GetCats()  }  private fun GetFavoriteCat(){  *viewModelScope*.*launch* **{** try{  val responce = Constants.supabase.*from*("user\_favorite\_cats").select **{** filter **{** eq("id\_user", *currentUser*.*toString*())  **}  }**.decodeList<UserFavoriteCat>()  favoriteCatList.value = responce  Log.d("HomeVM-GetFavoriteCat", favoriteCatList.value.toString())  }  catch (e:Exception){  Log.d("HomeVM-GetFavoriteCat", "Ошибка загрузки любимых котов: ${e.message}")  }  **}** }  private fun GetBreeds() {  *viewModelScope*.*launch* **{** try {  val response = Constants.supabase.*from*("breeds").select().decodeList<Breeds>()  breedsList.value = response  } catch (e: Exception) {  Log.d("HomeVM", "Ошибка загрузки пород: ${e.message}")  }  **}** }  private fun GetCats(){  *viewModelScope*.*launch* **{** try{  val response = Constants.supabase.*from*("cats").select().decodeList<Cats>()  if(response.isEmpty()){  Log.d("HomeVM-GetCats", "У вас проблемы, котики пустые")  }   // Сопоставляем данные о котах с названиями пород  catsList.value = response.*map* **{** cat **->** val breed = breedsList.value.*find* **{ it**.id == cat.breed\_id **}** val favorite = favoriteCatList.value.*find* **{ it**.id\_cat == cat.id **}** var boolFavCat: Boolean = false  if( favorite?.id\_cat == cat.id) boolFavCat = true   CustomCats(  id = cat.id,  name\_cat = cat.name\_cat,  gender\_id = cat.gender\_id,  age = cat.age,  breed\_id = cat.breed\_id,  weight = cat.weight,  description\_cats = cat.description\_cats,  image\_url = cat.image\_url,  breed\_name = breed?.breed ?: "Неизвестная порода", // Название породы  favorite = boolFavCat  )  **}** }  catch (e: Exception){  Log.d("HomeView-GetCats", e.message.*toString*())  }  **}** } } |



Рисунок 7. Главный экран приложения

Favorites – экран «избранного», на котором отображаются коты с отметкой сердечка. На данном экране практически все то же самое, что и на главном экране, только на отрисовку отправляются те коты, которые понравились текущему пользователю.

|  |
| --- |
| @HiltViewModel class FavoritesViewModel @Inject constructor(): ViewModel() {  var catsList = *mutableStateOf*<List<CustomCats>>(*emptyList*())  var breedsList = *mutableStateOf*<List<Breeds>>(*emptyList*())  var favoriteCatList = *mutableStateOf*<List<UserFavoriteCat>>(*emptyList*())  init{  GetComponent()  }  fun GetComponent(){  GetFavoriteCat()  GetBreeds()  GetCats()  }  private fun GetFavoriteCat(){  *viewModelScope*.*launch* **{** try{  val responce = Constants.supabase.*from*("user\_favorite\_cats").select **{** filter **{** eq("id\_user", *currentUser*.*toString*())  **}  }**.decodeList<UserFavoriteCat>()  favoriteCatList.value = responce  Log.d("FavoritesVM-GetFavoriteCat", favoriteCatList.value.toString())  }  catch (e:Exception){  Log.d("FavoritesVM-GetFavoriteCat", "Ошибка загрузки любимых котов: ${e.message}")  }  **}** }  private fun GetBreeds() {  *viewModelScope*.*launch* **{** try {  val response = Constants.supabase.*from*("breeds").select().decodeList<Breeds>()  breedsList.value = response  } catch (e: Exception) {  Log.d("FavoritesVM", "Ошибка загрузки пород: ${e.message}")  }  **}** }  private fun GetCats(){  *viewModelScope*.*launch* **{** try{  val cats = Constants.supabase.*from*("cats").select().decodeList<Cats>()  catsList.value = favoriteCatList.value.*map***{** items**->** val cat = cats.*find* **{ it**.id == items.id\_cat **}** val breed = breedsList.value.*find* **{ it**.id == cat?.breed\_id **}** CustomCats(  id = cat!!.id,  name\_cat = cat.name\_cat,  gender\_id = cat.gender\_id,  age = cat.age,  breed\_id = cat.breed\_id,  weight = cat.weight,  description\_cats = cat.description\_cats,  image\_url = cat.image\_url,  breed\_name = breed?.breed ?: "Неизвестная порода", // Название породы  favorite = true  )  **}** }  catch (e: Exception){  Log.d("FavoritesVM-GetCats", e.message.*toString*())  }  **}** }  } |

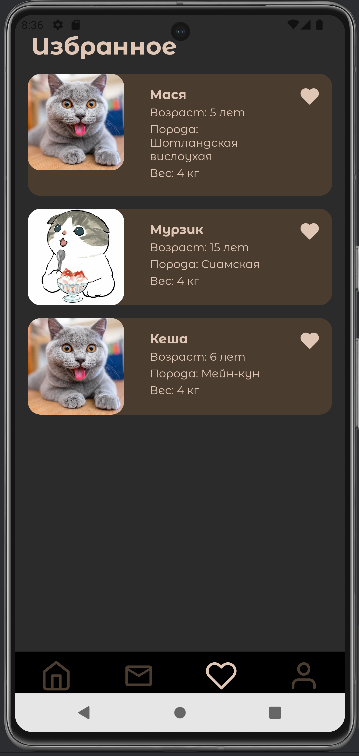


Рисунок 8. Экран избранное

Profile – данный экран подразделяется на несколько:

* Main – основной экран пользователя, который открывается прямиком из нижнего меню навигации. Здесь есть несколько основных кнопок, таких как: «Изменить профиль» - она переводит на другой экран для изменения профиля, «Выйти из профиля» - направляет на экран авторизации, «+» под надписью «Мои коты:» - направляет на страницу добавления кота.

|  |
| --- |
| class MainProfileViewModel @Inject constructor():ViewModel() {  var User = *mutableStateOf*<Users?>(null)  var catList = *mutableStateOf*<List<CustomCats>>(*emptyList*())   fun navigateToMyProfile(controller:NavHostController){  controller.navigate(NavigationRoutes.MYPROFILE) **{** popUpTo(NavigationRoutes.MAINPROFILE)**{** inclusive = true;  **}  }** }  fun navigateToLogIn(controller:NavHostController){  controller.navigate(NavigationRoutes.LOGIN) **{** popUpTo(NavigationRoutes.MAINPROFILE)**{** inclusive = true;  **}  }** }  fun GetData(){  *viewModelScope*.*launch* **{** try{  val fetchUser = Constants.supabase.*from*("users").select **{** filter **{** eq("id", *currentUser*.*toString*())  **}  }**.decodeSingle<Users>()  User.value = fetchUser  Log.d("ProfileVM", "Загружен пользователь: "+ User.value.*toString*())  } catch (e: Exception) {  Log.d("ProfileVM", "Ошибка загрузки пользователя: ${e.message}")  }  try{  val idCats = Constants.supabase.*from*("user\_cats").select **{** filter **{** eq("id\_user", *currentUser*.*toString*())  **}  }**.decodeList<UserCats>()  val result = idCats.*map* **{** it ->  val cat = Constants.supabase.*from*("cats").select **{**filter **{**eq("id", it.id\_cats)**}}**.decodeSingle<Cats>()  val breedCat = Constants.supabase.*from*("breeds").select **{** filter **{** eq("id", cat.breed\_id) **} }**.decodeSingle<Breeds>()  CustomCats(  id = cat.id,  name\_cat = cat.name\_cat,  gender\_id = cat.gender\_id,  age = cat.age,  breed\_id = cat.breed\_id,  weight = cat.weight,  description\_cats = cat.description\_cats,  image\_url = cat.image\_url,  breed\_name = breedCat.breed,  )  **}** catList.value = result  Log.d("ProfileVM", "Коты пользователя: ${catList.value}")  }catch (e: Exception) {  Log.d("ProfileVM", "Ошибка загрузки котов пользователя: ${e.message}")  }  **}** }  } |

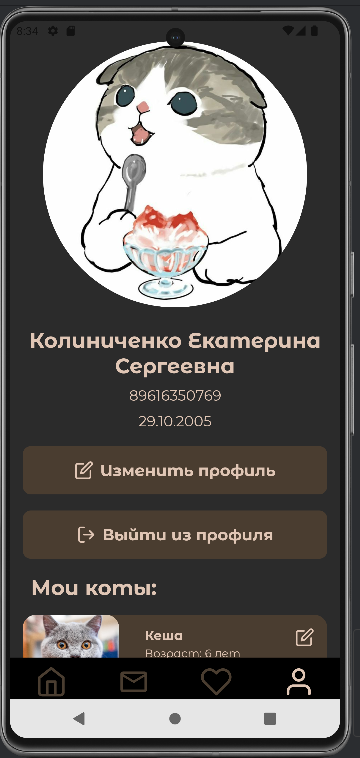


Рисунок 9. Основной профиль пользователя.

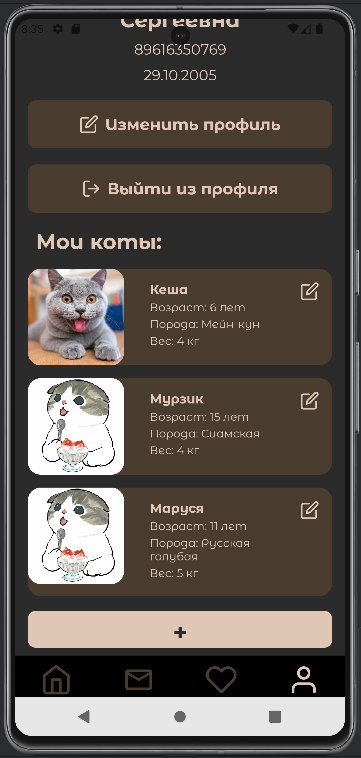


Рисунок 10. Главный профиль, отображение котов пользователя.

* Cat – профиль кота, когда мы кликаем: создать кота, изменить профиль кота открывается именно этот экран с заполненными данными или нет, в зависимости от того, добавляем мы кота или изменяем.

|  |
| --- |
| @HiltViewModel class CatProfileViewModel @Inject constructor(): ViewModel() {  var currentIdCat = *mutableStateOf*(0)  val name = *mutableStateOf*("")  val gender = *mutableStateOf*("")  val age = *mutableStateOf*("")  val breed = *mutableStateOf*("")  val weight = *mutableStateOf*("")  val descriptor = *mutableStateOf*("")  val image\_url = *mutableStateOf*("")    fun GetCat() {  *viewModelScope*.*launch* **{** if (currentIdCat.value != 0) {  var cat = Constants.supabase.*from*("cats")  .select **{** filter **{** eq("id", currentIdCat.value) **} }**.decodeSingle<Cats>()  Log.d("CatProfileVM", "Получили кота, юхуууу")  name.value = cat.name\_cat  gender.value = cat.gender\_id.toString()  age.value = cat.age.toString()  breed.value = cat.breed\_id.toString()  weight.value = cat.weight.toString()  descriptor.value = cat.description\_cats  image\_url.value = cat.image\_url  } else {  Log.d("CatProfileVM", "Начинаем создавать кота")  }  **}** }  fun SaveChangeCat(controller: NavHostController) {  *viewModelScope*.*launch* **{** if (currentIdCat.value!= 0) { //изменяем кота  GetCat()  try {  //изменение кота:  val updateCat = Cats(  name\_cat = name.value,  gender\_id = gender.value.*toInt*(),  age = age.value.*toInt*(),  breed\_id = breed.value.*toInt*(),  weight = weight.value.*toInt*(),  description\_cats = descriptor.value,  image\_url = image\_url.value  )  val responce = Constants.supabase.*from*("cats")  .update(updateCat) **{** filter **{** eq("id", currentIdCat.value) **} }** Log.d("CatProfileVM", "Изменяем кота: ${name.value}")  Log.d("CatProfileVM", "ID кота: ${currentIdCat.value}")  } catch (e: Exception) {  Log.d(  "CatProfileVM",  "Не удалось сохранить изменения: ${e.message.*toString*()}"  )  }   } else { //добавляем  try {  val cat = Cats(  name\_cat = name.value,  gender\_id = gender.value.*toInt*(),  age = age.value.*toInt*(),  breed\_id = breed.value.*toInt*(),  weight = weight.value.*toInt*(),  description\_cats = descriptor.value,  image\_url = ""  )  //добавляем и получаем объект добавленного кота  val catDb = Constants.supabase.*from*("cats").insert(cat) **{** select() **}** .decodeSingle<Cats>()   Log.d("CatProfileVM", "Добавляем кота: ${catDb.id}. ${catDb.name\_cat}")  //создаем объект связи нового кота и пользователя  val user\_cat = UserCats(  id\_user = *currentUser*.*toString*(),  id\_cats = catDb.id  )  //добавляем этот объект  val ansv = Constants.supabase.*from*("user\_cats").insert(user\_cat)  } catch (e: Exception) {  Log.d(  "CatProfileVM",  "Не удалось добавить изменения: ${e.message.*toString*()}"  )  }   }  controller.navigate(NavigationRoutes.MAINPROFILE) **{** popUpTo(NavigationRoutes.CATPROFILE) **{** inclusive = true  **}  }  }** } } |



Рисунок 11.Экран создания кота.

* People – профиль самого пользователя, он открывается когда мы кликаем из основного профиля (main) – изменить профиль. Открывается полностью заполненный профиль пользователя из базы данных.

|  |
| --- |
| @HiltViewModel class MyProfileViewModel @Inject constructor(): ViewModel() {  val surname= *mutableStateOf*("")  val name = *mutableStateOf*("")  val patro= *mutableStateOf*("")  val telephone = *mutableStateOf*("")  val birthdate = *mutableStateOf*("")  val image\_url = *mutableStateOf*("")   private val \_user = *MutableStateFlow*<Users?>(null)  val user: StateFlow<Users?> = \_user.*asStateFlow*()  fun GetUser(){  try{  *viewModelScope*.*launch* **{** val user = Constants.supabase.*from*("users").select**{** filter **{** eq("id",*currentUser*.*toString*()) **} }**.decodeSingle<Users>()  surname.value = user.surname  name.value = user.name  patro.value = user.patronymic  telephone.value = user.telephone ?: ""  birthdate.value = user.birthday ?: ""  image\_url.value = user.image\_url ?: ""  **}** Log.d("MyProfileVM", "Пользовательская картинка: "+ image\_url )  }  catch (e:Exception){  Log.d("MyProfileVM", "Возникла ошибка загрузки пользователя: "+ e.message.*toString*())  }  }  fun saveChanges(controller: NavHostController) {  *viewModelScope*.*launch* **{** try {  val user = Constants.supabase.*from*("users").select**{** filter **{** eq("id",*currentUser*.*toString*()) **} }**.decodeSingle<Users>()  val userToUpdate = Users(  id = user.id,  surname = surname.value,  name = name.value,  patronymic = patro.value,  birthday = birthdate.value,  image\_url = user.image\_url,  telephone = telephone.value  )   val result = Constants.supabase.*from*("users").update(userToUpdate)**{**filter **{** eq("id", *currentUser*.*toString*()) **}}** Log.e("MyProfileVM", "Update data success")  controller.navigate(NavigationRoutes.MAINPROFILE)**{** popUpTo(NavigationRoutes.MYPROFILE)**{** inclusive = true  **}  }** } catch (e: Exception) {  Log.e("MyProfileVM", "Exception saving user: ${e.message}")  // Consider showing an error message to the user  }  **}** }  } |

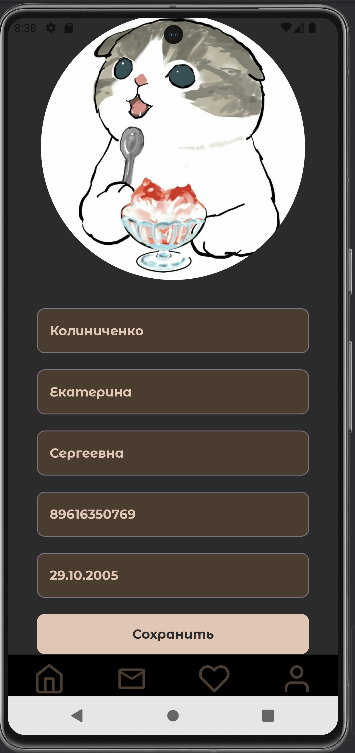


Рисунок 12. Изменение профиля пользователя

Message – не реализован. Пыталась реализовать через Real time. Не получилось, вроде бы и библиотеку подключила, вроде бы и пишу по документации – но ничего не выходит. Точнее, выходит много ошибок. В настоящем приложении там стоит заглушка.

# ГЛАВА 6. ТЕСТИРОВАНИЕ МОБИЛЬНОГО ПРИЛОЖЕНИЯ

## Тест план

### **Цели тестирования.**

Цель: убедиться, что все функции приложения работают в соотвествии с требованиями и спецификациями. Оценить интуитивность интерфейса и удобство навигации для пользователей.

### **Область тестирования**

Тестирование охватывает следующие аспекты:

Функциональные возможности: Все основные функции приложения, включая добавление, редактирование и удаление информации о котах и их владельцах.

Пользовательский интерфейс: Элементы интерфейса, такие как кнопки, формы ввода, списки и навигационные элементы.

### **Стратегия тестирования**

Подход к тестированию: Автоматизированное. Создание Unit-тестов для автоматизации проверки функциональности авторизации и регистрации. Также, создание Ui- тестов для тестирования пользовательского интерфейса – то, что кнопки нажимаются, на экране есть определенный текст.

### **Типы тестов**

* Функциональное тестирование: Проверка работы всех функций приложения (регистрация, добавление котов, поиск партнеров и т.д.).
* Интеграционное тестирование: Проверка взаимодействия между различными компонентами приложения (например, связь между пользователями и их котами).

### **Ресурсы.**

* Android Studio - Инструменты разработки
* Kotlin – язык програмирования, который будет использован для разработки автоматизированных тестов (ui и unit тестов)
* Эмуляторы Android – устройство для тестирования.

### **Критерии завершения**

Тестирование будет завершено, когда:

* все тесты будут успешно пройдены;
* будет подготовлена отчетность о результатах тестирования.

## Тестовый набор

Разработать следующие Unit-тесты:

* + - 1. Проверка состояния загрузки после перехода на экран авторизации;
      2. Проверка состояния успешной авторизации в приложение;
      3. Проверка состояния ошибки авторизации в приложении;
      4. Проверка состояния загрузки при переходе на экран регистрации;
      5. Проверка состояния успешной регистрации в приложении после регистрации.

Разработать следующие Ui-тесты:

1. То, что при запуске приложения определяется контекст приложения, то есть приложение корректно запусскается.
2. То, что при запуске экрана регистрации – на экране отображается надпись «sin up» и то, что есть кнопка «Зарегестрироваться»
3. При запуске экрана регистрации – на экране есть активная кнопка «Зарегестрироваться»
4. После успешной авторизации есть переход на главную страницу приложения с надписью «Главная».
5. После перехода с главной страницы приложения корректно открывается страница «Избранное»

## Тест кейс

Для проверки что все функции разработанного приложения работают в соответствии с требованиями и спецификациями были разработаны Unit и Ui тесты на языке программирования Kotlin в среде разработки Android Studio. Все созданные тесты пройдены успешно.

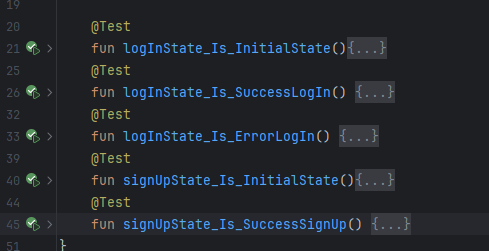


Рисунок 13. Успешно пройденные Unit- тесты

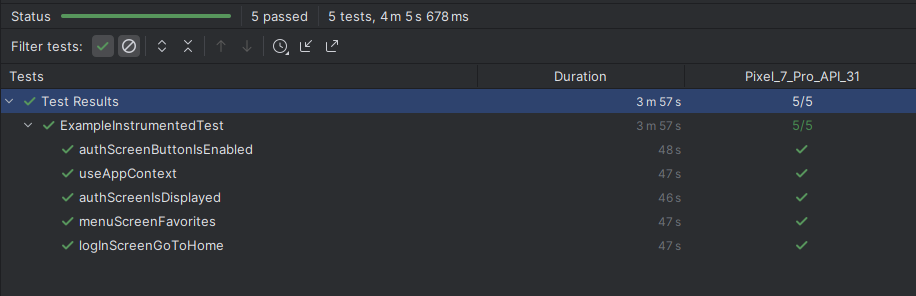


Рисунок 14. Успешно пройденные Ui-тесты

Таблица 2. Тест-кейсы для unit-тестов мобильного приложения MeowMates.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Название** | **Ожидаемый результат** | **Прил.** | **Пройде** |
| 1 | logInState\_Is\_InitialState | Состояние загрузки экрана после его открытия | [Код теста 1](#_Unit_тест_1) | + |
| 2 | logInState\_Is\_SuccessLogIn | Состояние успешной авторизации в приложение | [Код теста 2](#_Unit_тест_2) | + |
| 3 | logInState\_Is\_ErrorLogIn | Состояние ошибки авторизации в приложении | [Код теста 3](#_Unit_тест_3) | + |
| 4 | signUpState\_Is\_InitialState | Состояние загрузки при переходе на экран регистрации | [Код теста 4](#_Unit_тест_4) | + |
| 5 | signUpState\_Is\_SuccessSignUp | Состояние загрузки при переходе на экран регистрации | [Код теста 5](#_Unit_тест_5) | + |

Таблица 3. Тест-кейсы для ui-тестов мобильного приложения MeowMates.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Название** | **Ожидаемый результат** | **Прил.** | **Пройде** |
| 1 | useAppContext | При запуске приложения корректно задается контекст приложения. | [Код теста 1](#_Ui_тест_1) | + |
| 2 | regScreenIsDisplayed | Наличие на экране регистрации надписи «Sign Up» и кнопки «Зарегестирироваться» | [Код теста 2](#_Ui_тест_2) | + |
| 3 | regScreenButtonIsEnabled | При запуске приложения кнопка зарегистрироваться активна. | [Код теста 3](#_Ui_тест_3) | + |
| 4 | logInScreenGoToHome | При вводе корректных данных на странице авторизации происходит переход на страницу Главная. | [Код теста 4](#_Ui_тест_4) | + |
| 5 | menuScreenFavorites | При переходе на страницу «Избранное» корректно отображает информацию. | [Код теста 5](#_Ui_тест_5) | + |

# ПРИЛОЖЕНИЕ А

Документация тестирования серверной части приложения

# 1**. Тест план.**

## Введение.

Данный тест-план описывает стратегию тестирования базы данных приложения Meow Mates, которая управляет информацией о котах, их владельцах и взаимодействиями между пользователями через чаты. Тестирование будет проводиться с использованием Supabase как платформы для управления базой данных.

## Цели тестирования.

Цель: Обеспечить корректность, производимость и безопасность базы данных.

1. проверка корректности работы CRUD операций (создание, чтение, обновление, удаление) для всех таблиц.
2. убедиться в правильности работы внешних ключей и связей между таблицами;
3. проверка правильности применения политик безопасности (RLS);
4. проверка целостности данных.

## Область тестирования

В базе данных есть следующие таблицы:

* breeds: информация о породах котов.
* genders: информация о гендерах котов.
* cats: информация о котах (имя, порода, возраст и т.д.).
* users: информация о владельцах котов (ФИО, телефон и т.д.).
* user\_cats: связь между пользователями и их котами.
* chats: информация о чатах.
* chat\_members: участники чатов.
* messages: сообщения в чатах

## Стратегия тестирования

Подход к тестированию: Автоматизированное. Создание Unit-тестов для автоматизации проверки функциональности базы данных.

Уровень тестирования: Модульное. Тестирование работы базы данных, без ее взаимодействия с бизнес-логикой будущего мобильного приложения.

## Типы тестов

* Функциональное тестирование: Проверка работы всех функций приложения (регистрация, добавление котов, поиск партнеров и т.д.).
* Интеграционное тестирование: Проверка взаимодействия между различными компонентами приложения (например, связь между пользователями и их котами).

## Ресурсы.

* MSTest- Необходимо для создания Unit-тестов;
* С#: Язык программирования для написания Unit-тестов
* Microsoft Visual Studio – IDE, где будут писаться тесты
* Supabase - Облачное СУБД для хранения данных мобильного приложения, объект тестирования

## Критерии завершения

Тестирование будет завершено, когда:

* все тесты будут успешно пройдены;
* будет подготовлена отчетность о результатах тестирования.

# Тестовый набор

* + - 1. Создание нового пользователя – добавляем нового пользователя в схему Auth в базе данных в Supabase. Функция должна проверять, что созданный пользователь не равен null.
      2. Выборка пользователей - получение листа всех пользователей. Функция должна возвращать лист.
      3. Выборка котов - получение листа всех котов. Функция должна возвращать лист.
      4. Выборка чатов - получение листа всех чатов. Функция должна возвращать лист.
      5. Выборка сообщений пользователей - получение листа всех сообщений пользователей. Функция должна возвращать лист.
      6. Выборка пород котов - получение листа всех пород котов. Функция должна возвращать лист.

# Тест кейс

Для проверки работоспособности серверной части с использованием языка программирования C# и фреймворка MSTest был разработан проект. С целью проведения тестирования были разработаны классы, которые описывают таблицы из базы данных в Supabase. Также были созданы методы для проверки самих данных из базы данных. Все тесты пройдены успешно.

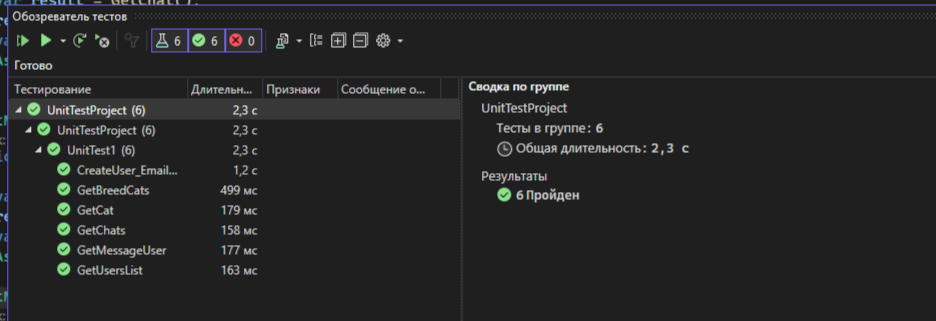


Рисунок 13. Успешно пройденные Unit-тесты

Таблица 1. Тест-кейсы для unit-тестов системы авторизации.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№** | **Название** | **Описание** | **Ожидаемый результат** | **Прил.** | **Пройде** |
| 1 | CreateUser\_EmailPassword | Создание нового пользователя | Assert.IsNotNull(user); | [Код теста 1](#_Код_теста_1.) | + |
| 2 | GetUsersList | Выборка пользователей | Assert.IsNotNull(users); | [Код теста 2](#_Код_теста_2.) | + |
| 3 | GetCat | Выборка котов | Assert.IsNotNull(cats); | [Код теста 3](#_Код_теста_3.) | + |
| 4 | GetChats | Выборка чатов | Assert.IsNotNull(chats); | [Код теста 4](#_Код_теста_4.) | + |
| 5 | GetMessageUser | Выборка сообщений пользователей | Assert.IsNotNull(message); | [Код теста 5](#_Код_теста_5.) | + |
| 6 | GetBreedCats | Выборка пород котов | Assert.IsNotNull(breed); | [Код теста 6](#_Код_теста_6.) | + |

## Код теста 1. Создание нового пользователя

|  |
| --- |
| [TestMethod]  public void CreateUser\_EmailPassword()  {  string email = "kolinichenko.05@mail.ru";  string password = "123456";  var session = CreateUser(email, password);  session.Wait();  var user = session.Result.User;  Assert.IsNotNull(user);  } |

## Код теста 2. Выборка польхователей

|  |
| --- |
| [TestMethod]  public void GetUsersList()  {  var result = GetUsers();  result.Wait();  var users = result.Result.Models;  Assert.IsNotNull(users);  } |

## Код теста 3. Выборка котов

|  |
| --- |
| [TestMethod]  public void GetCat()  {  var result = GetCats();  result.Wait();  var cats = result.Result.Models;  Assert.IsNotNull(cats);  } |

## Код теста 4. Выборка чатов

|  |
| --- |
| [TestMethod]  public void GetChats()  {  var result = GetChat();  result.Wait();  var chats = result.Result.Models;  Assert.IsNotNull(chats);  } |

## Код теста 5. Выборка сообщений пользователей

|  |
| --- |
| [TestMethod]  public void GetMessageUser()  {  var result = GetMessage();  result.Wait();  var message = result.Result.Models;  Assert.IsNotNull(message);  } |

## Код теста 6. Выборка пород котов

|  |
| --- |
| [TestMethod]  public void GetBreedCats()  {  var result = GetBreed();  result.Wait();  var breed = result.Result.Models;  Assert.IsNotNull(breed);  } |

# ПРИЛОЖЕНИЕ Б

## Unit тест 1

|  |
| --- |
| @Test fun logInState\_Is\_InitialState(){  val viewModel = LogInViewModel()  assertEquals(viewModel.state, ResultStateSignIn.Loading) } |

## Unit тест 2

|  |
| --- |
| fun logInState\_Is\_SuccessLogIn() {  val viewModel = LogInViewModel()  viewModel.state = ResultStateSignIn.Success("Success")  // Проверяем, что состояние logInState было правильно установлено  assertEquals(viewModel.state, ResultStateSignIn.Success("Success")) } |

## Unit тест 3

|  |
| --- |
| @Test fun logInState\_Is\_ErrorLogIn() {  val viewModel = LogInViewModel()  viewModel.state = ResultStateSignIn.Error("Invalid login credentials")  // Проверяем, что состояние logInState было правильно установлено  assertEquals(viewModel.state, ResultStateSignIn.Error("Invalid login credentials")) } |

## Unit тест 4

|  |
| --- |
| @Test fun signUpState\_Is\_InitialState(){  val viewModel = SignUpViewModel()  assertEquals(viewModel.state, ResultSignUp.Loading) } |

## Unit тест 5

|  |
| --- |
| @Test fun signUpState\_Is\_SuccessSignUp() {  val viewModel = SignUpViewModel()  viewModel.state = ResultSignUp.Success("Success")  // Проверяем, что состояние logInState было правильно установлено  assertEquals(viewModel.state, ResultSignUp.Success("Success")) } |

## Ui тест 1

|  |
| --- |
| @Test fun useAppContext() {  // Context of the app under test.  val appContext = InstrumentationRegistry.getInstrumentation().*targetContext* assertEquals("com.example.meowmates", appContext.*packageName*) } |

## Ui тест 2

|  |
| --- |
| @Test fun regScreenIsDisplayed() {  scenario.onActivity **{  it**.*setContent* **{** signUpViewModel = SignUpViewModel()  SignUp(controller, signUpViewModel)  **}  }** composeTestRule.*onNodeWithText*("Sign Up").*isDisplayed*()  composeTestRule.*onNodeWithText*("Зарегистрироваться").*assertIsDisplayed*() } |

## Ui тест 3

|  |
| --- |
| @Test fun regScreenButtonIsEnabled() {  scenario.onActivity **{  it**.*setContent* **{** signUpViewModel = SignUpViewModel()  TestNavigation(NavigationRoutes.SIGNUP)  **}  }** composeTestRule.*onNodeWithText*("Зарегистрироваться").*assertIsEnabled*() } |

## Ui тест 4

|  |
| --- |
| @Test fun logInScreenGoToHome() {  scenario.onActivity **{  it**.*setContent* **{** homeViewModel = HomeViewModel()  logInViewModel = LogInViewModel()  logInViewModel.emailUser.value = "user12@mail.ru"  logInViewModel.passwordUser.value = "user12"  logInViewModel.LogIn(controller,context)  TestNavigation(NavigationRoutes.HOME)  **}  }** //Произошел переход на другую страницу  composeTestRule.*onNodeWithText*("Главная").*isDisplayed*() } |

## Ui тест 5

|  |
| --- |
| @Test fun menuScreenFavorites() {  scenario.onActivity **{  it**.*setContent* **{** favoritesViewModel = FavoritesViewModel()  homeViewModel = HomeViewModel()  *currentUser* = "95c28f9a-d823-4219-a7ba-ba1e09fed9ee"  TestNavigation(NavigationRoutes.FAVORITES)  **}  }** composeTestRule.*onNodeWithText*("Избранное").*isDisplayed*() } |