

**Министерство науки и высшего образования РФ**  
**Федеральное государственное автономное образовательное учреждение**  
**высшего образования**  
**«Казанский (Приволжский) Федеральный Университет»**

**ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И**  
**ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ**

**КАФЕДРА ТЕОРЕТИЧЕСКОЙ КИБЕРНЕТИКИ**  
Направление: 01.03.02. Прикладная математика и информатика  
Профиль: Прикладная математика и информатика

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**  
**Разработка и реализация безопасного мобильного приложения**  
**для конфиденциального обмена данными с использованием**  
**шифрования.**

**Работа завершена:**

Студент гр.09-011 ИВМиИТ

«\_\_» \_\_\_\_\_ 20\_\_ г. \_\_\_\_\_ Д.В.Семёнов

**Работа допущена к защите:**

Научный руководитель,  
ассистент, б.с., КТК ИВМиИТ

«\_\_» \_\_\_\_\_ 20\_\_ г. \_\_\_\_\_ Д.К.Баширова

Заведующий кафедрой,  
д.ф.м.н., профессор

«\_\_» \_\_\_\_\_ 20\_\_ г. \_\_\_\_\_ Ф.М.Аблаев

**Казань – 2024**

## ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ .....	3
1. Исследование предметной области.....	5
1.1 Криптография .....	5
1.2 Криптоанализ .....	9
1.3 Алгоритмы шифрования .....	10
1.4 Потокосые шифры.....	11
1.5 Мобильные приложения .....	12
1.6 Swift, SwiftUI.....	13
1.7 Шаблоны проектирования приложений для устройств.....	13
1.8 Способы тестирования .....	15
1.9 UML диаграммы .....	15
2. Разработка алгоритма шифрования .....	17
2.1 Описание шифра.....	17
2.2 Реализация.....	20
2.3 Тестирование .....	22
2.4 Анализ.....	25
3. Разработка сервиса для обмена данными между двумя мобильными устройствами.....	27
3.1 Описание мобильного приложения .....	28
3.2 Firebase.....	28
3.3 UML схема .....	29
3.4 Разработка .....	33
3.5 Результат .....	47
3.6 Тестирование .....	48
ЗАКЛЮЧЕНИЕ.....	53
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ.....	60
ПРИЛОЖЕНИЕ .....	61

## **ВВЕДЕНИЕ**

Криптография, зародившаяся еще в древности, к концу 20-го века превратилась в сложную и мощную науку, неотъемлемую часть современной жизни. С тех пор она стала ключевым элементом защиты информации, обеспечивая безопасность личных данных, финансовых транзакций и государственной переписки. Сейчас криптография проникла практически во все сферы нашей жизни, от банковских операций до общения в социальных сетях. Первые криптографические системы были сложными и требовали специального оборудования и знаний для их использования. Сегодня же современные технологии позволили упростить использование криптографии, сделав ее доступной каждому.

С развитием интернета и мобильных технологий потребность в защите данных возросла многократно. Личные сообщения, финансовые данные и конфиденциальная информация стали уязвимы перед угрозами кибератак. В 21-м веке криптография переживает новый этап развития, и благодаря современным шифровальным методам мы можем обеспечить безопасность данных всего за несколько кликов.

Раньше защита информации была доступна только специализированным службам и крупным компаниям. Теперь же, благодаря развитию технологий, каждый человек может воспользоваться криптографическими методами для защиты своих данных. Появились мобильные приложения и веб-сервисы, которые позволяют легко и надежно шифровать, и расшифровывать информацию, обмениваясь ею в защищенном виде.

Однако, многие существующие решения нацелены на широкий круг пользователей и часто оказываются слишком сложными или перегруженными функционалом. Это может привести к снижению уровня безопасности и неудобству в использовании. Шифрование данных пользователей посредством серверов компаний – является не лучшим решением для пользователей, которые заботятся о своей информации. Надёжность компании определяется её закрытой системой. Поэтому сейчас набирают популярность

сервисы, которые предоставляют пользователям лишь хранилище для данных, а пользователь сам шифрует и расшифровывает свои данные. В данном случае сервисы выступают лишь в качестве посредников.

Целью выпускной квалификационной работы является разработка мобильного приложения для конфиденциального обмена данными, ориентированного на узкую аудиторию пользователей, которые нуждаются в простом и эффективном инструменте для защиты своей информации.

Значимость этой работы заключается в том, что сервисов, которые обеспечивают надлежащую безопасность не так много. Уникальность этой работы заключается в том, что мобильное приложение будет обеспечивать безопасное общение пользователь посредством уникального потокового шифра. Причина, по которой выбрана идея создания уникального потокового шифра заключается в том, чтобы лучше разобраться в теме криптографии и защиты информации.

Для достижения цели были поставлены следующие задачи: разработка шифра, проведение исследований по криптостойкости шифра, разработка сервиса для обмена данными между двумя мобильными устройствами, внедрение шифрования в разработанный сервис, тестирование и анализ результатов работы сервиса;

## **1. Исследование предметной области**

Для того, чтобы привольно создать и спроектировать рабочий алгоритм шифрования, нужны передовые знания в области криптографии, криптоанализа. Только путём тщательного изучения и анализа можно приступить в разработке алгоритма шифрования, в противном случае, ошибки в реализации и обеспечения безопасного кода не избежать.

Для создания современного мобильно приложения, необходимо изучить передовые технологии в проектировании программ, способы взаимодействия с базой данных и способы тестирования.

Для описания всей системы как в общем, так и, в частности, необходимо знать методы, схематичного представления. UML диаграммы являются прекрасным инструментом, которые дают полное понимание работы приложения как специалисту, так и обычному пользователю.

### **1.1 Криптография**

Слово криптография происходит от греческого “тайнопись”. У криптографии долгая и яркая история, насчитывающая несколько тысяч лет. [1] На протяжении тысячелетий криптография защищала информацию от нежелательных лиц, которые старались использовать полученные данные против отправителя.

Криптография — это наука и искусство обеспечения конфиденциальности, целостности и аутентичности информации. Она занимается разработкой методов защиты данных посредством их преобразования в формат, который не может быть прочитан без специального ключа. Исторически криптография начала свое развитие с простейших шифров, таких как шифр Цезаря, и эволюционировала до сложных современных алгоритмов, использующих передовые математические методы.

Шифр, в терминологии компьютеров, представляет собой процесс побитового или посимвольного преобразования данных, не зависящего от лингвистической структуры сообщения. Современная криптография делится на два основных типа шифрования. Симметричное шифрование: один и тот же

ключ используется для зашифровки и расшифровки данных. Асимметричное шифрование: используется пара ключей — открытый для шифрования и закрытый для расшифровки. Данный метод используется в протоколе Диффи-Хеллмана, который будет рассмотрен ниже.

Основная особенность симметричного шифра, что расшифровка и зашифровка сообщения происходит одним ключом.

Сообщения, которые нужно зашифровать, называются открытым текстом. Они преобразуются с помощью функции, параметром которой является ключ. Результат этого преобразования называется зашифрованным текстом, который обычно передаётся через посредством связи. Действия, направленные на взлом шифров, называется криптоанализом, а создание шифров — криптографией. Вместе эти два направления образуют криптологию. На рисунке 1.1.1 можно наблюдать шифрование и дешифрование в обычном понимании. Но в криптографии часто используют ключи, которые являются необходимым элементом каждого шифра. Можно представить аналогию с замком, когда есть ключ от замка, мы можем открыть замок.



Рисунок 1.1.1 – Шифрование и дешифрование

Для обозначения открытого/зашифрованного текста и ключей всегда использую специальную терминологию, которую криптографы часто применяют для описания. Использую формулу  $C = E_k(P)$ , обозначающую, что при зашифровке открытого текста  $P$  с помощью ключа  $K$  получается зашифрованный текст  $C$ . Формула  $P = D_k(C)$  означает расшифровку зашифрованного текста  $C$  для восстановления открытого текста. Из этих формул следует, что  $D_k(E_k(P)) = P$ .  $E$  и  $D$  — это математические функции шифрования и дешифрования соответственно. На рисунке 1.1.2 представлена модель шифрования симметричным ключом.

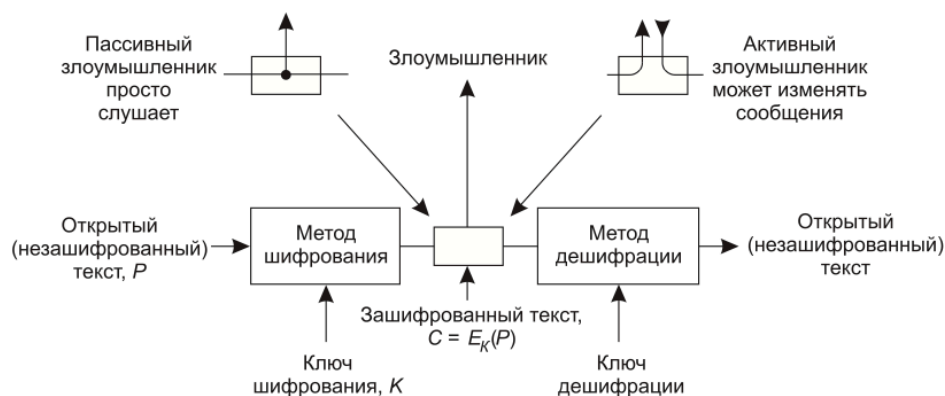


Рисунок 1.1.2 – Модель шифрования (для шифра с симметричным ключом)

Стоит сказать, что симметричный алгоритм шифрования подразумевает, что мы имеем ключ, который способен шифровать и расшифровывать данные. Это модель защиты данных хороша, у неё высокая защита и криптостойкость к атакам. Но в такой модели существуют сложности при транспортировке этого ключа, собеседникам, которые находятся весьма далеко друг от друга, будет проблематично передать ключ. Данную проблему решают ассиметричные алгоритмы шифрования. Как было сказано выше, ассиметричные алгоритмы предполагают наличие открытого и закрытого ключа.

В ассиметричном шифре происходит шифрование следующим образом, есть два участника переписки Алиса и Боб. У каждого участника есть открытый и закрытый ключ. Открытые ключи известны всем, а закрытые ключи пользователи сохраняют у себя. Суть заключается в том, что, если участник Алиса хочет отправить сообщение Бобу, она должна взять открытый ключ Боба и свой закрытый ключ, применить алгоритм генерации симметричного ключа, который использует уже для обычного шифрования. Бобу, получая зашифрованное сообщение, для расшифровки требуется взять открытый ключ Алисы и свой закрытый ключ, сгенерировать симметричный ключ и расшифровать сообщение путём того же шифра. Цель ассиметричной криптографии является в способе доставки ключа, потом уже применяется

симметричный способ для шифрования данных. На рисунке 1.1.3 вы можете наблюдать иллюстрацию асимметричной криптографии.

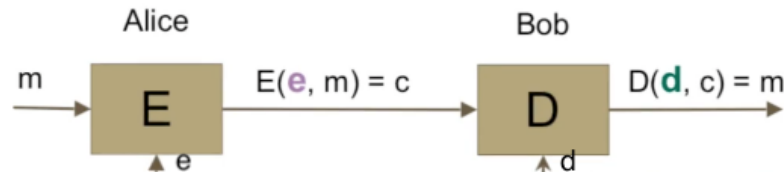


Рисунок 1.1.3 – Асимметричная криптография

Суть всей асимметричных шифров заключается в том, чтобы применять модель для вычисления ключа. Есть разные модели и способы — это сделать. Существуют модели логарифмические вычисления и эллиптические кривые. Расскажем более подробно о каждом из них.

Одним протокол из модели о логарифмическом вычислении является протокол Диффи-Хельмана. Он работает следующим образом. Алиса и Боб договариваются о двух больших простых числах,  $n$  и  $g$ , где  $(n-1)/2$  также является простым числом, кроме того, на число  $g$  накладываются некоторые дополнительные ограничения. Эти числа могут быть публичными, поэтому каждый просто устанавливает значение  $n$  и  $g$  и открыто сообщает о них собеседнику. Затем Алиса выбирает большое число  $x$  и держит его в тайне. Аналогично, Боб выбирает большое секретное число  $y$ .

Алиса использует протокол обмена ключами, отправив Бобу сообщение (открытым текстом), содержащее  $(n, g, g^x \bmod n)$ , как показано на рисунке 1.1.4. Боб отвечает ей сообщением, содержащим  $g^y \bmod n$ . Алиса возводит присланное Бобом число в степень  $x$  и делит его по модулю  $n$ , получая  $(g^{xy} \bmod n)$  — это и является секретным ключём для Алисы и Боба. Сложность заключается в том, что взломщику трудно определить числа, которые определяют симметричный ключ.



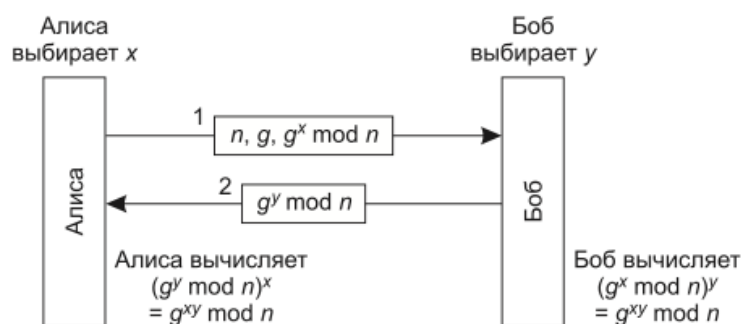


Рисунок 1.1.4 – Протокол обмена ключами Диффи-Хелмана

Эллиптическая криптография основана на математических свойствах эллиптических кривых, которые описываются уравнением вида  $y^2 = x^2 + ax + b \bmod p$ . При этом должно выполняться условие  $4a^3 + 27b^2 \neq 0 \bmod p$ . Пример эллиптической кривой представлен на рисунке 1.1.5. В эллиптической криптографии основная сложность заключается в вычислительной проблеме, известной как проблема дискретного логарифма на эллиптической кривой: имея начальную точку  $P$  и конечную точку  $Q$ , которые находятся на этой кривой, сложно определить число  $k$ , при котором  $Q = kP$ . Хотя начальная и конечная точки  $P$  и  $Q$  известны, вычисление множителя  $k$  является сложной задачей, что обеспечивает безопасность алгоритма.

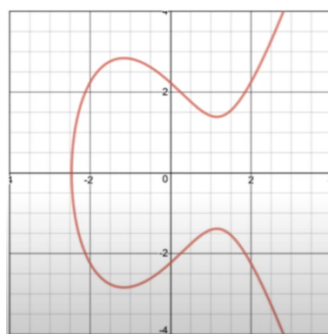


Рисунок 1.1.5 – Пример эллиптической кривой

## 1.2 Криптоанализ

Криптоанализ направлен на нахождение уязвимостей и методов для расшифровки данных без знания ключа. Существуют некоторые распространённые методы для криптоанализа, рассмотрим несколько из них.

Анализ методом грубой силы – данный метод использует перебор всех возможных ключей для нахождения правильного. Этот метод требует

огромных вычислительных ресурсов, особенно для современных алгоритмов с длинными ключами.

Криптоанализ по открытому тексту и шифротексту – нападающий имеет доступ к паре открытого текста и его зашифрованной версии. Используя эти данные, можно попытаться выявить ключ и структуру шифра. Данный способ наиболее подходит для анализа слабых и сильных сторон самого шифра.

### **1.3 Алгоритмы шифрования**

Перестановочные шифры меняют порядок следования символов в сообщении, но не изменяют их сами по себе. Для этого создается таблица открытого текста, где количество колонок равно длине ключа. Затем колонки нумеруются в соответствии с буквами ключа: первой становится колонка под буквой, расположенной ближе всего к началу алфавита, и так далее.

Одноразовые блокноты представляют собой метод шифрования, который обеспечивает высокий уровень безопасности. Принцип работы заключается в следующем: генерируется случайная последовательность бит, длина которой равна длине сообщения. Затем сообщение шифруется с использованием операции исключающего ИЛИ (XOR) между сообщением и ключом. В результате получается шифротекст, который практически невозможно разгадать, поскольку каждый бит шифротекста зависит от случайного ключа. Для расшифровки используется тот же ключ, что и для шифрования. Сложность заключается лишь в “случайном” выборе ключа. Дело в том, что абсолютно случайных алгоритмов в детерминированных машинах не существуют. В компьютерах создают так называемые псевдослучайные алгоритмы, которые являются, на первый взгляд, случайными.

В симметричных алгоритмах используют разные методы для создания алгоритмов шифрования. Основные методы: перестановка байтов, блочное шифрование. Блочные шифры весьма сложные для реализации и понимания с точки зрения человека. Машины их с лёгкостью читают и понимают. Поэтому в данной работе был сделан выбор в пользу потоковых шифров.

## 1.4 Потокковые шифры

Потоковые шифры преобразуют открытый текст в шифротекст по одному биту за операцию. Генератор потока ключей выдаёт поток битов:  $k_1, k_2, k_3, \dots, k_i$ . Этот поток ключей и поток битов открытого текста,  $p_1, p_2, p_3, \dots, p_i$ , подвергаются операции “исключающее или”, и в результате получается поток битов шифротекста:  $c_i = p_i \oplus k_i$ . [2]

При дешифровании операция XOR выполняется над битами шифротекста и тем же самым потоком ключей для восстановления битов открытого текста:  $p_i = c_i \oplus k_i$ ,  $p_i \oplus k_i \oplus k_i = p_i$ .

Потоковые шифры особенно полезны для шифрования бесконечных потоков коммуникационного трафика, например, канала, связывающего два компьютера, приложений, требующих высокой скорости обработки и малой задержки, таких как потоковое аудио и видео. [3] Схема потокового шифра представлена на рисунке 1.4.1.

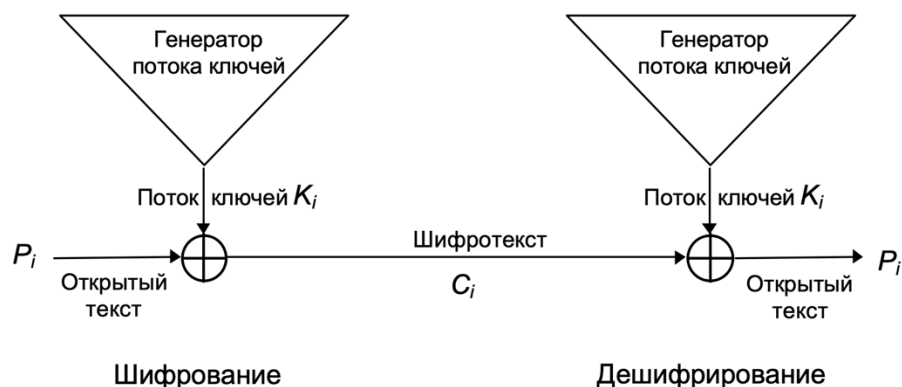


Рисунок 1.4.1 – Схема потокового шифра

Безопасность системы целиком зависит от характеристик генератора потока ключей. Этот генератор включает три основных компонента, смотрите рисунок 1.4.2. Внутреннее состояние описывает текущее состояние генератора потока ключей. Два генератора с одинаковыми ключами и одинаковым внутренним состоянием создают идентичные потоки ключей. Функция выхода преобразует внутреннее состояние в бит потока ключей. Функция обновления состояния генерирует новое внутреннее состояние из текущего.

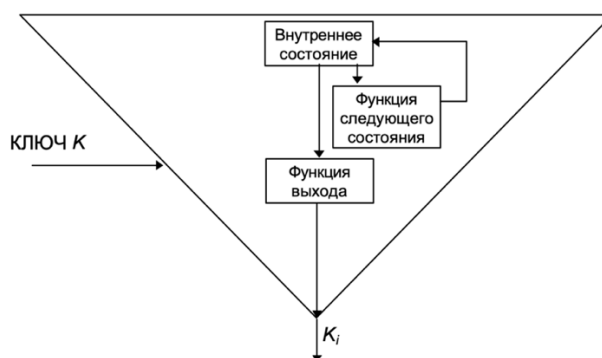


Рисунок 1.4.2 – Устройство генератора потока ключей

Примеры популярных потоковых шифров включают RC4, Salsa20 и ChaCha20, каждый из которых имеет свои уникальные характеристики и области применения.

### 1.5 Мобильные приложения

Мобильное устройство — это портативный вычислительный аппарат, предназначенный для мобильного использования. Мобильные устройства включают смартфоны, планшеты, умные часы и другие гаджеты, которые обеспечивают доступ к информации и функциям в любое время и в любом месте. В данной работе мы будем приравнивать понятие мобильного устройства к телефону.

Существует огромное разнообразие мобильных устройств, но для обычных пользователей преобладают две основные платформы: iOS и Android. Для разработки приложений под Android необходимо знание языков программирования, таких как Java и Kotlin. Также для кроссплатформенной разработки всё более популярным становится использование Flutter, который позволяет создавать приложения для обеих платформ с использованием языка Dart.

Для разработки приложений под iOS нужно владеть языком программирования Swift, который был разработан Apple специально для их экосистемы. Swift предоставляет множество современных возможностей для удобного и безопасного программирования.

Помимо знания языков программирования, для создания полноценных мобильных приложений требуются различные фреймворки и библиотеки. Они

используются как для создания интерфейса, так и для реализации функциональных возможностей приложения. Например, для Android разработчики часто используют Android Jetpack, набор библиотек, предназначенных для упрощения разработки. В iOS-экосистеме широко используются фреймворки, такие как UIKit и SwiftUI для создания интерфейсов.

В данной работе будет использоваться язык программирования Swift с использованием фреймворка SwiftUI, речь о котором пойдет ниже.

## **1.6 Swift, SwiftUI**

Swift – мощный язык программирования, разработанный компанией Apple для создания приложений на платформах iOS, macOS, watchOS и tvOS. Он был представлен в 2014 году и с тех пор стал основным языком для разработки приложений в экосистеме Apple, заменив Objective-C.

SwiftUI — это современный фреймворк для создания пользовательских интерфейсов на платформах Apple. Введенный в 2019 году, SwiftUI позволяет разработчикам создавать интерфейсы с помощью декларативного синтаксиса. Это значит, что вместо традиционного программирования, где описываются шаги для создания интерфейсов, разработчики описывают, как интерфейс должен выглядеть и вести себя при различных состояниях.

Более подробно будет рассмотрен синтаксис Swift и SwiftUI в разработке самого приложения.

## **1.7 Шаблоны проектирования приложений для устройств**

Понимание и применение архитектурных шаблонов помогает гарантировать, что ваше приложение будет хорошо структурировано. Что позволит легко поддерживать и продвигать продукт. В большинстве своём, многие крупные приложения работают структурированно, что позволяет разделять ответственность на модульные компоненты, которые легко поддерживаются и заменяются. Каждый модуль должен отвечать за свою функциональность, поэтому в хорошей архитектуре, представление интерфейса никак не использует бизнес логику в своём коде.

Архитектурный паттерн (или шаблон) — это повторяемое решение общей проблемы, встречающейся в разработке программного обеспечения. Паттерны описывают базовые структурные схемы и взаимодействия компонентов системы, которые могут быть повторно использованы для эффективного решения типовых задач в различных проектах. Они помогают разработчикам создать более структурированный и легко поддерживаемый код, улучшая масштабируемость и надежность программных систем. Примеры архитектурных паттернов включают MVC (Model-View-Controller), MVVM (Model-View-ViewModel). В работе будет использован паттерн MVVM, рассмотрим более подробно этот архитектурный шаблон.

Паттерн MVVM разделяет код приложения на три компонента: модель (model), представление (View) и модель представления (ViewModel). Рассмотрим каждый компонент отдельно.

Model отвечает за управление данными и бизнес-логикой приложения. Не зависит от представления и модели представления. Обычно, модель содержит некий сервис, который предоставляет данные. Это могут быть данные из сети интернет или локальной базы данных. Также часто в модели можно увидеть структуры самих объектов. Структуры описывают методы и свойства объектов.

View отвечает за визуальное отображение данных. Представление связывается моделью представления для получения данных и обновления интерфейса в ответ на изменения данных. Также представление должно быть простым и не содержать бизнес-логики.

ViewModel является посредником между моделью и представлением.

Архитектурный паттерн MVVM вы можете наблюдать на рисунке 1.7.1. На рисунке изображены три компонента. View общается с ViewModel посредством действий пользователя (User Action). View знает о ViewModel, но ViewModel не знает о View. ViewModel сообщает лишь об изменениях данных (binding), в свою очередь, View прослушивает изменения и мгновенно

отображает их на экране. ViewModel общается с Model, которая содержит все необходимые данные из базы и методы.

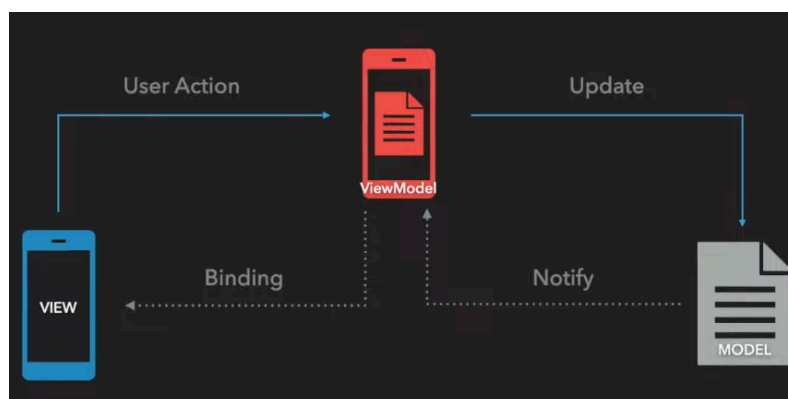


Рисунок 1.7.1 – Архитектурный шаблон MVVM

## 1.8 Способы тестирования

В данном разделе пойдёт речь о тестировании мобильных приложений. Существует множество методов тестирования, но рассмотрим Unit-тесты, которые будут использованы в проекте.

Unit-тесты – это вид тестов, которые проверяют функциональность модулей, проверяя каждый компонент отдельно. В Xcode, среде разработки под устройства Apple, есть встроенная функциональность для написания и запуска тестов — XCTest. Эти тесты проверяют отдельные методы и функции на корректность их работы, обеспечивая надёжность и предсказуемость отдельных частей кода. В рамках разработки алгоритма шифрования мы подробно рассмотрим использование XCTest для модульного тестирования.

UI-тесты (тесты пользовательского интерфейса) проверяют работу приложения с точки зрения пользователя. Они автоматизируют взаимодействие с интерфейсом, проверяя, что элементы управления работают правильно и интерфейс отображается корректно. Данный вид тестов мы будем проводить вручную, чтобы найти ошибки и недочёты в мобильном приложении. Этот вид тестов будет подробнее описан в разделе разработки мобильного приложения.

## 1.9 UML диаграммы

UML (англ. Unified Modeling Language — унифицированный язык моделирования) — язык графического описания для объектного

моделирования в области разработки ПО и другого. UML нужен, чтобы превращать абстракции в визуальные модели и схемы. Поэтому применять его можно во многих областях: в программировании — чтобы наглядно видеть связи между классами и другими частями приложения или чтобы построить карту поведения пользователя на сайте; в дизайне — чтобы создавать интерфейсы и понимать, как пользователи будут взаимодействовать с ними; в бизнесе — чтобы визуально представлять, как работают бизнес-процессы или ведётся документооборот в организации.

В данной работе будет использовано четыре вида диаграмм: диаграмма классов, диаграмма use case, диаграмма состояний и диаграмма действий. Рассмотрим подробнее каждый из них.

Диаграмма классов (описание базы данных) – структурная диаграмма UML, демонстрирующая общую структуру иерархии классов системы, из коопераций, атрибутов (полей), методов, интерфейсов и взаимосвязей (отношений) между ними.

Диаграмма use case (сценариев использования) – данный вариант диаграмм предназначен для упрощения взаимодействия с будущими пользователями системы, с клиентами, и особенно пригодятся для определения необходимых характеристик системы.

Диаграмма состояний показывает, как объект переходит из одного состояния в другое.

Диаграмма деятельности – UML-диаграмма, на которой показаны действия, состояния которых описано на диаграмме состояний.



## **2. Разработка алгоритма шифрования**

Данный раздел описывает реализацию уникального потокового алгоритма шифрования. В процессе разработки алгоритма поточного шифрования были учтены следующие факторы. [4]

**Безопасность:** основным критерием выбора была стойкость алгоритма к криптоанализу. В алгоритме были использованы современные криптографические методы и тщательно протестированы на наличие уязвимостей.

**Производительность:** поточные шифры обеспечивают высокую скорость шифрования и дешифрования, что делает их идеальным выбором для приложений, требующих обработку данных в реальном времени, таких как защита сетевого трафика или потоковое видео.

**Ресурсоэкономичность:** алгоритм разработан с учетом ограниченных ресурсов, таких как вычислительная мощность и объем памяти. Он требует минимального количества оперативной памяти для хранения ключа и состояния генератора псевдослучайных чисел.

**Простота реализации:** алгоритм разработан с учетом его легкости в реализации и интеграции в различные приложения. Это обеспечивает удобство использования и распространения.

Стоит отметить, что нам необходима псевдослучайный алгоритм, который бы мог при малейшем изменении ключа, сильно изменять данные.

### **2.1 Описание шифра**

Суть потовых шифров заключается в том, что мы генерируем псевдослучайную последовательность, и применяя концепцию, как в одноразовых блокнотах, выполняем операцию XOR над текстом, полученный текст очень сложно расшифровать в понятный для человека язык.

В реализации алгоритма была принята теория, что сложность алгоритма заключается в запутанности. Для компьютеров этот подход не имеет значения, поскольку машины просто выполняют последовательность операций. Следуя принципу Керкгоффса, который гласит, что алгоритмы шифрования должны

быть общедоступными, а секретными должны оставаться только ключи, был разработан данный алгоритм. [5]

За основу алгоритма была взята идея, концепция s-блоков. Суть s-блоков заключается в том, что мы создаём некий блок с размеров 256. Так как символ представляется в памяти компьютера как 1 байт или 8 бит, следовательно, мы можем в 1 байте хранить 256 значений. То s-блоки будут выступать неким псевдослучайным генератором для ключей. Суть заключается в том, что, используя ключ, алгоритм формирует псевдослучайную последовательность чисел, которая в дальнейшем будет использована для создания шифротекста из исходного текста.

Был придуман алгоритм для генерации s-блока, более подробно вы можете ознакомиться на рисунке 2.1.1. На рисунке представлен алгоритм генерации s-блоков.

```
for i in 0.. $S.count$  {  
     $j = ((S.count - j) + S[i] + \text{Int}(K[i \% K.count])) \% S.count$   
    swap(i, j)  
}
```

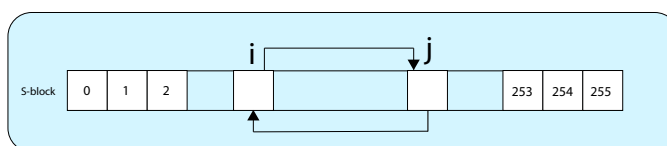


Рисунок 2.1.1 – Алгоритм генерации s-блоков

В данной реализации используется идея перестановки. Проходя по всем индексам s-блока, мы меняем их с другим индексом, который генерируем на основе ключа. Стоит отметить, что ключ  $K$  является входным параметром для генерации, и ключ представляет собой массив типа  $UInt8$ .  $UInt8$  – тип данных в языке Swift, который принимает значения от 0 до 255. Функция `swap` представляет собой перестановку значений массива. В данном случае индекс  $i$  становится на место  $j$ , а индекс  $j$  на место  $i$ .

Для запутывания алгоритма, был придуман ещё один метод, который выполняет перестановку уже в сгенерированном s-блоке, более подробно вы можете ознакомиться на рисунке 2.1.2.

```
function getKey(i int, j inout int) {
    j = (S[i]+S.count % (j + 1)) % S.count
    swap(i, j)
    return UInt8(S[(S[i] + S[j]) % S.count])
}
```

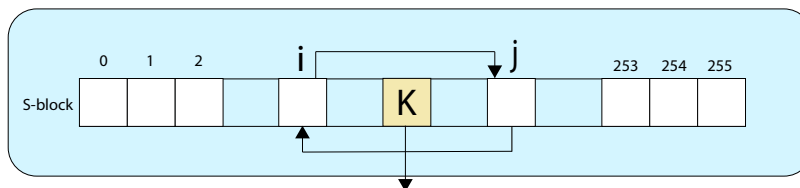


Рисунок 2.1.2 – Алгоритм генерации ключа шифрования

В данной реализации мы применяем ту же концепцию перестановки. На рисунке мы генерируем новую последовательность уже сгенерированного s-блока. После уже получаем значение ключа для операции XOR над данными. Нужно заметить, что параметр *j* имеет атрибут *inout*, данный атрибут говорит о том, что параметр является ссылочным, то есть, если он изменяется в самой функции, он изменяется и за её пределами – это особенность языка программирования Swift. Параметр *i* представляет собой индекс символа в исходном тексте.

Таким образом, генерация секретного ключа происходит в два этапа, второй этап характеризуется тем, что обновление s-блока происходит каждый раз, когда мы получаем значение ключа для шифрования сообщения. Стоит отметить, что в самой реализации будет происходить шифрование непосредственно, то есть каждый байт сообщения вызывает функцию генерации значения ключа, затем происходит операция XOR. Так выполняется до тех пор, пока сообщение не закончиться.

## 2.2 Реализация

Реализация алгоритма началась с создания класса Cipher. Далее было создано свойство SBlock, который как раз является тем s-блоком в схеме выше. Смотрите рисунок 2.2.1.

```
var SBlock: [Int] = Array(repeating: 0, count: 256) // S-block
```

Рисунок 2.2.1 – Инициализация S-блока

Было принято решение реализовать генерацию ключей с использованием метода перестановки. Этот метод позволяет привнести в алгоритм дополнительный уровень псевдослучайности, что повышает криптографическую стойкость генерируемых ключей. Метод перестановки будет доступен исключительно внутри класса, обеспечивая безопасность и инкапсуляцию процесса генерации ключей. Подробная реализация этого метода представлена на рисунке 2.2.2. На данном рисунке вы можете наблюдать, что создаётся дополнительная переменная, которая необходима для корректной перестановки.

```
// Перестановка элементов
private func swap(i: Int, j: Int) {
    let tmp = SBlock[i]
    SBlock[i] = SBlock[j]
    SBlock[j] = tmp
}
```

Рисунок 2.2.2 – Реализация метода Swap

В рамках проекта был реализован метод stringFromBytes(\_ bytes: [UInt8]) -> String, который преобразует массив байтов в строку типа String. Этот метод позволяет конвертировать исходные данные в формат, понятный человеку, что является важным этапом при работе с криптографическими алгоритмами и другими задачами. Подробная реализация этого метода представлена на рисунке 2.2.3.

```
func stringFromBytes(_ bytes: [UInt8]) -> String {
    return NSString(bytes: bytes,
                    length: bytes.count,
                    encoding: String.Encoding.utf8.rawValue)
    as? String ?? ""
}
```

Рисунок 2.2.3 – Реализация метода stringFromBytes

Далее был создан метод `encryptDecrypt(data: [UInt8], key: [UInt8]) -> [UInt8]`, который принимает на вход массив байтов сообщения и массив байтов ключа, а на выходе функция возвращает массив байтов. Более подробно ознакомиться с данным методом вы можете на рисунке 2.2.4.

```
// Шифрование & Расшифрование
func encryptDecrypt(data: [UInt8], key: [UInt8]) -> [UInt8] {
    var outputBytes: [UInt8] = []
    generationSBlock(key: key)
    var indexForSwapKey1 = 0
    var indexForSwapKey2 = 0
    // шифрование данных
    for index in 0..

```

Рисунок 2.2.4 – Реализация метода `encryptDecrypt`

На рисунке вы можете наблюдать массив `outputBytes`, который является выходным массивом байтов. Стоит отметить, так как потоковые шифры работают с операцией XOR, то для шифрования и дешифрования нужен всего один ключ.

Далее идёт функция генерации s-блока, по схеме, которую описывали выше. Более подробно с реализацией вы можете ознакомиться на рисунке 2.2.5. Пару слов об реализации `generationSBlock`. На вход мы принимает массив байтов ключа. Далее происходит инициализация значений массива `SBlock` от 0 до 255. После происходит цикл генерации самого `SBlock` по схеме, которая изображена на рисунке 2.1.1.

```
// генерация S-блока
private func generationSBlock(key: [UInt8]) {
    for i in 0..

```

Рисунок 2.2.5 – Реализация метода `generationSBlock`

Далее, на рисунке 2.2.4 можно наблюдать инициализацию переменных `indexForSwapKey1` и `indexForSwapKey2`, которые являются индексами для

генерации ключа. Затем в методе encryptDecrypt происходит шифрование и дешифрование данных посредством операции XOR.

Вы можете наблюдать на рисунке 2.2.4, что цикл проходит по всем байтам сообщения, шифруя их значением сгенерированного ключа. Реализацию метода getKey, которую вы наблюдали на рисунке 2.1.2, вы можете видеть на рисунке 2.2.6. Первый параметр является индексом байта сообщения, второй представляет собой индекс, который используется для последующей генерации ключа. В данной функции применён параметр inout для второго параметра, следовательно, он изменяется и за пределами функции.

[6]

```
// Получение ключа
private func getKey(indexForSwapKey1: Int, indexForSwapKey2: inout Int) -> UInt8 {
    indexForSwapKey2 = (SBlock[indexForSwapKey1] + SBlock.count % (indexForSwapKey2 + 1)) % SBlock.count
    swap(i: indexForSwapKey1, j: indexForSwapKey2)
    return UInt8(SBlock[(SBlock[indexForSwapKey1] + SBlock[indexForSwapKey2]) % SBlock.count])
}
```

Рисунок 2.2.6 – Реализация метода getKey

Для дешифрования сообщения, на вход функции подаётся массив байтов зашифрованного сообщения и тот же ключ, что был использован для шифрования. Таким образом, был реализован шифр, который генерирует псевдослучайную последовательность, а затем использует эту последовательность для шифрования текста посредством операции XOR. По той причине, что это потоковый шифр, функция шифрования и дешифрования одинаковы.

## 2.3 Тестирование

Сначала были проведены ручные тесты, где данные были подобраны. Для данного вида тестов были созданы переменные cipher, keyString, dataString. Расскажем подробнее о каждой переменной. Cipher – является объектом класса, который содержит методы шифрования и дешифрования. KeyString – является переменной типа String, которая представляет собой ключ для алгоритма. DataString – является переменной типа String, которая представляет собой сообщение. Данные переменные вы можете наблюдать на рисунке 2.3.1.

```
let cipher = Cipher()
let keyString = "key8"
let dataString = "Привет мир!"
```

Рисунок 2.3.1 – Переменные cipher, keyString, dataString

Далее были реализованы выводы посредством многократного вызова стандартной функции print, смотрите рисунок 2.3.2. Для передачи в качестве параметра для функции шифрования строк, использовался метод Array(<String>.utf8), который преобразует строку в массив байтов. Так как на выходе шифрования и дешифрования функции получается массив байтов, для корректного преобразования из массива байтов в строку, используется метод stringFromBytes.

```
print("Ключ: \t\t\t\t\t", keyString)
print("Текст: \t\t\t\t\t", dataString)
print("Текст в байтах: \t\t\t\t", Array(dataString.utf8))

var ciphertext = cipher.encryptDecrypt(data: Array(dataString.utf8), key: Array(keyString.utf8))
print("Зашифрованный в байтах: \t\t", ciphertext)

var decryptedText = cipher.encryptDecrypt(data: ciphertext, key: Array(keyString.utf8))
print("Расшифрованный: \t\t\t\t", cipher.stringFromBytes(decryptedText))
```

Рисунок 2.3.2 – Реализация многократного вызова результата

После запуска проекта можно наблюдать результат, который представлен на рисунке 2.3.3.

Ключ:	key8
Текст:	Привет мир!
Текст в байтах:	[208, 159, 209, 128, 208, 184, 208, 178, 208, 181, 209, 130, 32, 208, 188, 208, 184, 209, 128, 33]
Зашифрованный в байтах:	[255, 121, 158, 69, 160, 140, 234, 92, 92, 101, 140, 8, 20, 46, 207, 185, 250, 74, 239, 179]
Расшифрованный:	Привет мир!

Рисунок 2.3.3 – Вывод результата

Поскольку Swift работает с кодировкой unicode, это накладывает некоторые особенности на обработку данных, многие символы представимы в виде нескольких байт. Например, символ "ё" в unicode представлен двумя кодовыми точками: "е" и диакритическим знаком. Этот факт усложняет задачу криптоаналитикам, так как зашифрованный текст, включающий такие символы, становится труднее анализировать. Подробный пример этого случая можно увидеть на рисунке 2.3.4. Данный пример прекрасно иллюстрирует, что даже если и удастся получить читаемое сообщение, не факт, что это именно оно, ведь может произойти так, что несколько байт какого-то сложного

символа объединяться в некое слово, но никак не сложный символ. В данном случае кодировка unicode очень сильно помогает криптографам.

```
Ключ:          key
Текст:         ø
Текст:         [201, 135]
Зашифрованный: [145, 100]
Расшифрованный: [201, 135]
Program ended with exit code: 0
```

Рисунок 2.3.4 – Тестирование

Для демонстрации случая, когда при малейшем изменении ключа, изменяется полностью шифротекст, вы можете наблюдать на рисунках 2.3.5 и 2.3.6.

```
Ключ:          key
Текст:         Hello World!
Текст:         [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33]
Зашифрованный: [16, 134, 17, 110, 9, 192, 126, 160, 117, 155, 175, 1]
Расшифрованный: [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33]
Program ended with exit code: 0
```

Рисунок 2.3.5 – Тестирование

```
Ключ:          ke
Текст:         Hello World!
Текст:         [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33]
Зашифрованный: [252, 89, 33, 29, 244, 143, 97, 18, 14, 117, 8, 188]
Расшифрованный: [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33]
Program ended with exit code: 0
```

Рисунок 2.3.6 – Тестирование

Далее были проведены unit тесты. Unit тесты хороши, когда проводят большое количество тестов. В данном случае мы провели более 50 тестов на сложные символы, где в состав ключей и сообщений входили сложные символы. Результаты тестирования вы можете наблюдать на рисунке 2.3.7. Все тесты прошли и показали положительный результат.

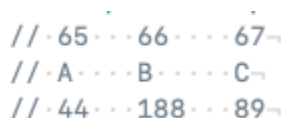




Инициализация ключа занимает время выполнения  $O(n)$ , где  $n$  – длина ключа. Генерация ключевого потока занимает время выполнения  $O(1)$  для каждой итерации. Следовательно, общая временная сложность  $O(n + m)$ , где  $n$  – длина ключа,  $m$  – длина сообщения.

Для проведения атаки грубой силы понадобится  $2^n$ , где  $n$  – длина ключа в битах. То есть, если у нас ключ в 1 байт, то нам нужно  $2^8$  итераций, чтобы разгадать исходное сообщение. Поэтому небезопасно будет выбирать ключ меньшего размера. И желательно, чтобы сообщения было большим для обеспечения безопасности. То есть к исходному тексту добавляется некая последовательность символов, которая после расшифровки удаляется, данное замечание можно учесть и в дальнейшем улучшить шифр.

Далее были проведены небольшие тесты на выявление зависимостей, корреляции между данными. Один из тестов представлен на рисунке 2.4.1. Стоит пояснить, верхняя строчка говорит о том, какая цифра в таблице Unicode соответствует букве, которая расположена на второй строчке. Были проведены, где ключи являлись одиночные символы, которые представлены на второй строчке. Тесты проводились над сообщением в один символ. Этот символ был “А”, в таблице Unicode этому символу соответствует число 65. Результат шифрования одиночного символа “А” символами из второй строчки вы можете наблюдать на третьей строчке. То есть результат шифрования символа “А” при ключе “А” равен 44 и так далее. Стоит сказать, что явной закономерности между символами, которые идут подряд, а именно: “А”, “В”, “С”, не наблюдалось, если разница между первым и вторым результатом составляет 144 единицы, то разница между вторым и третьим тестом составляет 76 единицы по модулю 256. Этот и другие примеры показывают, что шифр безопасен и работает корректно на небольшом и, казалось бы, похожих данных.



```
// 65 66 67
// A B C
// 44 188 89
```

Рисунок 2.4.1 – Тестирование

### **3. Разработка сервиса для обмена данными между двумя мобильными устройствами**

Перед тем как создать приложения, были выделены некоторые свойства, которыми должно обладать приложения, рассмотрим каждый из этих свойств по отдельности.

Безопасность приложения должно обеспечиваться конфиденциальным обменом данных и защитой передаваемых данных пользователей. Также у пользователей не должно возникать недоверия к самому приложению и к сервисам, которое оно использует. Общение пользователей должно происходить в туннельной концепции. Туннельная концепция заключается в том, что привязка пользователей идёт к устройствам, которые они используют. То есть, при смене учётной записи, меняются ключи шифрования и данные прошлой переписки становятся недоступны. Данная концепция предотвращает случаи, когда третьи лица, знающие пароль и логин, не смогут прочитать предыдущее общение. В данном случае лучшим решением для обеспечения безопасности будет использование асимметричное шифрования для передачи ключей, а уже затем будет использован потоковый алгоритм шифрования, который был разработан в предыдущем разделе.

В приложении важны передаваемые данные. Так как большую часть общения пользователей составляют текстовые сообщения, то выбор пал на чат-приложение. Чат является наилучшим решением для визуального отображения сообщений пользователей. В самом предложении должен присутствовать функционал авторизации и создания аккаунта. Чат также должен иметь некое информационное окно, в котором будет предупреждение о нечитаемых данные при смене устройств одного из пользователей. В чате должна быть вкладка, где пользователь выбирает, кому он может написать сообщение. На главном экране должны присутствовать недавние сообщения и временные метки, когда чат был изменён. Должна быть кнопка выхода из учётной записи. У пользователей есть возможность без труда отправить сообщение.

Простота в приложении очень важна. Приложение будет использовать зелёный цвет как основной. Интерфейс должен быть интуитивно понятным и не быть двусмысленным.

В приложении должен быть сервис, который обеспечивает обмен сообщениями по сети интернет. Сервис должен быть проверенный. Сервис предоставляет лишь хранилище зашифрованных сообщений. Более подробно речь о сервисе пойдёт в разделе разработки мобильного приложения.

### **3.1 Описание мобильного приложения**

Приложение будет представлять собой чат, где безопасность будет определяться разработанным потоковым шифром в данной работе.

Для безопасного обмена ключами будет использоваться протокол Диффи-Хельмана на эллиптических кривых, данный протокол обеспечивает генерацию одного общего ключа, который будет применён при шифровании. Данный протокол уже есть в стандартной библиотеке swift, а именно в библиотеке `cryptokit`. Более подробно об этом протоколе и способе его использования будет рассказано в разделе разработки мобильного приложения.

В качестве основного цвета будет выбран зелёный цвет.

В качестве сервиса, который обеспечивает хранение как данных пользователей, так и их сообщений, был выбран сервис от google, firebase. Более подробно о firebase будет рассказано в следующем разделе. Причина, по которой был выбран firebase, заключается в том, что данный сервис прост в интеграции и предоставляет собой бесплатное пространство для размещения данных в сети интернет.

Приложение будет написано на языке swift, с использованием фреймворка swiftui. При разработке приложения будет использован архитектурный паттерн mvvm.

### **3.2 Firebase**

Firebase – это платформа для разработки мобильных и веб-приложений, созданная компанией google. Он представляет множество инструментов и

услуг, которые помогают разработчикам ускорить процесс создания, развертывания и управления приложениями. Перечислим основные компоненты: firebase authentication, cloud firestore, firebase storage. Рассмотрим каждый компонент по отдельности.

Firebase authentication – служба аутентификации, которая позволяет легко добавлять в приложение функциональность регистрации и входа пользователей с использованием различных методов, таких как электронная почта и пароль, телефонный номер, а также сторонние провайдеры, такие как google и другие.

Cloud firestore – расширенная облачная база данных NoSQL, которая обеспечивает мощную поддержку запросов и масштабируемость. Firestore также синхронизирует данные в реальном времени и предлагает более сложные возможности для структурирования и запроса данных.

Firebase storage – служба для хранения и передачи файлов, таких как изображения, видео и другие медиафайлы. Firebase storage интегрируется с firebase authentication для обеспечения безопасности доступа к файлам.

Более подробно об использовании и интеграции будет сказано в разделе разработки мобильного приложения.

### **3.3 UML схема**

Были созданы таблицы uml, которые используются для демонстрации функциональных возможностей приложения. Стоит отметить, что данные диаграммы создавались, учитывая возможности firebase. Более подробнее об collection и document написано в разделе разработки мобильного приложения.

Была реализована диаграмма классов, с которой можно ознакомиться на рисунке 3.3.1. На рисунке вы можете рассмотреть таблицы authentication, storage, users, users (данные), messages (от кого отправлено), messages (кому отправлено), messages (данные), recent\_messages, messages (список последних сообщений), recent\_message (данные). Немного стоит сказать о каждой таблице.

Таблица authentication служит для списка пользователей, которые существуют в системе базы данных, данная таблица будет часто использоваться для авторизации и регистрации новых пользователей.

Таблица storage хранит в себе изображения профилей пользователей.

Таблицы users и users (данные) связаны между собой в системе firebase. Таблица users (данные) хранит в себе информацию пользователей, в данной таблице находится важное свойство publicKey или открытый ключ, который будет применён при шифровании.

Таблица messages (от кого отправлено), messages (кому отправлено), messages (данные) связаны в системе firebase. Таблица messages (данные) хранит в себе все сообщения пользователей и данные о самих сообщениях.

Таблицы recent\_messages, messages (список последних сообщений), recent\_message (данные) связаны между собой в системе firebase. Таблица recent\_message (данные) хранит в себе информацию о последнем сообщении между пользователями.

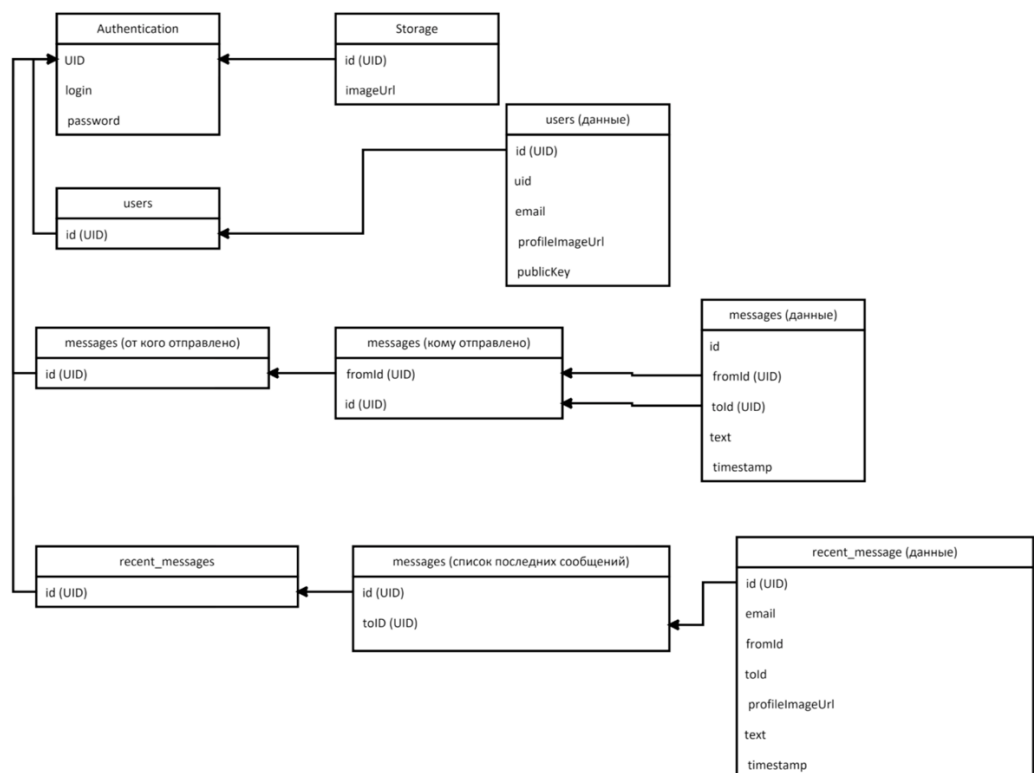


Рисунок 3.3.1 – Диаграмма классов

Далее была реализована диаграмма use case (сценариев использования), с которой можно ознакомиться на рисунке 3.3.2. Опишем некоторые функции, которые может выполнять обычный пользователь приложения. Пользователь имеет возможность проводить авторизацию/регистрацию, просмотр сообщений, общение посредством сообщений, возможность выбора собеседника.

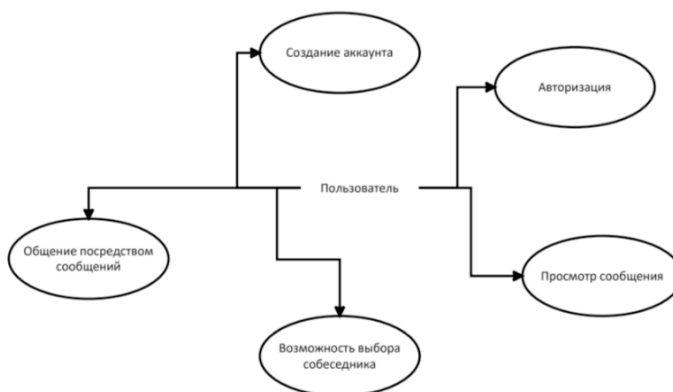


Рисунок 3.3.2 – Диаграмма use case (пользователь)

Так как приложение подразумевает только одну роль, а именно пользователя, то диаграмму, изображённую на рисунке 3.3.2, можно считать единственно верной для мобильного приложения. Но так как существует надзиратель, администратор, который может взаимодействовать с базой данных через сайт, то была выделена диаграмма, которую можно наблюдать на рисунке 3.3.3. Администратор имеет возможность изменять/удалять пользователей, изменять/удалять сообщения, изменять/удалять базу данных приложения.



Рисунок 3.3.3 – Диаграмма use case (администратор)

Далее была реализована диаграмма состояний, с которой можно ознакомиться на рисунке 3.3.4. Опишем каждый компонент, который изображён на диаграмме 3.3.4. После открытия приложения пользователем, приложения проверяет, авторизован пользователь или нет. Если пользователь не авторизован в системе, то появляется экран авторизации и регистрации, где пользователь вводит необходимые данные, чтобы авторизоваться в системе. Если пользователь авторизован в системе, тогда появляется главный экран, где изображены активные чаты, это такие чаты, в которые недавно были добавлены сообщения. Далее пользователь имеет возможность выбрать активный чат и отправлять сообщение выбранному собеседнику. Пользователь также имеет возможность начать новый чат с другим собеседником. У пользователя есть возможность выйти из системы, когда он нажимает “log out”. Выход из приложения осуществляется путём сворачивания приложения и удаления его из ленты активных приложений.

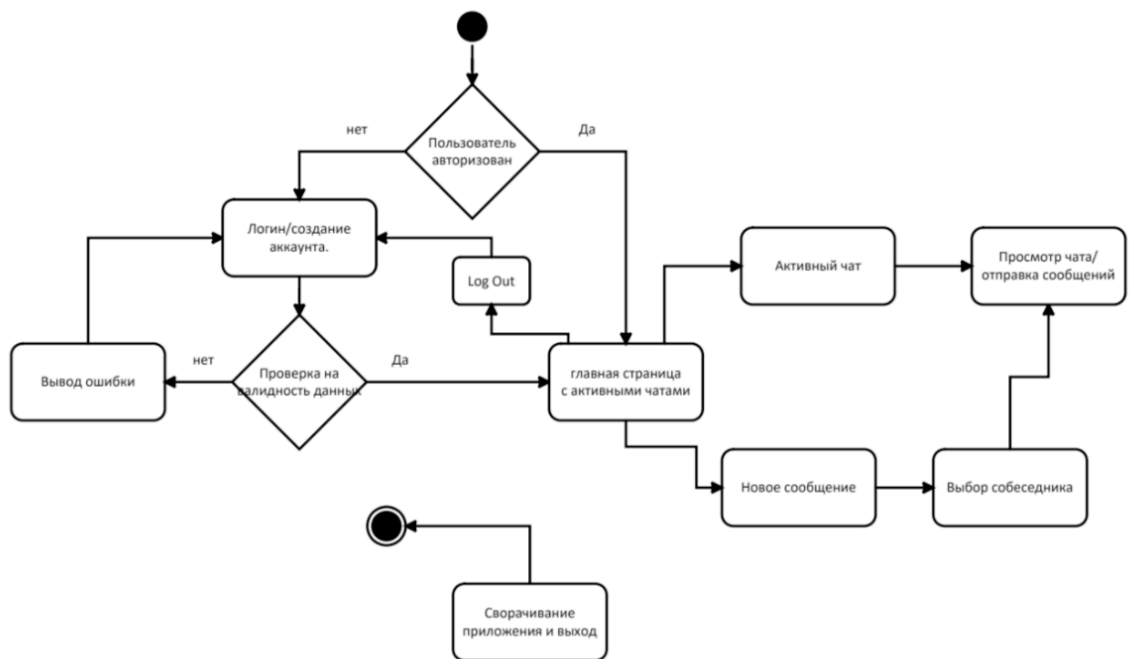


Рисунок 3.3.4 – Диаграмма состояний

Далее была реализована диаграмма действий, которая является расширенной версией диаграммы состояний. Диаграмма действий представлена на рисунке 3.3.5. Проведём анализ диаграммы, которая изображена на рисунке 3.3.5. Стоит сказать, что данная диаграмма содержит



три уровня, а именно: пользователь, front-end, back-end. Приложение начинается свою работу из состояния back-end, где происходит проверка на авторизацию пользователя. Далее все проверки, загрузка и отправка сообщений происходят в состоянии back-end. В состоянии front-end происходит прорисовка всех компонентов, с которыми пользователь может взаимодействовать. В любом из состояний приложения пользователь может свернуть и удалить сессию приложения. В отличие от диаграммы состояний, в данной диаграмме появились дополнительные методы, которые условно показывают моменты, когда приложение загружает/отправляет данные, выполняет проверки или проходит авторизацию.

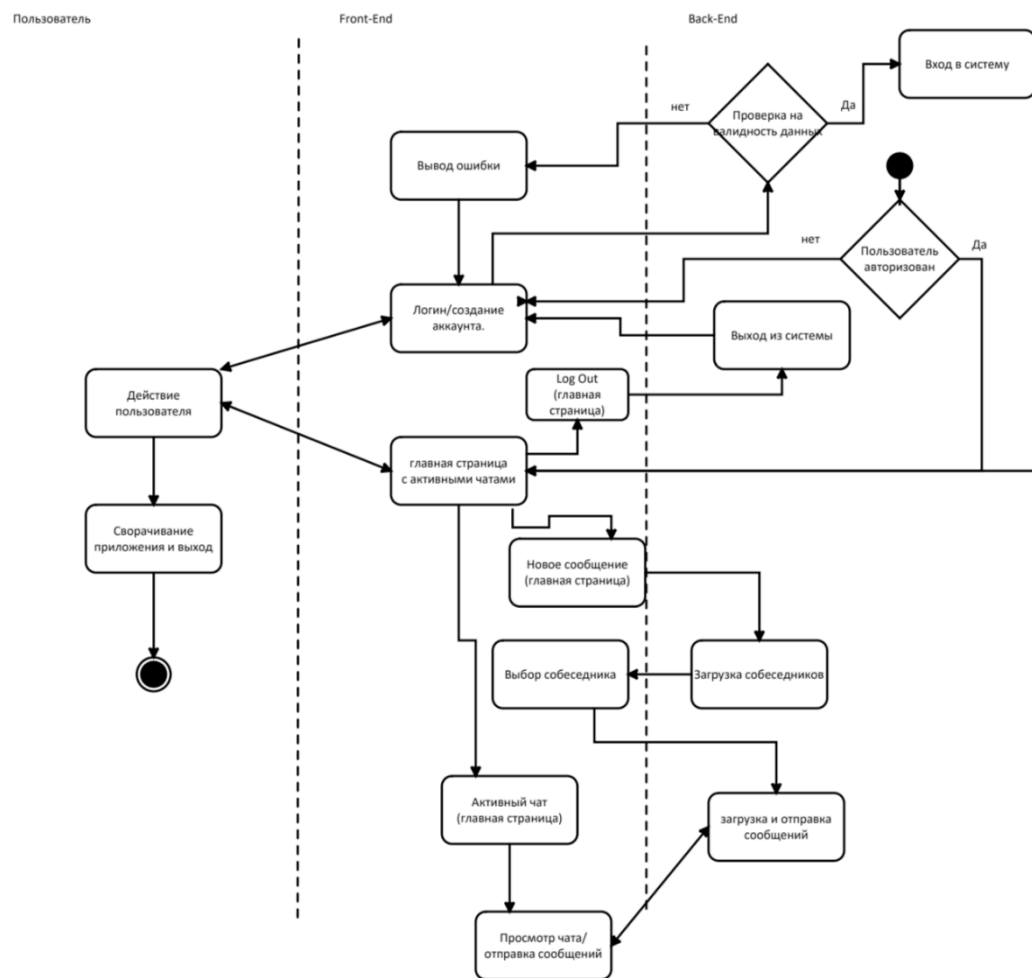


Рисунок 3.3.5 – Диаграмма действий

### 3.4 Разработка

Разработка была начата с использованием функции, которая предоставляет возможность для генерации общего симметричного ключа.

Рисунок 3.4.1 демонстрирует два метода, `generatePrivateKey() -> String` и `generatePublicKey(privateKey: String) throws -> String`. Данные функции осуществляют генерацию открытого и закрытого ключа. В этих функциях используется протокол диффи-хельмана на эллиптических кривых. Используя функцию `p256`, которая представляет собой эллиптическую кривую, алгоритм генерирует открытый и закрытый ключ. Стоит сказать, что открытый ключ генерируется на основе закрытого ключа. Данные функции возвращают ключи с типом `string`. Отметим, что доступ к эллиптической кривой осуществляется путём импортирования стандартной библиотеки `cryptokit`.

Второстепенные функции с префиксом `import`, позволяют преобразовать ключи из формата `string` в тип, который используется для генерации ключей. Второстепенные функции с префиксом `export`, позволяют преобразовать ключи из формата, который используется для генерации ключей, в тип `string`. Более подробно с каждой функцией возможно ознакомиться в приложении работы.

```
// Генерация случайного закрытого ключа
func generatePrivateKey() -> String {
    ... let privateKey = P256.KeyAgreement.PrivateKey()
    ... return exportPrivateKey(privateKey)
}

// Генерация открытого ключа на основе закрытого ключа
func generatePublicKey(privateKey: String) throws -> String {
    ... let privateKeyObject = try importPrivateKey(privateKey)
    ... let publicKeyObject = privateKeyObject.publicKey
    ... return exportPublicKey(publicKeyObject)
}
```

Рисунок 3.4.1 – Алгоритм генерации общего ключа

Далее была создана функция для генерации ключа `deriveSymmetricKey`. Данная функция представлена на рисунке 3.4.2. Генерации происходит посредством применения алгоритма `sha256` и соли “salt for key”, для сложности алгоритма, значение в 32 показывает, что генерируется ключ с размером 32 байта.

```
// Вычисление симметричного ключа на основе закрытого и открытого ключей
func deriveSymmetricKey(privateKey: String, publicKey: String) throws -> String {
    let privateKeyObject = try! importPrivateKey(privateKey)
    let publicKeyObject = try! importPublicKey(publicKey)
    let sharedSecret = try! privateKeyObject.sharedSecretFromKeyAgreement(with: publicKeyObject)
    let symmetricKey = sharedSecret.hkdfDerivedSymmetricKey(
        using: SHA256.self,
        salt: "salt for key".data(using: .utf8)!,
        sharedInfo: Data(),
        outputByteCount: 32
    )
    return exportSymmetricKey(symmetricKey)
}
```

Рисунок 3.4.2 – Алгоритм генерации общего ключа

Были проведены тесты для генерации общих ключей, смотрите рисунки 3.4.3 и 3.4.4. Как видно из рисунков, существовали две точки, А и В, каждый сгенерировал общий ключ и консоли был выведен результат сравнения общих ключей, смотрите рисунков 3.4.4. На данном этапе разработка для генерации общего ключа завершена.

```
let privateKeyA = generatePrivateKey()
let publicKeyA = try! generatePublicKey(privateKey: privateKeyA)
// Генерация ключей для участника В
let privateKeyB = generatePrivateKey()
let publicKeyB = try! generatePublicKey(privateKey: privateKeyB)
// Вычисление общего секретного ключа на стороне А
let symmetricKeyA = try! deriveSymmetricKey(privateKey: privateKeyA, publicKey: publicKeyB)
// Вычисление общего секретного ключа на стороне В
let symmetricKeyB = try! deriveSymmetricKey(privateKey: privateKeyB, publicKey: publicKeyA)
// Проверка совпадения общего секретного ключа на обеих сторонах
print(symmetricKeyA == symmetricKeyB)
```

Рисунок 3.4.3 – Тестирование

true

Рисунок 3.4.4 – Вывод в консоль

Далее был настроен сервис firebase. Перейдя на сайт: [console.firebase.google.com](https://console.firebase.google.com). Был создан новый проект, посредством нажатия “+ add new project” и изменения поля названия самого приложения. Далее в меню, рисунок которого вы можете наблюдать на изображении 3.4.5. Были выбраны поля authentication, storage, firestore database, в каждой вкладке была проведена авторизация и создана база данных, посредством нажатия кнопки “get started”. Стоит сказать, что база данных authentication была создана с использованием хранения почти и пароля.

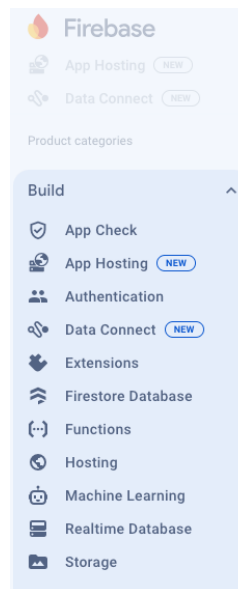


Рисунок 3.4.5 – Настройка firebase

Далее была настроена вкладка storage, где использовалась база данных, которая в тестовом режиме, то есть в открытом доступе для всех. Были созданы правила, которые ограничивали возможность вносить и читать данные, данный свод правил записан в виде кода на рисунке 3.4.6.

```
rules_version = '2';

service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

Рисунок 3.4.6 – Настройка storage

База данных firestore database была создана с использованием тестового режима, который позволяет предоставлять публичный доступ к базе данных. Свой правил, который был использован для этой базы данных, изображён на рисунке 3.4.7.

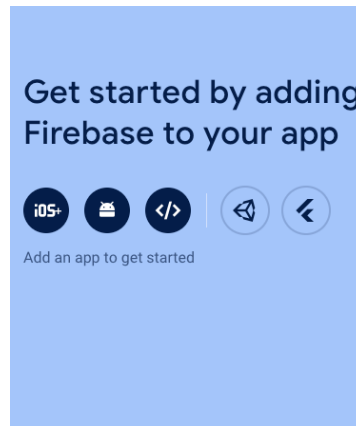
```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.time < timestamp.date(2024, 7, 6);
    }
  }
}
```

Рисунок 3.4.7 – Настройка firestore database

Далее на главной вкладке была выбрана функция “iOS+”, которая позволяет загрузить необходимые настройки проекта, чтобы обеспечивать

корректное соединение с базой данных firebase. Главная страница изображена на рисунке 3.4.8. После ввода всех необходимых значений, был скачен файл и загружен в мобильное приложение.



### Рисунок 3.4.8 – Главная вкладка проекта firebase

Далее происходил этап разработки. После открытия приложения происходит этап проверки на авторизацию пользователя в системе, если пользователь не авторизован, то происходит переход на страницу авторизации/регистрации. Рассмотрим код страницы авторизации. Используя архитектурный шаблон, было разработано представление, которые возможно наблюдать на рисунке 3.4.9. В данном разделе не будет описан синтаксис swiftui, но некоторые компоненты будут затронуты и описаны.

```

struct AuthView: View {
    @ObservedObject private var viewModel: AuthViewModel

    init(didCompleteLoginProcess: @escaping () -> ()) {
        self.viewModel = AuthViewModel(didCompleteLoginProcess: didCompleteLoginProcess)
    }

    var body: some View {
        NavigationView {
            ScrollView {
                VStack {
                    modePicker
                    if !viewModel.isLoginMode {
                        profileImagePicker
                    }
                    emailTextField
                    passwordTextField
                    actionButton
                    Text(viewModel.messageError)
                        .foregroundColor(Color.red)
                        .font(.system(size: 10))
                        .padding()
                }
            }.padding()
        }.navigationTitle(viewModel.isLoginMode ? "Log In" : "Create Account")
    }
}

```

Рисунок 3.4.9 – Код страницы авторизации/регистрации

На рисунке 3.4.9 возможно наблюдать `@ObservedObject viewModel`, представление наблюдает за всеми изменениями в определённых переменных

viewModel. В конструкторе инициализируется переменная viewModel. Переменная didCompleteLoginProcess является замыканием, то есть безымянной функцией, которая выполняется после вызова. Стоит отметить, что структура AuthView наследуется от структуры View, которая имеет настройки для отображения. Все структуры типа View имеют переменную body, которая хранит в себе продавленные и элементы, которые на этом представлении размещены. NavigationView является стеком, в который возможно складывать другие экраны-представления. Далее изображены графические элементы, с кодом которых вы можете ознакомиться в приложении проекта.

После был реализован класс FirebaseManeger, взаимодействие с которым даёт осуществлять соединение с базой данных firebase. Данный класс использует порождающий паттерн singleton, который создаёт только один экземпляр класса во всём проекте, обращение происходит посредством вызова FirebaseManager.shared. Более подробно с кодом вы можете ознакомиться в приложении проекта.

Далее был реализован файл AuthViewModel.swift, который является элементом mvvm. Данный класс имеет свойства с атрибутами @Published, изменения которых отслеживает представление. Опишем некоторые функции. Функция createNewAccount отвечает за создание нового пользователя. Используя созданный класс FirebaseManager, происходит вызов функции создания пользователя. В данной функции есть вызов другой функции sendImageToStorage, которая после успешного создания пользователя в системе загружает изображение в базу storage. После успешной загрузки изображения вызывается функция sendUserInformationToStoraga, которая создаёт пользователя в системе firestore. Данная функция изображена на рисунке 3.4.10. В этой функции происходит генерация пары ключей. Закрытый ключ помещается в кэше приложения, используя UserDefaults. Далее формируются массив свойств, которые имеет пользователь, смотрите переменную userData. В данном свойстве возможно наблюдать следующие

поля: email (почта), uid (идентификатор пользователя), profileImageUrl (url адресс изображения профиля), publicKey (публичный ключ). Далее эти данные отправляются в систему firestore. Здесь можно наблюдать свойство collection, который создаёт коллекцию, где будут размещены пользователи. Свойство document определяет сам документ, его название, в данном случае идентификатор пользователей является индексом его же данных в коллекции users. После успешного добавления происходит обратный вызов didCompleteLoginProcess. Вызов функций для создания пользователя является каскадным.

```
private func sendUserInfoToStorage(profileImageUrl: URL) {~
    let privateKey = generatePrivateKey()~
    UserDefaults.standard.set(privateKey, forKey: "userPrivateKey")~
    guard let publicKey = try? generatePublicKey(privateKey: privateKey) else {~
        print("Failed to generate public key")~
        self.messageError = "Failed to generate public key"~
        return~
    }~
    guard let uid = FirebaseManager.shared.auth.currentUser?.uid else { return }~
    let userData: [String : Any] = [~
        FirebaseConstants.uid: uid,~
        FirebaseConstants.email: email,~
        FirebaseConstants.profileImageUrl: profileImageUrl.absoluteString,~
        FirebaseConstants.publicKey: publicKey~
    ]~
    FirebaseManager.shared.firestore.collection("users").document(uid).setData(userData) { error in~
        if let error = error {~
            print("Failed to store user information: \(error)")~
            self.messageError = "\(error)"~
            return~
        }~
    }~
    self.didCompleteLoginProcess()~
}
```

Рисунок 3.4.10 – Код страницы авторизации/регистрации

Далее была реализована функция входа в систему loginUser. Данная функция помимо авторизации в системе, выполняет генерацию новой пары закрытого и открытого ключей, после генерации, новые ключи обновляют старые. В конце происходит обратный вызов didCompleteLoginProcess. Более подробно с кодом вы можете ознакомиться в приложении проекта. После разработки страниц авторизации/регистрации были достигнуты следующие результаты, которые вы можете наблюдать на рисунке 3.4.11. Более подробнее об демонстрации функционала и демонстрации тестов, вы можете ознакомиться в разделе результатов и тестирования.

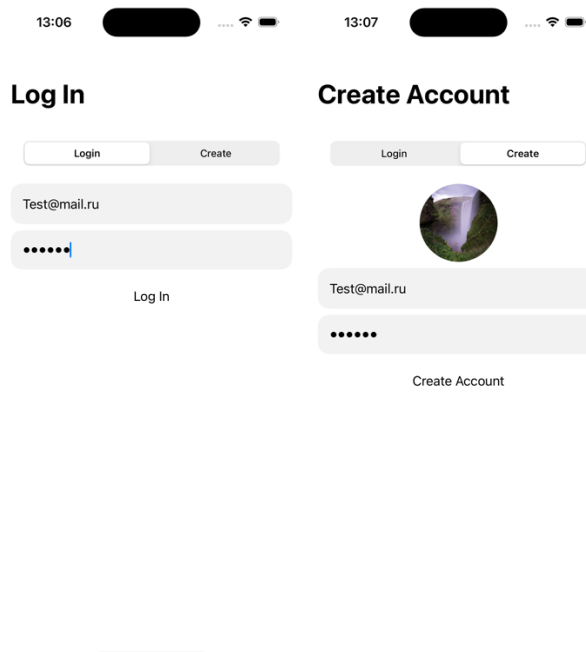


Рисунок 3.4.11 – Страница авторизации/регистрации

Далее была создана папка со всеми структурами, которые необходимы для дальнейшей работы приложения. Рассмотрим несколько из них.

На рисунке 3.4.12 изображена структура, которая описывает как текущего пользователя, так и его собеседника.

```
struct ChatUser: Identifiable {
    var id: String { uid }
    // Характеристики пользователя
    let uid, email, profileImageUrl, publicKey: String
    // Получение имени без @
    var username: String {
        email.components(separatedBy: "@").first ?? email
    }
    init(data: [String: Any]) {
        self.id = data[FirebaseConstants.uid] as? String ?? ""
        self.email = data[FirebaseConstants.email] as? String ?? ""
        self.profileImageUrl = data[FirebaseConstants.profileImageUrl] as? String ?? ""
        self.publicKey = data[FirebaseConstants.publicKey] as? String ?? ""
    }
}
```

Рисунок 3.4.12 – Структура ChatUser

На рисунке 3.4.13 изображена структура, которая описывает сообщения, которые будут отображены в активном чате.

```
struct ChatMessage: Identifiable {
    var id: String { documentId }
    let documentId: String
    // Характеристика письма
    let fromId, toId, text: String
    init(documentId: String, data: [String: Any]) {
        self.documentId = documentId
        self.fromId = data[FirebaseConstants.fromId] as? String ?? ""
        self.toId = data[FirebaseConstants.toId] as? String ?? ""
        self.text = data[FirebaseConstants.text] as? String ?? ""
    }
}
```

Рисунок 3.4.13 – Структура ChatMessage



На рисунке 3.4.14 изображена структура, которая описывает последние сообщения. В данной структуре можно выделить свойство `timeAgo`, которое вычисляет время, когда написано последнее сообщение.

```
// Структура для описания последнего сообщения~
struct LastMessage: Identifiable {~
    ~~~~~
    var id: String { documentId }~
    let documentId: String~
    let fromId, toId: String~
    let timestamp: Timestamp~
    let email, profileImageUrl: String~
    ~~~~~
    var timeAgo: String {~
        let date = Date(timeIntervalSince1970: TimeInterval(timestamp.seconds))~
        let formatter = RelativeDateTimeFormatter()~
        formatter.unitsStyle = .abbreviated~
        return formatter.localizedString(for: date, relativeTo: Date())~
    }~
    ~~~~~
    var username: String {~
        email.components(separatedBy: "@").first ?? email~
    }~
    ~~~~~
    init(documentId: String, data: [String: Any]) {~
        self.documentId = documentId~
        self.fromId = data[FirebaseConstants.fromId] as? String ?? ""~
        self.toId = data[FirebaseConstants.toId] as? String ?? ""~
        self.profileImageUrl = data[FirebaseConstants.profileImageUrl] as? String ?? ""~
        self.email = data[FirebaseConstants.email] as? String ?? ""~
        self.timestamp = data[FirebaseConstants.timestamp] as? Timestamp ?? Timestamp(date: Date())~
    }~
}
```

Рисунок 3.4.14 – Структура LastMessage

После авторизации на странице авторизации/регистрации пользователь попадает на главную страницу. Далее речь пойдёт о представлении главного экрана. Рассмотрим рисунок 3.4.15. В данной структуре можно наблюдать модель-представление с атрибутом `@ObservedObject`. Далее идут свойства с атрибутом `@State`, который даёт команду приложения отслеживать малейшие изменения этих переменных в самой структуре для корректного отображения страниц. [7] При нажатии на определённые кнопки эти переменные обновляются. Здесь есть переменные, отвечающие за, выход из системы, переход в активный чат, выбор нового собеседника, собеседник, с которым происходит на данный момент общение, переменная, отвечающая за появление информационного окна. Далее есть модель-представление активного чата, чтобы при переключении страниц, корректно передавался собеседник.

```
struct MainScreenView: View {~
    ~~~~~
    @ObservedObject private var viewModel = MainScreenViewModel()~
    ~~~~~
    @State private var shouldShowLogoutOption = false~
    @State private var shouldNavigateToChatLogView = false~
    @State private var shouldShowNewMessageScreen = false~
    @State private var chatUser: ChatUser?~
    @State private var showAlert = false~
    ~~~~~
    private var chatLogViewModel = ChatViewModel(chatUser: nil)~
}
```

Рисунок 3.4.15 – Код главной страницы

Далее описывать графические элементы и что они делают не представляется важным, поэтому с кодом вы можете ознакомиться в приложении данного проекта.

Далее был реализован файл `MainMessageViewModel.swift`, который является моделью-представлением для файла `MainScreenView.swift`. Класс `MainScreenViewModel` наследуется от `ObservableObject`, так как является наблюдаемым объектом для класса `MainScreenView`. Смотрите рисунок 3.4.16. На данном рисунке изображены такие свойства: `chatUser` (данное свойство хранит в себе данные о текущем пользователе), `recentMessages` (массив последних сообщений, которые будут отображены на главной странице), `isUserCurrentlyLoggedIn` (свойство которое проверяет, находится ли пользователь в системе или нет), `timer` (данное свойство необходимо для обновления представления, которое корректно отображает время изменения последних сообщений), `firestoreListener` (необходим для отслеживания изменений в базе данных). В конструкторе происходит инициализация текущего пользователя и получение последних сообщений.

```
class MainScreenViewModel: ObservableObject {  
    @Published var chatUser: ChatUser? =  
    @Published var recentMessages = [LastMessage]()  
    @Published var isUserCurrentlyLoggedIn = false  
    private var timer: Timer?  
    private var firestoreListener: ListenerRegistration?  
    init() {  
        self.isUserCurrentlyLoggedIn = FirebaseManager.shared.auth.currentUser?.uid == nil  
        getCurrentUser()  
        getLastMessages()  
    }  
}
```

Рисунок 3.4.16 – Код главной страницы

Далее рассмотрим функцию получения массива последних сообщений, смотрите на рисунок 3.4.17. В данной функции сначала происходит получение идентификатора текущего пользователя, затем идет прослушивание изменений в базе данных в таблице последних сообщений. Переменные `snapshot`, которые используются в запросе к базе данных, являются снимком текущего состояния системы, если данные новые, тогда в свойстве `documentChanges` появится новый элемент, затем происходит вставка в массив последних сообщений.

```

func getLastMessages() {
    guard let uid = FirebaseManager.shared.auth.currentUser?.uid else { return }
    firestoreListener?.remove()
    self.recentMessages.removeAll()
    firestoreListener = FirebaseManager.shared.firestore.collection(FirebaseConstants.recentMessages)
        .document(uid)
        .collection(FirebaseConstants.messages)
        .order(by: "timestamp")
        .addSnapshotListener { querySnapshot, error in
            if let error = error {
                print("Ошибка в загрузке последних сообщений: \(error)")
                return
            }
            querySnapshot?.documentChanges.forEach { change in
                let docId = change.document.documentID
                if let index = self.recentMessages.firstIndex(where: { $0.documentId == docId }) {
                    self.recentMessages.remove(at: index)
                }
                self.recentMessages.insert(.init(documentId: docId, data: change.document.data()), at: 0)
            }
        }
}

```

Рисунок 3.4.17 – Реализация метода getLastMessages

Ещё одной важной функцией является выход из системы, реализацию вы можете наблюдать на рисунке 3.4.18. В функции signOut, происходит остановка таймер обнуления представления в главной очереди. Затем происходит выход. Стоит отметить, что функция DispatchQueue.main.async выполняет код в главном потоке, это сделано для того, чтобы некоторый код не блокировался при переходе на страницу авторизации/регистрации. Более подробно с второстепенными методами класса MainScreenViewModel, вы можете ознакомиться в приложении данного проекта.

```

func signOut() {
    DispatchQueue.main.async {
        self.stopTimer()
    }
    isUserCurrentlyLoggedOut.toggle()
    try? FirebaseManager.shared.auth.signOut()
    firestoreListener?.remove()
}

```

Рисунок 3.4.18 – Реализация метода signOut

В результате проделанной работы был получен результат, который вы можете наблюдать на рисунке 3.4.19. В данном случае пользователь успешно зарегистрировал свой аккаунт и видит главную страницу.



Рисунок 3.4.19 – Главная страница

Далее была реализована возможность написания нового сообщения. Рассмотрим некоторый код, который есть в файле `NewMessageView.swift`, смотрите рисунок 3.4.20. На данном рисунке изображены следующие свойства: `didSelectNewUser` (данное замыкание вызывается в главном экране для получения выбранного пользователя), `presentationMode` (с атрибутом `@Environment` позволяет подключаться к стеку `NavigationView`, чтобы закрыть текущий экран), `viewModel` (модель-представление для данной страницы). Далее рассмотрения кода страницы не представляется важным, более подробно ознакомиться с кодом вы можете в приложении проекта.

```
struct NewMessageView: View {  
    ...  
    ...let didSelectNewUser: (ChatUser) -> ()  
    ...@Environment(\.presentationMode) var presentationMode  
    ...@ObservedObject var viewModel = NewMessageViewModel()  
}
```

Рисунок 3.4.20 – Код страницы выбора собеседника

После рассмотрим реализацию класса `NewMessageViewModel`, смотрите рисунок 3.4.21. На данном рисунке изображено свойство `users`, в котором записаны все возможные пользователи. Далее реализован метод `getAllUsers`, в котором происходит добавление пользователей в массив `users`.

```

class NewMessageViewModel: ObservableObject {
    @Published var users = [ChatUser]()
    init() {
        getAllUsers()
    }
    private func getAllUsers() {
        FirebaseManager.shared.firestore.collection("users").getDocuments { (weak self) documentsSnapshot, error in
            if let err = error {
                print("Ошибка с получением данных", err)
                return
            }
            documentsSnapshot?.documents.forEach { snapshot in
                let data = snapshot.data()
                let user = ChatUser(data: data)
                if user.uid != FirebaseManager.shared.auth.currentUser?.uid {
                    self?.users.append(user)
                }
            }
        }
    }
}

```

Рисунок 3.4.21 – Код страницы выбора собеседника

В ходе проделанной работы по реализации страницы выбора собеседника был сделан рисунок 3.4.22. Стоит пояснить, что существует уже в системе некий пользователь Test2, которому мы можем написать сообщение.

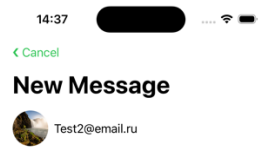


Рисунок 3.4.22 – Страница выбора собеседника

Далее будет рассмотрена часть, связанная со страницей написания сообщения собеседнику, смотрите на рисунок 3.4.23. На данном рисунке изображен класс ChatViewModel, который содержит следующие свойства: chatText (поле сообщения), count (счётчик который помогает представлению проматывать ленту переписки до низу), chatMessages (массив всех сообщений), chatUser (с кем происходит общение), firestoreListener (данное свойство замечает изменения в базе данных списка сообщений). В конструкторе мы получаем все сообщения и инициализируем собеседника.

```

class ChatViewModel: ObservableObject {
    ...
    ...@Published var chatText = ""...
    ...@Published var count = 0...
    ...@Published var chatMessages = [ChatMessage]()...
    ...
    ...var chatUser: ChatUser?...
    ...var firestoreListener: ListenerRegistration?...
    ...
    ...init(chatUser: ChatUser?) {...
    ...    ...self.chatUser = chatUser...
    ...    ...getMessages()...
    ...}...
}

```

Рисунок 3.4.23 – Код страницы чата

Представление кодов методов, которые выполняют действия, не представляется важным, поэтому подробнее с кодом вы можете ознакомиться в приложении. Опишем некоторые методы, которые используются в классе ChatViewModel. Метод `getMessages` инициализирует массив сообщений, каждый, и если возникают изменения в базе данных, он это фиксирует. Метод `processMessageChange` является связанным методом с `getMessages`, данным метод выполняет получение симметричного ключа и расшифрованное сообщений, которые находятся на сервере, затем добавляет расшифрованные сообщения в массив сообщений. Также существует метод `handleSend`, который шифрует данные и отправляет зашифрованное сообщение на сервер, посредством вызова метода `sendMessage` в коллекции “messages”. Также в методе `handleSend` происходит сохранение последнего сообщения в базе данных в коллекции “recent\_messages”, используя метод `saveLastMessage`. Все отправляемые данные на сервер шифруются общим ключом, что позволяет сделать данные на сервере не читаемыми. В данном классе также существует вспомогательные методы, в реализацию которых вы можете увидеть в приложении данного проекта.

Далее речь пойдёт о представлении чата, рассмотрим некоторый код, который важен, смотрите рисунок 3.4.24. На данном рисунке изображены свойства `vm` (модель-представление с атрибутом `@ObservedObject`), `presentationMode` (с атрибутом `@Environment`, который позволяет скрывать текущую страницу), далее идёт код самого представления в свойстве `body`. Описание графических элементов и какие действия они выполняют, опустим. С полным кодом файла `ChatView.swift` вы можете ознакомиться в приложении данного проекта.

```

struct ChatView: View {
    ...
    @ObservedObject var vm: ChatViewModel
    @Environment(\.presentationMode) var presentationMode
    ...
    var body: some View {
        VStack {
            ScrollView {
                ScrollViewReader { proxy in
                    VStack {

```

Рисунок 3.4.24 – Код страницы чата

На данном этапе разработка экрана чата завершена, полученный результат можно наблюдать на рисунке 3.4.25. На данном рисунке можно наблюдать, что мы находимся в активном чате, переписка с Test2.

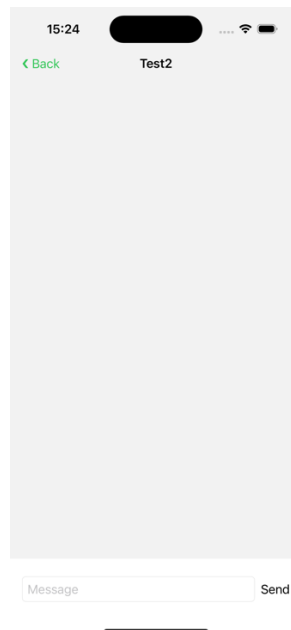


Рисунок 3.4.25 – Страница чата

На данном этапе разработка приложения завершена, в следующих разделах будет рассказано об результатах и тестировании приложения. С полным кодом приложения есть возможность ознакомиться в приложении данного проекта.

### 3.5 Результат

В ходе проделанной работы был реализован чат, который работает с базой данных firebase. Чат содержит уникальный потоковый шифр, который шифрует данные, чтобы никто кроме двух собеседников не смог прочитать их переписку. У пользователя есть возможно выбора собеседника, создание и вход в аккаунт. У пользователя есть изображение профиля, он также имеет

возможность в любой момент выйти из системы. Пользователь может осуществлять безопасную переписку.

### 3.6 Тестирование

В данном разделе речь пойдёт о ручном тестировании и проверки корректной работы мобильного приложения.

В странице логина, если будут введены некорректные данные, отобразится ошибка, которую вы можете наблюдать на рисунке 3.6.1. Данные ошибку пользователь получает от сервера, который обрабатывает полученные данные. Такой же вид ошибок происходит и при регистрации в системе. Существует множество ошибок, которые представляются некорректными в базе данных, перечислим несколько из них: отсутствие интернета, некорректный ввод, пользователь уже есть в системе, неверный пароль и так далее.

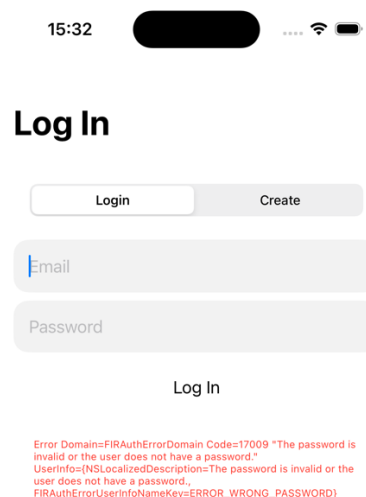


Рисунок 3.6.1 – Тестирование

Произведём регистрацию нового пользователя, предварительно очистим базу данных. После регистрации пользователя у нас отправывается главная



страница приложения. Далее будут приведены фото, которые показывают, что пользователь зарегистрирован в системе.

На рисунке 3.6.2 вы можете наблюдать зарегистрированного пользователя в базе authentication.

Identifier	Providers	Created ↓	Signed In	User UID
test@email...		Jun 13, ...	Jun 13, ...	IM0gBuP1HgaJ...

Rows per page: 50 1 - 1 of 1

Рисунок 3.6.2 – Тестирование

На рисунке 3.6.3 вы можете наблюдать зарегистрированную картинку профиля пользователя в базе storage.


<input type="checkbox"/>	Name	Size	Type	Last modified
<input checked="" type="checkbox"/>	 IM0gBuP1HgaJYpdryeg9To6Psp62	1.42 MB	application/octet-stream	Jun 13, 2024

Рисунок 3.6.3 – Тестирование

На рисунке 3.6.4 вы можете наблюдать данные пользователя, которые появились в базе firestore database.

users	>	IM0gBuP1HgaJYpdryeg9To6P...	>	+ Add field
				email: "Test@email.ru"
				profileImageUrl: "https://firebasestorage.googleapis.com/443/v... firebase.appspot.com/o/IM0gBuP1HgaJYpdry... alt=media&token=c9451872-068f-4cb7- a38a-5bf9eaa88f12"
				publicKey: "hfZV8lbrJQYDmO60V4oqWZi32PyTqofAsYMPkOjPzlv"
				uid: "IM0gBuP1HgaJYpdryeg9To6Psp62"

Рисунок 3.6.4 – Тестирование

На рисунке 3.6.5 изображено информационное окно.

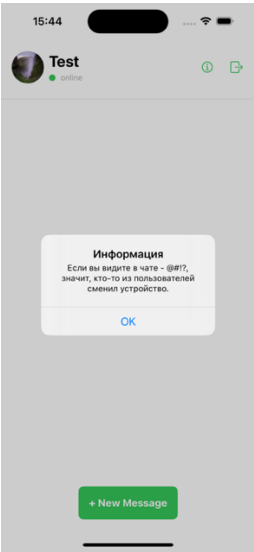


Рисунок 3.6.5 – Тестирование

Проверку выхода из системы и выбора собеседника мы опустим, так как эти тесты не представляются важными и были проведены в реализации приложения.

Далее проведём регистрацию нового пользователя test2 и проверим переписку пользователя test и test2.

Провели регистрацию пользователя test2 на новом устройстве и ввели новые сообщения от каждого пользователя, результат вы можете наблюдать на рисунке 3.6.6.

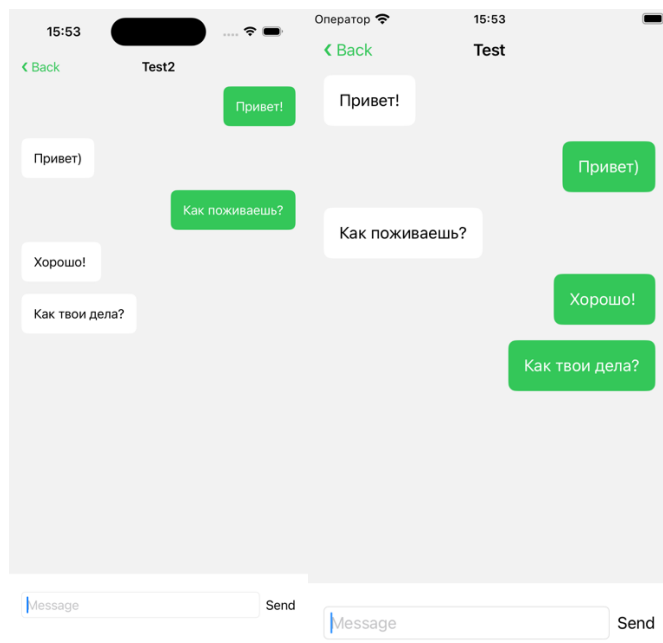


Рисунок 3.6.6 – Тестирование

Далее будут приведены рисунки, которые показывают, что база данных изменилась. На рисунке 3.6.7 вы можете наблюдать, что было создано две коллекции обычных сообщений и последних сообщений.

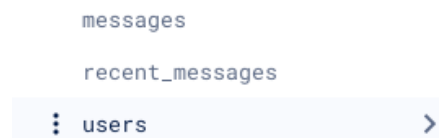


Рисунок 3.6.7 – Тестирование

На рисунке 3.6.8 вы можете наблюдать данные самого сообщения, в данном случае это последнее сообщение из списка сообщений. Вы можете наблюдать, что поле text представляет собой массив нечитаемых данных.

```
fromId: "IM0gBuP1HgaJYpdryeg9To6PSP62"
text: [107, 36, 153, 223, 3, 90,...]
timestamp: June 13, 2024 at 3:53:16 PM UTC+3
toId: "zFJPOPLLknaDb6QHZwOZMRwyLH33"
```

### Рисунок 3.6.8 – Тестирование

На рисунке 3.6.9 изображены данные последнего сообщения, на данном рисунке видно, что данные не читаемы.

```
email: "Test2@email.ru"
fromId: "zFJPOPLLknaDb6QHZwOZMRwyLH33"
profileImageUrl: "https://firebasestorage.googleapis.com/443/v0/b/swissmer-
firebase.appspot.com/o/zFJPOPLLknaDb6QHZwOZMRwyLH33?
alt=media&token=2170f3fc-a4b0-4b2b-a8f1-de1f9f1ceac7"
text: [107, 36, 153, 223, 3, 90,...]
timestamp: June 13, 2024 at 3:53:26 PM UTC+3
toId: "IM0gBuP1HgaJYpdryeg9To6PSP62"
```

### Рисунок 3.6.9 – Тестирование

Далее были проведены тесты, когда один из пользователей меняет устройство, то есть заходит с нового девайса. Данные, изображённые на рисунке 3.6.10, было до входа с нового устройства. А на рисунке 3.6.11, когда пользователь test2 зашёл с нового устройства. Заметим, что поле publicKey поменялось, следовательно, пользователь test2 не сможет видеть предыдущую переписку с test.

```
email: "Test2@email.ru"
profileImageUrl: "https://firebasestorage.googleapis.com/443/v0/b/swissmer-
firebase.appspot.com/o/zFJPOPLLknaDb6QHZwOZMRwyLH33?
alt=media&token=2170f3fc-a4b0-4b2b-a8f1-de1f9f1ceac7"
publicKey: "xEDKE6KP7HcbStZMMV9JT0zrpMLog4SnF9viD1%2FDePchsNRpTKc"
uid: "zFJPOPLLknaDb6QHZwOZMRwyLH33"
```

### Рисунок 3.6.10 – Тестирование

```
email: "Test2@email.ru"
profileImageUrl: "https://firebasestorage.googleapis.com/443/v0/b/swissmer-
firebase.appspot.com/o/zFJPOPLLknaDb6QHZwOZMRwyLH33?
alt=media&token=2170f3fc-a4b0-4b2b-a8f1-de1f9f1ceac7"
publicKey: "0cqBUK80DCvXDDuUgnDMhdvPEFJVrHMLCTntQLdE8MBFMeWsSw"
uid: "zFJPOPLLknaDb6QHZwOZMRwyLH33"
```

### Рисунок 3.6.11 – Тестирование

Ситуация, при которой пользователь test2 изменил устройства видна на рисунке 3.6.12. Отметим особенности, пользователь test как видел все сообщения, так и видит все сообщения и пользователя с нового устройства,

пока не повторит вход в чат. Пользователь test2 не видит предыдущую переписку, так как ключи шифрования были изменены.

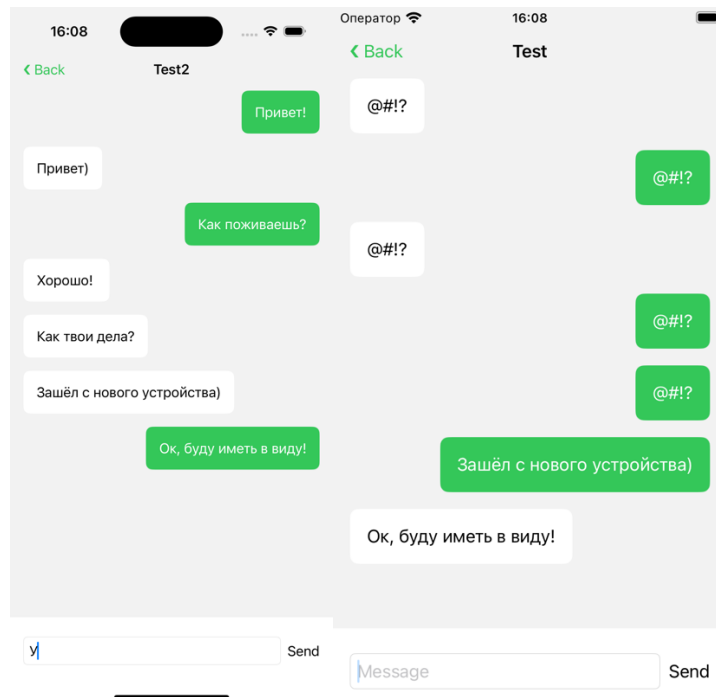


Рисунок 3.6.12 – Тестирование

Стоит сказать, что приложение работает корректно на всех краевых случаях и демонстрирует функциональность, обработка данных, которые поступают на сервер, происходит на самом сервере. Данные шифруются.

## ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы были проделаны следующие шаги: разработка шифра, проведение исследований по криптостойкости шифра, разработка сервиса для обмена данными между двумя мобильными устройствами, внедрение шифрования в разработанный сервис, тестирование и анализ результатов работы сервиса. По итогу, было разработан чат для конфиденциального общения, который использует протокол Диффи-Хеллмана для общения мобильных устройств и потоковый алгоритм шифрования. Таким образом, цель можно считать достигнутой. Функционал конечного приложения включает в себя: возможность зарегистрировать новый аккаунт, пройти авторизацию с помощью сервиса Firebase, обеспечение безопасного общения, посредством исключительно двух устройств, не давая 3-м лицам, знающим пароль или предоставляющим базу данных для хранения, прочесть переписку, возможность выбора собеседника, предварительный просмотр последнего сообщения, информационное сообщение для осведомления о приложении.

Мобильное приложение использует шаблон проектирования MVVM, поэтому была разделена логика на модель, представление и модель-представление. Для хранения данных пользователей и чата использовался сервис Firebase. Приложение написано на языке Swift, используя фреймворк SwiftUI. Разработан уникальный потоковый шифр, который обеспечивает надлежащую безопасность. Поставленные цели были достигнуты.

В таблице 1 указаны приобретенные в ходе работы компетенции:

Таблица 1 - Освоенные компетенции

Компетенция и расшифровка	Полученные результаты
УК-1 Способен осуществлять поиск, критический анализ и синтез информации, применять системный подход для решения поставленных задач	Изучил теоретический материалы по криптографии и криптоанализу. Изучил разработку мобильно приложения, Swift, SwiftUI, Firebase.
УК-2 Способен определять круг задач в рамках поставленной цели и выбирать оптимальные способы их решения, исходя из действующих правовых норм, имеющихся ресурсов и ограничений	Для достижения результатов поставленная цель была разбита на несколько задач, что позволило оптимизировать технологические ресурсы.
УК-3 Способен осуществлять социальное взаимодействие и реализовывать свою роль в команде	Получен опыт взаимодействия с научно-преподавательским составом.
УК-4 Способен осуществлять деловую коммуникацию в устной и письменной формах на государственном языке Российской Федерации и иностранном(ых) языке(ах)	В процессе взаимодействия с преподавателями применял навыки деловой письменной и устной коммуникации.
УК-5 Способен воспринимать межкультурное разнообразие общества в социально-историческом, этическом и философском контекстах	Студент легко адаптируется к различным обществам во всех их аспектах и контекстах.
УК-6 Способен управлять своим временем, выстраивать и реализовывать траекторию саморазвития на основе принципов образования в течение всей жизни	Перед началом выполнения работы был разработан план, распределяющий время на каждый этап изучения и разработки. Все задачи были выполнены в сроки.

Продолжение таблицы 1

УК-7 Способен поддерживать должный уровень физической подготовленности для обеспечения полноценной социальной и профессиональной деятельности	Придерживался регулярной физической активности для поддержания высокой интеллектуальной работоспособности.
УК-8 Способен создавать и поддерживать безопасные условия жизнедеятельности, в том числе при возникновении чрезвычайных ситуаций	Получены навыки создания и поддержания необходимых условий для безопасной жизнедеятельности, в том числе и в чрезвычайных ситуациях.
ОПК-1 Способен применять фундаментальные знания, полученные в области математических и (или) естественных наук, и использовать их в профессиональной деятельности	Изучены и применены на практике языки и алгоритмы шифрования для создания мобильного приложения.
ОПК-2 Способен использовать и адаптировать существующие математические методы и системы программирования для разработки и реализации алгоритмов решения прикладных задач	Были изучены методы, которые решают поставленные математические задачи и проблемы программного обеспечения.
ОПК-3 Способен применять и модифицировать математические модели для решения задач в области профессиональной деятельности	В ходе всей работы использовались передовые методы для создания алгоритма шифрования.
ОПК-4 Способен решать задачи профессиональной деятельности с использованием существующих информационно-коммуникационных технологий и с учетом основных требований информационной безопасности	Были изучены и применены передовые методы в создании и проектировании мобильного приложения.

Продолжение таблицы 1

ПК-1 Проверка работоспособности и рефакторинг кода программного обеспечения, интеграция программных модулей и компонент и верификация выпусков программного обеспечения	Проведен неоднократный рефакторинг и тестирование, как программного кода, так и метода шифрования
ПК-2 Мониторинг функционирования интеграционного решения в соответствии с трудовым заданием, работа обращениями пользователей по вопросам функционирования интеграционного решения в соответствии с трудовым заданием	Получен опыт по выявлению ошибок и мониторингу как мобильного приложения, так и алгоритма шифрования, посредством замечаний пользователей.
ПК-3 Проверка и отладка программного кода, тестирование информационных ресурсов с точки зрения логической целостности (корректность ссылок, работа элементов форм)	Были проведены многократные проверки кода на его функциональность и логические операции. Проведены проверки корректности данных и ссылок.
ПК-4 Ведение информационных баз данных	Изучены методы и способы взаимодействия с базой данных.
ПК-5 Обеспечение функционирования баз данных	Мобильное приложение использует сервис Firebase для хранения данных о чате и о пользователях.
ПК-6 Педагогическая деятельность по проектированию и реализации общеобразовательных программ	Для глубоко понимания темы были проведены консультации с преподавателями, которые дали понимание предметной области.



Продолжение таблицы 1

ПК-7 Разработка и документирование программных интерфейсов, разработка процедур сборки модулей и компонент программного обеспечения, разработка процедур развертывания и обновления программного обеспечения	Использование передовых библиотек и фреймворков для корректной работы приложения, подключение проекта к Git.
ПК-8 Применять методы и средства сборки модулей и компонент программного обеспечения, разработки процедур для развертывания программного обеспечения, миграции и преобразования данных, создания программных интерфейсов	Изучен язык Swift и фреймворк SwiftUI для создания пользовательского интерфейса.
ПК-9 Описание возможной архитектуры развертывания каждого компонента, включая оценку современного состояния предлагаемых архитектур, оценка архитектур с точки зрения надежности правовой поддержки	Был изучены архитектурные паттерны проектирования приложений. Разработка мобильного приложения проходила с использованием паттерна MVVM.
ПК-10 Документальное предоставление прослеживаемости требований, согласованности с системными требованиями; приспособленность стандартов и методов проектирования; осуществимость, функционирования и сопровождения; осуществимость программных составных частей	Разработка приложения проходила с использованием стандартных требований к аппаратной части и интерфейса самого приложения. Проводилось активное функциональное сопровождение каждого компонента мобильного приложения.
ПК-11 Техническое сопровождение возможных вариантов архитектуры компонентов, включающее описание вариантов и технико-экономическое обоснование выбранного варианта	В ходе разработки мобильного приложения был выбран шаблон проектирования MVVM. Все необходимые компоненты паттерна MVVM реализованы в полном объеме.

Продолжение таблицы 1

ПК-12 Выполнение работ по созданию (модификации) и сопровождению ИС, автоматизирующих задачи организационного управления и бизнес-процессы	Получен опыт создания, использования существующих технологий, которые обеспечивали корректность работы мобильного приложения
ПК-13 Создание и сопровождение требований и технических заданий на разработку и модернизацию систем и подсистем малого и среднего масштаба и сложности	Были получен опыт в составлении технологических требований в разработки. Документирование позволило придерживаться плана реализации мобильного приложения и шифра в установленные сроки.
ПК-14 Способность использовать основы экономических знаний в профессиональной деятельности	Изучены передовые экономические и социальные методы, которые использовались для проектирования шифра и мобильного приложения.
ПК-15 Способность к коммуникации, восприятию информации, умение логически верно, аргументировано и ясно строить устную и письменную речь на русском языке для решения задач профессиональной коммуникации	Получены навыки воспринимать сложную информацию, строить коммуникацию, а также логически аргументировать свою точку зрения, используя грамотную устную и письменную речь.
ПК-16 Способность находить организационно-управленческие решения в нестандартных ситуациях и готовность нести за них ответственность	Получены навыки работы в стрессовых ситуациях. Изучены методы решения поставленных задач в нестандартных ситуациях. Получен навык ответственности.

Продолжение таблицы 1

<p>ПК-17 Знания своих прав и обязанностей как гражданина своей страны, способностью использовать действующее законодательство и другие правовые документы в своей профессиональной деятельности, демонстрировать готовность и стремление к совершенствованию и развитию общества на принципах гуманизма, свободы и демократии</p>	<p>Ознакомлен со всеми обязанностями и правами гражданина Российской Федерации.</p>
---	---

## СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Э. Таненбаум, Д. Уэзеролл, Н. Фимстер. Компьютерные сети. 6-е изд. – СПб.: Питер, 2023 г. – 992 с.
2. Б. Шнайер. Прикладная криптография. Протоколы, алгоритмы и исходные коды на языке С. – Диалектика-Вильямс 2019 г. – 1040 с.
3. С. Панасенко. Алгоритмы шифрования. Специальный справочник. – СПб.: БВХ-Петербург, 2009 г. – 576 с.
4. М. Масааки, С. Синъити, Х. Идэро. Занимательная информатика. Криптография. – ДМК-Пресс 2019 г. – 238 с.
5. Д. Кан. Взломщики кодов. – Центрполиграф 2000 г. – 1164 с.
6. Документация Swift [Электронный ресурс]. – 2014 – URL: <https://www.swift.org> (Дата обращения 5.05.24).
7. Документация SwiftUI [Электронный ресурс]. – 2019 – URL: <https://developer.apple.com/documentation/swiftui/> (Дата обращения 20.05.24)

## ПРИЛОЖЕНИЕ

### KeyGenerator.swift

```
import Foundation
import CryptoKit
// Генерация случайного закрытого ключа
func generatePrivateKey() -> String {
    let privateKey = P256.KeyAgreement.PrivateKey()
    return exportPrivateKey(privateKey)
}
// Генерация открытого ключа на основе закрытого ключа
func generatePublicKey(privateKey: String) throws -> String
{
    let privateKeyObject = try importPrivateKey(privateKey)
    let publicKeyObject = privateKeyObject.publicKey
    return exportPublicKey(publicKeyObject)
}
// Экспорт закрытого ключа в строку
func exportPrivateKey(_ privateKey:
P256.KeyAgreement.PrivateKey) -> String {
    let rawPrivateKey = privateKey.rawRepresentation
    let privateKeyBase64 =
rawPrivateKey.base64EncodedString()
    let percentEncodedPrivateKey =
privateKeyBase64.addingPercentEncoding(withAllowedCharacters:
.alphanumerics)!
    return percentEncodedPrivateKey
}
// Экспорт открытого ключа в строку
func exportPublicKey(_ publicKey:
P256.KeyAgreement.PublicKey) -> String {
    let rawPublicKey = publicKey.rawRepresentation
    let publicKeyBase64 = rawPublicKey.base64EncodedString()
    let percentEncodedPublicKey =
publicKeyBase64.addingPercentEncoding(withAllowedCharacters:
.alphanumerics)!
    return percentEncodedPublicKey
}
// Экспорт симметричного ключа в строку
func exportSymmetricKey(_ symmetricKey: SymmetricKey) ->
String {
    let rawSymmetricKey = symmetricKey.withUnsafeBytes {
Data(Array($0)) }
    return rawSymmetricKey.base64EncodedString()
}
// Импорт закрытого ключа из строки
func importPrivateKey(_ privateKey: String) throws ->
P256.KeyAgreement.PrivateKey {
    let privateKeyBase64 =
privateKey.removingPercentEncoding!
    let rawPrivateKey = Data(base64Encoded:
privateKeyBase64)!
```

```

        return try
P256.KeyAgreement.PrivateKey(rawRepresentation: rawPrivateKey)
    }
    // Импорт открытого ключа из строки
    func importPublicKey(_ publicKey: String) throws ->
P256.KeyAgreement.PublicKey {
        let publicKeyBase64 = publicKey.removingPercentEncoding!
        let rawPublicKey = Data(base64Encoded: publicKeyBase64)!
        return try
P256.KeyAgreement.PublicKey(rawRepresentation: rawPublicKey)
    }
    // Импорт симметричного ключа из строки
    func importSymmetricKey(_ keyString: String) -> SymmetricKey
{
        let rawSymmetricKey = Data(base64Encoded: keyString)!
        return SymmetricKey(data: rawSymmetricKey)
    }
    // Вычисление симметричного ключа на основе закрытого и
открытого ключей
    func deriveSymmetricKey(privateKey: String, publicKey:
String) throws -> String {
        let privateKeyObject = try importPrivateKey(privateKey)
        let publicKeyObject = try importPublicKey(publicKey)
        let sharedSecret = try
privateKeyObject.sharedSecretFromKeyAgreement(with:
publicKeyObject)
        let symmetricKey = sharedSecret.hkdfDerivedSymmetricKey(
            using: SHA256.self,
            salt: "salt for key".data(using: .utf8)!,
            sharedInfo: Data(),
            outputByteCount: 32
        )
        return exportSymmetricKey(symmetricKey)
    }
}

```

### Cipher.swift

```

import Foundation
class Cipher {
    // Паттерн Singleton
    static var shared = Cipher()
    private init() {}
    var SBlock: [Int] = Array(repeating: 0, count: 256) //
S-block
    // Шифрование & Расшифрование
    func encryptDecrypt(data: [UInt8], key: [UInt8]) ->
[UInt8] {
        var outputBytes: [UInt8] = []
        generationSBlock(key: key)
        var indexForSwapKey1 = 0
        var indexForSwapKey2 = 0
        // шифрование данных
        for index in 0..

```

```

        indexForSwapKey1 = index
        outputBytes.append(data[index] ^
getKey(indexForSwapKey1: indexForSwapKey1, indexForSwapKey2:
&indexForSwapKey2))
    }
    return outputBytes
}
// генерация S-блока
private func generationSBlock(key: [UInt8]) {
    for i in 0..

```

### ChatUser.swift

```

import Foundation
struct ChatUser: Identifiable {
    var id: String { uid }
    // Характеристики пользователя
    let uid, email, profileImageUrl, publicKey: String
    // Получение имени без @
    var username: String {
        email.components(separatedBy: "@").first ?? email
    }
    init(data: [String: Any]) {

```

```

        self.uid = data[FirebaseConstants.uid] as? String ??
""
        self.email = data[FirebaseConstants.email] as?
String ?? ""
        self.profileImageUrl =
data[FirebaseConstants.profileImageUrl] as? String ?? ""
        self.publicKey = data[FirebaseConstants.publicKey]
as? String ?? ""
    }
}

```

### FirestoreConstants.swift

```

struct FirestoreConstants {
    static let uid = "uid"
    static let fromId = "fromId"
    static let toId = "toId"
    static let text = "text"
    static let timestamp = "timestamp"
    static let profileImageUrl = "profileImageUrl"
    static let email = "email"
    static let recentMessages = "recent_messages"
    static let messages = "messages"
    static let publicKey = "publicKey"
}

```

### ChatMessage.swift

```

import FirebaseFirestore
struct ChatMessage: Identifiable {
    var id: String { documentId }
    let documentId: String
    // Характеристика письма
    let fromId, toId, text: String
    init(documentId: String, data: [String: Any]) {
        self.documentId = documentId
        self.fromId = data[FirebaseConstants.fromId] as?
String ?? ""
        self.toId = data[FirebaseConstants.toId] as? String
?? ""
        self.text = data[FirebaseConstants.text] as? String
?? ""
    }
}

```

### LastMessage.swift

```

import FirebaseFirestore
import Foundation
// Структура для описания последнего сообщения
struct LastMessage: Identifiable {
    var id: String { documentId }
    let documentId: String
    let fromId, toId: String
}

```



```

        let timestamp: Timestamp
        let email, profileImageUrl: String
        var timeAgo: String {
            let date = Date(timeIntervalSince1970:
TimeInterval(timestamp.seconds))
            let formatter = RelativeDateTimeFormatter()
            formatter.unitsStyle = .abbreviated
            return formatter.localizedString(for: date,
relativeTo: Date())
        }
        var username: String {
            email.components(separatedBy: "@").first ?? email
        }
        init(documentId: String, data: [String: Any]) {
            self.documentId = documentId
            self.fromId = data[FirebaseConstants.fromId] as?
String ?? ""
            self.toId = data[FirebaseConstants.toId] as? String
            self.profileImageUrl =
data[FirebaseConstants.profileImageUrl] as? String ?? ""
            self.email = data[FirebaseConstants.email] as?
String ?? ""
            self.timestamp = data[FirebaseConstants.timestamp]
as? Timestamp ?? Timestamp(date: Date())
        }
    }
}

```

### **FirestoreManager.swift**

```

import FirebaseFirestore
import FirebaseAuth
import FirebaseFirestore
import FirebaseFirestore
// Менеджер Firestore (Singleton) для взаимодействия с
Firestore
class FirestoreManager: NSObject {
    let auth: Auth
    let storage: Storage
    let firestore: Firestore
    var currentUser: ChatUser?
    static let shared = FirestoreManager()
    override init() {
        FirebaseFirestore.configure()
        self.auth = Auth.auth()
        self.storage = Storage.storage()
        self.firestore = Firestore.firestore()
        super.init()
    }
}

```

### **AuthView.swift**

```

import SwiftUI

```

```

import PhotosUI
struct AuthView: View {
    @ObservedObject private var viewModel: AuthViewModel
    init(didCompleteLoginProcess: @escaping () -> ()) {
        self.viewModel =
AuthViewModel(didCompleteLoginProcess: didCompleteLoginProcess)
    }
    var body: some View {
        NavigationView {
            ScrollView {
                VStack {
                    modePicker
                    if !viewModel.isLoginMode {
                        profileImagePicker
                    }
                    emailTextField
                    passwordTextField
                    actionButton
                    Text(viewModel.messageError)
                        .foregroundColor(Color.red)
                        .font(.system(size: 10))
                        .padding()
                }
                .padding()
            }
            .navigationTitle(viewModel.isLoginMode ? "Log
In" : "Create Account")
        }
        private var modePicker: some View {
            Picker(selection: $viewModel.isLoginMode, label:
Text("Mode")) {
                Text("Login").tag(true)
                Text("Create").tag(false)
            }
            .pickerStyle(SegmentedPickerStyle())
            .padding()
        }
        private var profileImagePicker: some View {
            Button(action: {}) {
                PhotosPicker(selection: $viewModel.selectedItem,
matching: .images) {
                    if let imageData =
viewModel.selectedImageData, let uiImage = UIImage(data:
imageData) {
                        Image(uiImage: uiImage)
                            .resizable()
                            .frame(width: 100, height: 100)
                            .clipShape(Circle())
                    } else {
                        Image(systemName: "person")
                            .font(.system(size: 64))
                            .padding()
                    }
                }
            }
        }
    }
}

```

```

        .foregroundColor(Color.black)
    }
}
.buttonStyle(PlainButtonStyle())
.onChange(of: viewModel.selectedItem) { newItem
in
    if let newItem = newItem {
        Task {
            if let data = try? await
newItem.loadTransferable(type: Data.self) {
                viewModel.selectedImageData =
data
            }
        }
    }
}
}
}
private var emailTextField: some View {
    TextField("Email", text: $viewModel.email)
        .foregroundColor(Color.black)
        .keyboardType(.emailAddress)
        .padding(15)
        .background(Color(.init(white: 0.95, alpha: 1)))
        .cornerRadius(15)
}
private var passwordTextField: some View {
    SecureField("Password", text: $viewModel.password)
        .foregroundColor(Color.black)
        .padding(15)
        .background(Color(.init(white: 0.95, alpha: 1)))
        .cornerRadius(15)
}
private var actionButton: some View {
    Button(action: viewModel.changeScreen) {
        Text(viewModel.isLoginMode ? "Log In" : "Create
Account")
        .foregroundColor(Color.black)
        .padding()
    }.buttonStyle(PlainButtonStyle())
}
}

```

### MainScreenView.swift

```

import SwiftUI
import SDWebImageSwiftUI
struct MainScreenView: View {
    @ObservedObject private var viewModel =
MainScreenViewModel()
    @State private var shouldShowLogoutOption = false
    @State private var shouldNavigateToChatLogView = false
    @State private var shouldShowNewMessageScreen = false

```

```

        @State private var chatUser: ChatUser?
        @State private var showAlert = false
        private var chatLogViewModel = ChatViewModel(chatUser:
nil)
        var body: some View {
            NavigationView {
                VStack {
                    userInfoView
                    Divider()
                    messagesListView
                    newMessageButton
                }
                .fullScreenCover(isPresented:
$viewModel.isUserCurrentlyLoggedOut, onDismiss: nil, content: {
                    AuthView(didCompleteLoginProcess: {
                        viewModel.startTimer()
                        viewModel.isUserCurrentlyLoggedOut =
false
                        viewModel.getCurrentUser()
                        viewModel.getLastMessages()
                    })
                })
                .onAppear {
                    if !$viewModel.isUserCurrentlyLoggedOut {
                        viewModel.startTimer()
                    }
                }
                .onDisappear {
                    viewModel.stopTimer()
                }
                .alert(isPresented: $showAlert) {
                    Alert(
                        title: Text("Информация"),
                        message: Text("Если вы видите в чате -
@#!?, значит, кто-то из пользователей сменил устройство."),
                        dismissButton: .default(Text("OK"))
                    )
                }
            }
        }
        private var userInfoView: some View {
            HStack {
                WebImage(url: URL(string:
$viewModel.chatUser?.profileImageUrl ?? ""))
                    .resizable()
                    .frame(width: 50, height: 50)
                    .clipShape(Circle())
                VStack(alignment: .leading, spacing: 4) {
                    Text($viewModel.chatUser?.username ?? "User")
                        .font(.system(size: 24, weight: .bold))
                    HStack {
                        Circle()
                            .foregroundColor(Color.green)

```

```

        .frame(width: 10, height: 10)
        Text("online")
        .font(.system(size: 12))
        .foregroundColor(Color(.lightGray))
    }
}
Spacer()
infoButton
logoutButton
}
.padding()
}
private var infoButton: some View {
    Button(action: {
        showAlert.toggle()
    }) {
        Image(systemName: "info.circle")
        .padding()
        .foregroundColor(.green)
    }
    .buttonStyle(PlainButtonStyle())
}
private var messagesListView: some View {
    ScrollView {
        ForEach(viewModel.recentMessages) {
recentMessage in
            VStack {
                Button {
                    handleSelectMessage(recentMessage:
recentMessage)
                } label: {
                    HStack(spacing: 16) {
                        WebImage(url: URL(string:
recentMessage.imageUrl))
                            .resizable()
                            .frame(width: 50, height:
50)
                            .clipShape(Circle())
                        Text(recentMessage.username)
                            .foregroundColor(.black)
                        Spacer()
                        Text(recentMessage.timeAgo)
                            .font(.system(size: 14,
weight: .semibold))
                            .foregroundColor(Color(.label))
                    }
                }
                .buttonStyle(PlainButtonStyle())
                Divider()
            }
        .padding(.horizontal)
    }
}

```

```

        }
        .background(
            NavigationLink("", isActive:
$shouldNavigateToChatLogView) {
                ChatView(vm: chatLogViewModel)
            }
            .hidden()
        )
    }
    private var newMessageButton: some View {
        Button(action: {
            shouldShowNewMessageScreen.toggle()
            DispatchQueue.main.async {
                self.viewModel.stopTimer()
            }
        }) {
            Text("+ New Message")
                .font(.system(size: 16, weight: .bold))
                .foregroundColor(.white)
                .padding()
                .background(Color.green)
                .cornerRadius(8)
        }
        .buttonStyle(PlainButtonStyle())
        .padding()
        .fullScreenCover(isPresented:
$shouldShowNewMessageScreen, onDismiss: nil, content: {
            NewMessageView(didSelectNewUser: { user in
                self.shouldNavigateToChatLogView = true
                self.chatUser = user
                self.chatLogViewModel.chatUser = user
                self.chatLogViewModel.getMessages()
            })
        })
    }
    private var logOutButton: some View {
        Button {
            shouldShowLogOutOption.toggle()
        } label: {
            Image(systemName:
"rectangle.portrait.and.arrow.forward")
                .foregroundColor(Color.green)
        }
        .buttonStyle(PlainButtonStyle())
        .actionSheet(isPresented: $shouldShowLogOutOption) {
            .init(title: Text("Settings"), message:
Text("What do you want to do?"), buttons: [
                .destructive(Text("Log Out"), action: {
                    viewModel.signOut()
                    print("Log out!")
                }),
                .cancel()
            ])
        }
    }
}

```

```

    }
    }
    private func handleSelectMessage(recentMessage:
LastMessage) {
        let uid =
FirebaseManager.shared.auth.currentUser?.uid ==
recentMessage.fromId ? recentMessage.toId : recentMessage.fromId

FirebaseManager.shared.firestore.collection("users").document(ui
d).getDocument { snapshot, error in
    if let error = error {
        print("Не удалось получить пользователя:
\\(error)")
        return
    }
    guard let data = snapshot?.data() else { return
}

    self.chatUser = .init(data: data)
    self.chatLogViewModel.chatUser = self.chatUser
    self.chatLogViewModel.getMessages()
    self.shouldNavigateToChatLogView = true
}
    }
}

```

### NewMessageView.swift

```

import SwiftUI
import SDWebImageSwiftUI
struct NewMessageView: View {
    let didSelectNewUser: (ChatUser) -> ()
    @Environment(\\.presentationMode) var presentationMode
    @ObservedObject var viewModel = NewMessageViewModel()
    var body: some View {
        NavigationView {
            ScrollView {
                ForEach(viewModel.users) { user in
                    Button(action: {
presentationMode.wrappedValue.dismiss()
                        didSelectNewUser(user)
                    }) {
                        UserRow(user: user)
                    }
                    .buttonStyle(PlainButtonStyle())
                    Divider()
                }
            }
            .navigationTitle("New Message")
            .toolbar {
                ToolbarItemGroup(placement:
.navigationBarLeading) {

```





```

        ))
    }
}
.background(Color(.init(white: 0.95, alpha: 1)))

messageInputView
}
.navigationTitle(vm.chatUser?.username ?? "")
.navigationBarTitleDisplayMode(.inline)
.navigationBarBackButtonHidden(true)
.toolbar {
    ToolbarItem(placement: .navigationBarLeading) {
        Button(action: {
            presentationMode.wrappedValue.dismiss()
        }) {
            Text("< Back")
                .foregroundColor(.green)
        }
    }
}
.onDisappear {
    vm.firestoreListener?.remove()
}
}
private var messageInputView: some View {
    HStack {
        TextField("Message", text: $vm.chatText)

.textFieldStyle(RoundedBorderTextFieldStyle())
        Button(action: vm.handleSend) {
            Text("Send").foregroundStyle(Color.black)
        }
    }
    .padding()
}
}
struct MessageView: View {
    let message: ChatMessage
    var body: some View {
        HStack {
            if message.fromId ==
FirebaseManager.shared.auth.currentUser?.uid {
                Spacer()
                HStack {
                    Text(message.text)
                        .foregroundColor(.white)
                        .padding()
                        .background(Color.green)
                        .cornerRadius(8)
                }
                .padding(.horizontal)
            } else {
                HStack {

```

```

        Text(message.text)
            .foregroundColor(.black)
            .padding()
            .background(Color.white)
            .cornerRadius(8)
    }
    .padding(.horizontal)
    Spacer()
}
}
.padding(.vertical, 4)
}
}

```

### AuthViewModel.swift

```

import SwiftUI
import PhotosUI
import Firebase
class AuthViewModel: ObservableObject {
    @Published var messageError = ""
    @Published var isLoginMode = true
    @Published var email = ""
    @Published var password = ""
    @Published var selectedItem: PhotosPickerItem? = nil
    @Published var selectedImageData: Data? = nil
    let didCompleteLoginProcess: () -> ()
    init(didCompleteLoginProcess: @escaping () -> ()) {
        self.didCompleteLoginProcess =
didCompleteLoginProcess
    }
    func changeScreen() {
        isLoginMode ? loginUser() : createNewAccount()
    }
    private func createNewAccount() {
        guard let selectedImageData = selectedImageData else
{
            print("Please select a profile image.")
            self.messageError = "Please select a profile
image."
            return
        }
        FirebaseManager.shared.auth.createUser(withEmail:
email, password: password) { result, error in
            if let error = error {
                print("Failed to create user: \(error)")
                self.messageError = "\(error)"
                return
            }

            self.sendImageToStorage(imageData:
selectedImageData)
        }
    }
}

```

```

    }
    private func loginUser() {
        FirebaseManager.shared.auth.signIn(withEmail: email,
password: password) { result, error in
            if let error = error {
                print("Failed to sign in user: \(error)")
                self.messageError = "\(error)"
                return
            }
            // Генерация ключей
            let privateKey = generatePrivateKey()
            UserDefaults.standard.set(privateKey, forKey:
"userPrivateKey")
            guard let publicKey = try?
generatePublicKey(privateKey: privateKey) else {
                print("Failed to generate public key")
                self.messageError = "Failed to generate
public key"
                return
            }
            guard let uid =
FirebaseManager.shared.auth.currentUser?.uid else { return }

            FirebaseManager.shared.firestore.collection("users").document(ui
d).getDocument { snapshot, error in
                if let error = error {
                    print("Не удалось получить пользователя:
\(error)")
                    self.messageError = "\(error)"
                    return
                }
                guard let data = snapshot?.data() else {
return }

                FirebaseManager.shared.currentUser =
.init(data: data)

                let userData: [String : Any] = [
                    FirebaseConstants.uid:
FirebaseManager.shared.currentUser?.uid as Any,
                    FirebaseConstants.email:
FirebaseManager.shared.currentUser?.email as Any,
                    FirebaseConstants.profileImageUrl:
FirebaseManager.shared.currentUser?.profileImageUrl as Any,
                    FirebaseConstants.publicKey: publicKey
                ]
                FirebaseManager.shared.firestore.collection("users").documen
t(uid).setData(userData) { error in
                    if let error = error {
                        print("Failed to store user
information: \(error)")
                        self.messageError = "\(error)"
                        return
                    }
                }
                self.didCompleteLoginProcess()
            }
        }
    }
}

```

```

        }
    }
}

private func sendImageToStorage(imageData: Data) {
    guard let uid =
FirebaseManager.shared.auth.currentUser?.uid else { return }
    let ref =
FirebaseManager.shared.storage.reference(withPath: uid)
    ref.putData(imageData, metadata: nil) { metadata,
error in
        if let error = error {
            print("Failed to upload image to storage:
\\(error)")
            self.messageError = "\\(error)"
            return
        }
        ref.downloadURL { url, error in
            if let error = error {
                print("Failed to retrieve download URL:
\\(error)")
                self.messageError = "\\(error)"
                return
            }
            guard let url = url else { return }

self.sendUserInformationToStorage(profileImageUrl: url)
        }
    }
}

private func
sendUserInformationToStorage(profileImageUrl: URL) {
    let privateKey = generatePrivateKey()
    UserDefaults.standard.set(privateKey, forKey:
"userPrivateKey")
    guard let publicKey = try?
generatePublicKey(privateKey: privateKey) else {
        print("Failed to generate public key")
        self.messageError = "Failed to generate public
key"
        return
    }
    guard let uid =
FirebaseManager.shared.auth.currentUser?.uid else { return }
    let userData: [String : Any] = [
        FirebaseConstants.uid: uid,
        FirebaseConstants.email: email,
        FirebaseConstants.profileImageUrl:
profileImageUrl.absoluteString,
        FirebaseConstants.publicKey: publicKey
    ]
}

```

```

FirebaseManager.shared.firestore.collection("users").document(ui
d).setData(userData) { error in
    if let error = error {
        print("Failed to store user information:
\\(error)")
        self.messageError = "\\(error)"
        return
    }
    self.didCompleteLoginProcess()
}
}
}

```

### MainMessageViewModel.swift

```

import SwiftUI
import SDWebImageSwiftUI
import FirebaseFirestore
class MainScreenViewModel: ObservableObject {
    @Published var errorMessage = ""
    @Published var chatUser: ChatUser?
    @Published var recentMessages = [LastMessage]()
    @Published var isUserCurrentlyLoggedOut = false
    private var timer: Timer?
    private var firestoreListener: ListenerRegistration?
    init() {
        self.isUserCurrentlyLoggedOut =
FirebaseManager.shared.auth.currentUser?.uid == nil
        getCurrentUser()
        getLastMessages()
    }
    deinit {
        firestoreListener?.remove()
        DispatchQueue.main.async {
            self.stopTimer()
        }
    }
    func startTimer() {
        timer = Timer.scheduledTimer(withTimeInterval: 15,
repeats: true) { _ in
            self.updateTimeAgo()
        }
    }
    func stopTimer() {
        timer?.invalidate()
        timer = nil
    }
    func updateTimeAgo() {
        self.recentMessages = self.recentMessages.map {
message in
            let updatedMessage = LastMessage(
                documentId: message.documentId,

```

```

        data: [
            "fromId": message.fromId,
            "toId": message.toId,
            "profileImageUrl":
message.profileImageUrl,
            "email": message.email,
            "timestamp": message.timestamp
        ]
    )
    return updatedMessage
}
}
func getLastMessages() {
    guard let uid =
FirebaseManager.shared.auth.currentUser?.uid else { return }
    firestoreListener?.remove()
    self.recentMessages.removeAll()
    firestoreListener =
FirebaseManager.shared.firestore.collection(FirebaseConstants.re
centMessages)
        .document(uid)
        .collection(FirebaseConstants.messages)
        .order(by: "timestamp")
        .addSnapshotListener { querySnapshot, error in
            if let error = error {
                print("Ошибка в загрузке последних
сообщений: \(error)")
            }
            return
        }
        querySnapshot?.documentChanges.forEach({
change in
            let docId = change.document.documentID

            if let index =
self.recentMessages.firstIndex(where: { $0.documentId == docId
}) {
                self.recentMessages.remove(at:
index)
            }

            self.recentMessages.insert(.init(documentId: docId, data:
change.document.data()), at: 0)
        })
    }
}
func getCurrentUser() {
    guard let uid =
FirebaseManager.shared.auth.currentUser?.uid else {
        print("Не удалось найти пользователя")
        return
    }
}

```

```

FirebaseManager.shared.firestore.collection("users").document(ui
d).getDocument { snapshot, error in
    if let error = error {
        print("Не удалось получить пользователя:
\\(error)")
        return
    }
    guard let data = snapshot?.data() else { return
}

    self.chatUser = .init(data: data)
    FirebaseManager.shared.currentUser =
self.chatUser
    }
}
func signOut() {
    DispatchQueue.main.async {
        self.stopTimer()
    }
    isUserCurrentlyLoggedOut.toggle()
    try? FirebaseManager.shared.auth.signOut()
    firestoreListener?.remove()
}
}

```

### NewMessageViewModel.swift

```

import SwiftUI
import FirebaseFirestore
class NewMessageViewModel: ObservableObject {
    @Published var users = [ChatUser]()
    init() {
        getAllUsers()
    }
    private func getAllUsers() {

FirebaseManager.shared.firestore.collection("users").getDocument
s { [weak self] documentsSnapshot, error in
    if let err = error {
        print("Ошибка с получением данных", err)
        return
    }
    documentsSnapshot?.documents.forEach({ snapshot
in
        let data = snapshot.data()
        let user = ChatUser(data: data)
        if user.uid !=
FirebaseManager.shared.auth.currentUser?.uid {
            self?.users.append(user)
        }
    })
}
}

```

```
}
```

## ChatViewModel.swift

```
import SwiftUI
import FirebaseFirestore
class ChatViewModel: ObservableObject {
    @Published var chatText = ""
    @Published var count = 0
    @Published var chatMessages = [ChatMessage]()
    var chatUser: ChatUser?
    var firestoreListener: ListenerRegistration?
    init(chatUser: ChatUser?) {
        self.chatUser = chatUser
        getMessages()
    }
    func getMessages() {
        guard let fromId =
FirebaseManager.shared.auth.currentUser?.uid,
            let toId = chatUser?.uid else { return }
        firestoreListener?.remove()
        chatMessages.removeAll()
        firestoreListener = FirebaseManager.shared.firestore
            .collection(FirebaseConstants.messages)
            .document(fromId)
            .collection(toId)
            .order(by: FirebaseConstants.timestamp)
            .addSnapshotListener { [weak self] snapshot,
error in
                guard let self = self else { return }
                if let error = error {
                    print("Failed to fetch messages:",
error)
                }
                return
            }
            snapshot?.documentChanges.forEach { change
in
                if change.type == .added {
                    self.processMessageChange(change)
                }
            }
        }
    }
    private func processMessageChange(_ change:
DocumentChange) {
        var dataChat = change.document.data()
        fetchChatUser { [weak self] result in
            guard let self = self else { return }
            switch result {
            case .success(let chatUser):
                self.chatUser = chatUser
                guard let privateKey =
UserDefaults.standard.string(forKey: "userPrivateKey") else {
```



```

        print("Не удалось получить приватный
ключ!")

        return
    }
    do {
        let keyString = try
deriveSymmetricKey(privateKey: privateKey, publicKey:
chatUser.publicKey)
        if let ciphertext = dataChat["text"] as?
[UInt8] {
            let decryptedText =
Cipher.shared.encryptDecrypt(data: ciphertext, key:
Array(keyString.utf8))
            dataChat["text"] =
Cipher.shared.stringFromBytes(decryptedText)
            DispatchQueue.main.async {

self.chatMessages.append(ChatMessage(documentId:
change.document.documentID, data: dataChat))
                self.count += 1
            }
        }
    } catch {
        print(error)
    }
    case .failure(let error):
        print("Failed to fetch user:", error)
    }
}

func handleSend() {
    guard !self.chatText.isEmpty,
        let fromId =
FirebaseManager.shared.auth.currentUser?.uid,
        let toId = chatUser?.uid else { return }
    fetchChatUser { [weak self] result in
        guard let self = self else { return }
        switch result {
        case .success(let chatUser):
            self.chatUser = chatUser
            guard let privateKey =
UserDefaults.standard.string(forKey: "userPrivateKey") else {
                print("Не удалось получить приватный
ключ!")
                return
            }
        }
        do {
            let keyString = try
deriveSymmetricKey(privateKey: privateKey, publicKey:
chatUser.publicKey)
            let ciphertext =
Cipher.shared.encryptDecrypt(data: Array(self.chatText.utf8),
key: Array(keyString.utf8))

```

```

        let messageData: [String: Any] = [
            FirebaseConstants.fromId: fromId,
            FirebaseConstants.toId: toId,
            FirebaseConstants.text: ciphertext,
            FirebaseConstants.timestamp:
Timestamp()
        ]
        self.sendMessage(messageData, fromId:
fromId, toId: toId)

        self.saveLastMessage(with: messageData)
        DispatchQueue.main.async {
            self.chatText = ""
            self.count += 1
        }
    } catch {
        print("Failed to send message:", error)
    }
    case .failure(let error):
        print("Failed to fetch user:", error)
    }
}

private func sendMessage(_ messageData: [String: Any],
fromId: String, toId: String) {
    let document = FirebaseManager.shared.firestore
        .collection(FirebaseConstants.messages)
        .document(fromId)
        .collection(toId)
        .document()
    document.setData(messageData) { error in
        if let error = error {
            print("Failed to send message:", error)
        } else {
            print("Message sent successfully.")
        }
    }
    let recipientDocument =
FirebaseManager.shared.firestore
        .collection(FirebaseConstants.messages)
        .document(toId)
        .collection(fromId)
        .document()
    recipientDocument.setData(messageData) { error in
        if let error = error {
            print("Failed to send message to
recipient:", error)
        } else {
            print("Message sent to recipient
successfully.")
        }
    }
}
}

```

```

        private func saveLastMessage(with messageData: [String:
Any]) {
            guard let chatUser = chatUser,
                  let uid =
FirebaseManager.shared.auth.currentUser?.uid,
                  let toId = self.chatUser?.uid else { return }
            var recentMessageData = messageData
            recentMessageData[FirebaseConstants.profileImageUrl]
= chatUser.profileImageUrl
            recentMessageData[FirebaseConstants.email] =
chatUser.email
            let recentMessageDocument =
FirebaseManager.shared.firestore
                .collection(FirebaseConstants.recentMessages)
                .document(uid)
                .collection(FirebaseConstants.messages)
                .document(toId)
            recentMessageDocument.setData(recentMessageData) {
error in
                if let error = error {
                    print("Failed to save recent message:",
error)
                }
            }
            guard let currentUser =
FirebaseManager.shared.currentUser else { return }
            var recipientRecentMessageData = messageData

            recipientRecentMessageData[FirebaseConstants.profileImageUrl] =
currentUser.profileImageUrl
            recipientRecentMessageData[FirebaseConstants.email]
= currentUser.email
            let recipientRecentMessageDocument =
FirebaseManager.shared.firestore
                .collection(FirebaseConstants.recentMessages)
                .document(toId)
                .collection(FirebaseConstants.messages)
                .document(currentUser.uid)

            recipientRecentMessageDocument.setData(recipientRecentMessageDat
a) { error in
                if let error = error {
                    print("Failed to save recipient recent
message:", error)
                }
            }
        }

        private func fetchChatUser(completion: @escaping
(Result<ChatUser, Error>) -> Void) {
            guard let chatUserId = chatUser?.uid else {
                completion(.failure(NSError(domain:
"ChatViewModel", code: 404, userInfo:
[NSLocalizedStringKey: "Chat user ID not found."])))
            }
        }
    }
}

```

```

        return
    }

    FirebaseManager.shared.firestore.collection("users").document(chatUserId).getDocument { snapshot, error in
        if let error = error {
            completion(.failure(error))
            return
        }
        guard let data = snapshot?.data() else {
            completion(.failure(NSError(domain:
"ChatViewModel", code: 404, userInfo:
[NSLocalizedStringKey: "User data not found."])))
            return
        }
        let chatUser = ChatUser(data: data)
        completion(.success(chatUser))
    }
}
deinit {
    firestoreListener?.remove()
}
}

```

### ChatAppApp.swift

```

import SwiftUI
@main
struct ChatAppApp: App {
    var body: some Scene {
        WindowGroup {
            MainScreenView()
                .preferredColorScheme(.light)
        }
    }
}

```