**1.Введение**

Современный мир немыслим без средств мгновенного обмена сообщениями. Мобильные мессенджеры прочно вошли в повседневную жизнь, став основным инструментом для личной и деловой коммуникации. Их популярность обусловлена удобством, скоростью и доступностью, позволяя пользователям оставаться на связи в любой точке мира. Наряду с глобальными платформами, такими как WhatsApp, Telegram и Signal, существует потребность в разработке специализированных или корпоративных мессенджеров, отвечающих уникальным требованиям безопасности, функциональности или интеграции с существующими системами. Актуальность данной курсовой работы обусловлена именно этой потребностью в кастомных коммуникационных решениях, а также постоянным развитием технологий мобильной разработки.

В рамках данной работы рассматривается задача разработки клиентского приложения для платформы Android. Это приложение предназначено для взаимодействия с существующим серверным компонентом мессенджера, функционирующим по протоколу TCP и использующим формат JSON для обмена данными. Сервер предоставляет базовый функционал, включающий регистрацию и аутентификацию пользователей, обмен текстовыми сообщениями в реальном времени и управление списком задач с уведомлениями. Клиентское приложение должно предоставить пользователю интуитивно понятный интерфейс для доступа ко всем этим функциям.

Целью настоящей курсовой работы является теоретическое обоснование выбора используемых технологий и инструментов, проектирование архитектуры и ключевых модулей клиентского Android-приложения, способного эффективно взаимодействовать с заданным TCP-сервером мессенджера.

Для достижения поставленной цели необходимо решить следующие задачи:

Провести анализ предметной области, рассмотрев функциональность и архитектурные подходы существующих мобильных мессенджеров.

Выполнить сравнительный анализ доступных платформ для мобильной разработки и обосновать выбор платформы Android в качестве целевой.

Сравнить основные подходы к разработке мобильных приложений (нативный и кроссплатформенный) и выбрать наиболее подходящий инструментарий, обосновав использование фреймворка Flutter.

Спроектировать архитектуру клиентского приложения, обеспечивающую его масштабируемость, тестируемость и простоту поддержки.

Исследовать специфику сетевого взаимодействия по протоколу TCP на мобильных устройствах и методы его реализации в выбранном фреймворке.

Проанализировать современные подходы к управлению состоянием приложения в рамках фреймворка Flutter и выбрать оптимальное решение.

Рассмотреть основополагающие принципы проектирования пользовательского интерфейса (UI) и взаимодействия (UX) для мобильных приложений.

Объектом исследования выступает процесс проектирования и разработки клиентской части мобильных приложений. Предметом исследования являются конкретные методы, технологии и инструменты, применяемые при разработке Android-приложений с использованием фреймворка Flutter для построения системы обмена сообщениями, взаимодействующей с сервером по протоколу TCP.

**2. Анализ Предметной Области и Существующих Решений**

Рынок мобильных мессенджеров характеризуется высокой конкуренцией и разнообразием предложений. Лидирующие приложения, такие как WhatsApp, Telegram, Signal, Viber, Facebook Messenger, предоставляют пользователям широкий спектр функций, выходящий далеко за рамки простого обмена текстовыми сообщениями. К стандартным возможностям относятся обмен мультимедийными файлами (фото, видео, аудио), голосовые и видеозвонки, создание групповых чатов, каналов, статусов, использование стикеров и GIF-анимаций, а также сквозное шифрование (end-to-end encryption) для обеспечения конфиденциальности переписки.

Архитектурно большинство современных мессенджеров используют клиент-серверную модель. Для обмена сообщениями и сигнализации часто применяются протоколы поверх TCP, такие как HTTPS (для запросов API), WebSockets (для постоянного соединения и real-time обновлений) или специализированные протоколы, например, MTProto в Telegram или XMPP (Extensible Messaging and Presence Protocol), который исторически был популярен, но сейчас реже используется в массовых продуктах из-за своей избыточности. Голосовые и видеозвонки часто реализуются с использованием протокола UDP и технологий Peer-to-Peer (P2P) для минимизации задержек, хотя сигнализация для установки соединения все равно проходит через сервер. Безопасность обеспечивается применением TLS/SSL для шифрования канала связи с сервером и, в некоторых случаях, сквозным шифрованием, когда ключи для расшифровки есть только у конечных пользователей.

Разрабатываемое в рамках данной работы приложение имеет существенные отличия от массовых аналогов. Во-первых, оно ориентировано на взаимодействие с конкретным, уже существующим серверным компонентом, использующим чистый TCP-протокол без стандартных оберток вроде HTTP или WebSockets, и обменивающимся данными в формате JSON по установленному сокет-соединению. Во-вторых, функциональность ограничена текстовым обменом сообщениями и управлением задачами, без поддержки мультимедиа, звонков или групповых чатов.

Эти отличия делают нецелесообразным или невозможным использование готовых библиотек или адаптацию существующих клиентов. Необходимость прямого взаимодействия по TCP с кастомным протоколом на основе JSON, а также интеграция специфичной функции управления задачами, диктуют разработку собственного клиентского приложения "с нуля". Данный подход позволяет полностью контролировать процесс взаимодействия с сервером, реализовать требуемый пользовательский интерфейс и, что немаловажно в контексте курсовой работы, глубже изучить фундаментальные аспекты сетевого программирования и разработки мобильных приложений.

**3. Выбор Платформы и Инструментов Разработки**

Первым шагом при проектировании мобильного приложения является выбор целевой платформы. Двумя доминирующими платформами на сегодняшний день являются Android от Google и iOS от Apple. Android обладает значительно большей долей мирового рынка мобильных операционных систем, что обеспечивает потенциально более широкий охват аудитории. Кроме того, разработка под Android более доступна, поскольку не требует обязательного наличия оборудования Apple (Mac) и позволяет использовать различные операционные системы для разработки, включая Linux. Платформа iOS, в свою очередь, характеризуется высокой платежеспособностью аудитории и меньшей фрагментацией устройств, но разработка под нее жестко привязана к экосистеме Apple. Учитывая доступность среды разработки на ОС Linux и широкое распространение платформы, для данной курсовой работы в качестве целевой платформы была выбрана Android.

Следующим важным решением является выбор подхода к разработке. Существуют два основных подхода: нативный и кроссплатформенный.

Нативная разработка под Android предполагает использование языков Kotlin или Java и официального Android SDK. Этот подход гарантирует максимальную производительность, полный доступ ко всем возможностям платформы и API, а также соответствие последним гайдлайнам дизайна. Основным недостатком является то, что код, написанный для Android, не может быть использован для других платформ, таких как iOS, что требует отдельной разработки и поддержки для каждой платформы.

Кроссплатформенная разработка позволяет использовать единую кодовую базу для создания приложений под несколько платформ (Android, iOS, а иногда и Web, Desktop). Это может значительно сократить время и стоимость разработки, упростить поддержку и обеспечить консистентность пользовательского опыта на разных устройствах. Однако кроссплатформенные решения могут иметь некоторые ограничения по производительности по сравнению с нативными, а доступ к специфическим функциям платформы может потребовать написания дополнительного нативного кода или использования плагинов.

В последние годы кроссплатформенные фреймворки достигли значительной зрелости, предлагая производительность, близкую к нативной, и богатые возможности для создания UI. Среди наиболее популярных кроссплатформенных решений выделяются Flutter, React Native и Xamarin.

Для реализации клиентского приложения в рамках данной работы был выбран фреймворк Flutter, разработанный Google. Этот выбор обусловлен рядом ключевых преимуществ:

Язык Dart: Flutter использует язык программирования Dart, также разработанный Google. Dart — это современный, объектно-ориентированный язык со строгой типизацией, оптимизированный для разработки клиентских приложений. Он поддерживает как Just-in-Time (JIT) компиляцию (для быстрой разработки и Hot Reload), так и Ahead-of-Time (AOT) компиляцию в эффективный нативный код (ARM, x64), что обеспечивает высокую производительность конечного приложения. Dart имеет отличную встроенную поддержку асинхронного программирования (async/await, Future, Stream), что критически важно для работы с сетью и отзывчивости UI.

Высокопроизводительный рендеринг: В отличие от многих других кроссплатформенных фреймворков, которые используют нативные UI-компоненты платформы, Flutter применяет собственный движок рендеринга Skia (тот же, что используется в Google Chrome и Android) для отрисовки каждого пикселя на экране. Это обеспечивает полный контроль над UI, высокую производительность (часто 60 или 120 кадров в секунду) и консистентность внешнего вида на всех платформах.

Богатый UI Toolkit: Flutter предлагает обширную библиотеку готовых, кастомизируемых виджетов, построенных в соответствии с гайдлайнами Material Design (для Android) и Cupertino (для iOS). Декларативный подход к построению UI, где интерфейс описывается как функция от состояния, упрощает разработку и управление сложными интерфейсами. Фундаментальная концепция "всё есть виджет" позволяет легко комбинировать и создавать собственные UI-элементы.

Hot Reload / Hot Restart: Одна из самых ценимых разработчиками функций Flutter. Hot Reload позволяет применять изменения в коде к работающему приложению за доли секунды, сохраняя при этом его текущее состояние. Это значительно ускоряет процесс разработки, экспериментирования с UI и отладки. Hot Restart перезагружает приложение быстрее, чем полная пересборка.

Активное сообщество и поддержка Google: Flutter имеет быстрорастущее и активное сообщество разработчиков, обширную документацию и большое количество сторонних пакетов и плагинов на репозитории pub.dev. Поддержка со стороны Google гарантирует дальнейшее развитие и стабильность фреймворка.

По сравнению с React Native, Flutter предлагает потенциально более высокую производительность из коробки за счет AOT-компиляции и отсутствия "моста" (bridge) для взаимодействия с нативными компонентами при отрисовке UI. Выбор Flutter для данного проекта представляется оптимальным, так как он сочетает высокую производительность, гибкость в создании UI и продуктивность разработки.

В качестве основной среды разработки (IDE) можно использовать как Android Studio, так и Visual Studio Code. Обе IDE предоставляют отличную поддержку Flutter и Dart через специализированные плагины, включая подсветку синтаксиса, автодополнение, инструменты отладки, интеграцию с Flutter DevTools и запуск приложений на эмуляторах и устройствах. Выбор между ними часто является делом личных предпочтений разработчика.

**4. Проектирование Архитектуры Клиентского Приложения**

Качество программного продукта во многом определяется его архитектурой. Хорошо спроектированная архитектура обеспечивает не только работоспособность приложения, но и его масштабируемость, тестируемость, поддерживаемость и понятность для разработчиков. Ключевым принципом построения архитектуры является разделение ответственностей (Separation of Concerns, SoC), когда различные логические части системы изолированы друг от друга и отвечают только за свою конкретную задачу.

В контексте мобильных приложений существует множество архитектурных паттернов, каждый из которых предлагает свой способ организации кода и взаимодействия между компонентами. Наиболее известные из них:

MVC (Model-View-Controller): Классический паттерн, разделяющий данные (Model), пользовательский интерфейс (View) и логику управления (Controller).

MVP (Model-View-Presenter): Вариация MVC, где Presenter выступает посредником между Model и View, делая View пассивным и улучшая тестируемость.

MVVM (Model-View-ViewModel): Популярен в современных фреймворках. ViewModel предоставляет данные и команды для View, часто используя привязку данных (data binding) или потоки данных для автоматического обновления UI. View становится еще более пассивным.

MVI (Model-View-Intent): Основан на однонаправленном потоке данных (Unidirectional Data Flow), где пользовательские действия (Intents) преобразуются в изменения состояния (Model), которое затем отображается (View).

Clean Architecture: Набор принципов, направленных на создание слабо связанных, тестируемых и независимых от внешних факторов (UI, БД, сеть) слоев приложения с четким направлением зависимостей (от внешних слоев к внутренним).

Для данного клиентского приложения на Flutter выбрана многоуровневая архитектура, принципы которой во многом пересекаются с Clean Architecture и MVVM. Эта архитектура предполагает разделение приложения на следующие основные слои:

Слой Представления (Presentation Layer): Этот слой отвечает за все, что видит и с чем взаимодействует пользователь. Он включает в себя:

Виджеты (UI): Экраны, кнопки, списки, поля ввода и другие элементы интерфейса, созданные с помощью виджетов Flutter. Их задача — отображать данные и передавать действия пользователя дальше.

Управление состоянием UI (локальное): Некоторые виджеты могут иметь свое собственное, локальное состояние (StatefulWidget с setState).

Взаимодействие с нижележащим слоем: Виджеты получают данные для отображения и вызывают методы из слоя управления состоянием/бизнес-логики в ответ на действия пользователя.

Слой Управления Состоянием / Бизнес-логики (State Management / Domain Layer): Этот слой является ядром приложения. Он содержит:

Провайдеры состояния / ViewModel / BLoC: Классы, отвечающие за управление состоянием конкретных экранов или функциональных модулей (например, AuthProvider, ChatProvider, TaskProvider). Они получают запросы от слоя представления, обрабатывают их (возможно, взаимодействуя со слоем данных), изменяют свое состояние и уведомляют слой представления о необходимости обновления UI.

(Опционально) Модели Представления (View Models): Могут использоваться для подготовки данных в удобном для отображения формате.

(Опционально) Use Cases / Interactors: Могут инкапсулировать конкретные сценарии бизнес-логики (например, LoginUserUseCase).

Слой Данных (Data Layer): Этот слой отвечает за взаимодействие с внешним миром — получение и отправку данных. Он включает:

Репозитории (Repositories - опционально): Паттерн, который абстрагирует источники данных. Репозиторий предоставляет интерфейс для слоя бизнес-логики, скрывая детали того, откуда берутся данные (из сети, локальной БД, кэша).

Источники Данных (Data Sources): Конкретные классы, отвечающие за взаимодействие с одним источником данных (например, NetworkDataSource для работы с сетью через NetworkService, LocalDataSource для работы с базой данных).

Сервисы (Services): Классы, инкапсулирующие низкоуровневую логику взаимодействия (например, NetworkService для управления TCP-соединением, DatabaseService для работы с SQLite).

Модели Данных (Data Models): Классы (Message, Task), представляющие структуру данных, получаемых от сервера или из БД. Могут включать логику парсинга/сериализации (например, методы fromJson, toJson).

Зависимости между слоями направлены сверху вниз: Слой Представления зависит от Слоя Управления Состоянием, который, в свою очередь, зависит от Слоя Данных. Это обеспечивает слабую связанность и высокую тестируемость каждого слоя по отдельности.

Выбор такой архитектуры обусловлен ее гибкостью, хорошей поддержкой инструментами управления состоянием в Flutter (например, Provider легко интегрируется с этим подходом) и четким разделением ответственностей, что упрощает разработку и дальнейшую поддержку приложения. Структура директорий проекта будет отражать это разделение (lib/presentation, lib/application (или providers), lib/domain (модели, репозитории), lib/infrastructure (сервисы, источники данных)).

**5. Сетевое Взаимодействие по Протоколу TCP**

Мобильные приложения часто работают в условиях нестабильного сетевого окружения. Соединение может прерываться при переключении между Wi-Fi и мобильной сетью, при входе в зону слабого покрытия или просто из-за ограничений операционной системы на фоновую активность приложений для экономии энергии. Поэтому реализация сетевого взаимодействия должна быть устойчивой к таким ситуациям и предусматривать механизмы обработки ошибок и восстановления соединения.

В основе взаимодействия разрабатываемого клиента с сервером лежит протокол TCP (Transmission Control Protocol). TCP является одним из основных протоколов стека TCP/IP и обеспечивает надежную, упорядоченную доставку потока данных между двумя узлами сети. Перед началом обмена данными TCP устанавливает соединение (трехэтапное рукопожатие), а затем гарантирует, что все отправленные байты будут доставлены получателю в правильном порядке и без потерь, используя механизмы подтверждений и повторных передач. Это делает TCP подходящим для задач, где важна целостность и последовательность данных, таких как передача файлов или команд управления, что актуально для обмена сообщениями в мессенджере. Основным недостатком TCP по сравнению, например, с UDP (User Datagram Protocol), являются большие накладные расходы на установку соединения и управление им, а также возможные задержки из-за необходимости ожидания подтверждений и повторных передач.

Фреймворк Flutter, используя язык Dart, предоставляет встроенные средства для работы с TCP-сокетами через библиотеку dart:io. Ключевым классом является Socket. Процесс взаимодействия включает следующие шаги:

Установка соединения: Метод Socket.connect(host, port, [timeout]) асинхронно пытается установить TCP-соединение с указанным хостом и портом. Параметр timeout позволяет ограничить время ожидания соединения. В случае успеха метод возвращает Future<Socket>, в случае неудачи (недоступность сервера, неверный адрес, таймаут) – выбрасывает исключение SocketException.

Отправка данных: После успешного соединения данные отправляются с помощью метода socket.add(List<int> data) или socket.write(Object data). Важно отправлять данные в виде списка байт (List<int> или Uint8List). Перед отправкой строковые данные (например, JSON) должны быть закодированы в байты, как правило, с использованием кодировки UTF-8 (utf8.encode(string)). Метод socket.flush() можно использовать для немедленной отправки данных из внутреннего буфера сокета.

Получение данных: Данные от сервера поступают асинхронно. Для их получения используется подписка на поток данных сокета: socket.listen((Uint8List data) { ... }, onError: (error) { ... }, onDone: () { ... }). Обработчик onData вызывается каждый раз, когда поступает новая порция данных (чанк) в виде Uint8List. Обработчик onError вызывается при возникновении ошибки сокета, а onDone – когда удаленная сторона закрыла соединение.

Закрытие соединения: Соединение закрывается методом socket.close(), который отправляет FIN-пакет удаленной стороне и дожидается завершения отправки буферизованных данных, или socket.destroy(), который немедленно разрывает соединение.

Одной из ключевых проблем при работе с TCP является то, что он представляет собой потоковый протокол. Это означает, что данные не обязательно приходят ровно теми порциями, которыми были отправлены. Одно сообщение может прийти несколькими частями (чанками), или несколько коротких сообщений могут прийти в одном чанке. Поэтому на прикладном уровне необходим механизм фрейминга – определения границ сообщений в потоке байт. В данном проекте, согласно протоколу сервера, в качестве разделителя сообщений используется символ новой строки (\n). Клиент должен накапливать поступающие байты в буфере, декодировать их в строку (UTF-8) и проверять наличие символа \n. При обнаружении разделителя из буфера извлекается полная строка сообщения (до \n), а оставшаяся часть буфера сохраняется для обработки следующих сообщений.

Извлеченная строка представляет собой JSON-объект. Для его преобразования в объект Dart используется библиотека dart:convert и функция jsonDecode(). Полученный Map<String, dynamic> затем используется для создания экземпляров соответствующих классов моделей (например, Message или Task с помощью фабричных конструкторов fromJson). При отправке данных объекты Dart преобразуются в JSON-строку с помощью jsonEncode(), к ней добавляется символ \n, и результат кодируется в байты UTF-8 для передачи в сокет. Необходимо также реализовывать обработку ошибок на этапе парсинга JSON, так как данные от сервера могут прийти в некорректном формате.

Важной частью сетевого слоя является управление состоянием соединения. Клиент должен отслеживать, установлено ли соединение в данный момент, и уметь обрабатывать его разрывы. При возникновении ошибки (onError) или закрытии соединения сервером (onDone), если разрыв не был инициирован пользователем, клиент должен предпринять попытку автоматического переподключения через определенный интервал времени (например, 5-10 секунд), чтобы восстановить связь без активного вмешательства пользователя. Статус соединения должен быть доступен другим частям приложения (например, UI), чтобы информировать пользователя или блокировать функции, требующие активного соединения.

**6. Управление Состоянием Приложения (State Management) в Flutter**

Одной из центральных задач при разработке интерактивных приложений, особенно с использованием декларативных UI-фреймворков, таких как Flutter, является управление состоянием (State Management). Состояние – это данные, которые могут изменяться со временем и влиять на то, что отображается на экране. Это может быть что угодно: статус аутентификации пользователя, содержимое поля ввода, список полученных сообщений, текущая выбранная вкладка или статус сетевого соединения.

Проблема заключается в эффективной организации этого состояния и обеспечении синхронизации между данными и пользовательским интерфейсом. Когда состояние изменяется (например, приходит новое сообщение), соответствующая часть UI должна автоматически обновиться. Примитивный подход с передачей данных и колбэков вниз по дереву виджетов (prop drilling) быстро становится громоздким и трудно поддерживаемым в приложениях сложнее "Hello World".

Во Flutter состояние можно условно разделить на два типа:

Локальное (Ephemeral) состояние: Состояние, которое относится к одному конкретному виджету или небольшой части дерева виджетов (например, текущая страница в PageView, статус завершения анимации). Для управления таким состоянием часто достаточно использовать StatefulWidget и вызывать метод setState(), который перестраивает виджет при изменении данных.

Глобальное (App) состояние: Состояние, которое используется в разных частях приложения и должно быть доступно из разных экранов (например, информация о залогиненном пользователе, список задач, настройки приложения). Для управления таким состоянием требуются более продвинутые решения.

Хотя Flutter предоставляет встроенный механизм InheritedWidget для передачи данных вниз по дереву виджетов, его прямое использование может быть многословным. Поэтому сообщество Flutter разработало множество подходов и библиотек для управления состоянием приложения. Наиболее популярные из них:

Provider: Простая и гибкая библиотека, рекомендованная командой Flutter. Она использует InheritedWidget под капотом, но предоставляет более удобный API для предоставления (providing) и потребления (consuming) состояния. Часто используется в связке с ChangeNotifier – классом из Flutter SDK, который позволяет объектам уведомлять слушателей об изменениях.

Riverpod: Разработан автором Provider как его эволюция. Решает некоторые проблемы Provider (например, безопасность при получении состояния во время компиляции), не зависит от BuildContext и предлагает более гибкие типы провайдеров.

BLoC (Business Logic Component) / Cubit: Популярный паттерн, особенно для сложных приложений. Он строго разделяет бизнес-логику от UI, используя потоки (Stream) для передачи состояний и событий. Cubit является упрощенной версией BLoC. Требует написания большего количества шаблонного кода по сравнению с Provider.

GetX: Многофункциональный фреймворк, предоставляющий решения не только для управления состоянием, но и для навигации, внедрения зависимостей и др. Критикуется некоторыми членами сообщества за излишнюю "магичность" и отход от стандартных практик Flutter.

MobX: Библиотека, реализующая принципы Transparent Functional Reactive Programming (TFRP). Использует аннотации и кодогенерацию для автоматического отслеживания зависимостей и обновления UI при изменении наблюдаемых данных.

Для данного проекта, учитывая его сложность (несколько экранов, асинхронное взаимодействие с сетью, необходимость разделения логики аутентификации, чата и задач) и желание сохранить относительную простоту реализации, был выбран Provider в связке с ChangeNotifier.

Обоснование выбора Provider:

Простота: Provider имеет низкий порог вхождения и понятный API, что хорошо подходит для изучения и применения в рамках курсовой работы.

Интеграция с Flutter: ChangeNotifier является частью Flutter SDK, а Provider органично его дополняет.

Разделение логики: Позволяет легко вынести логику управления состоянием в отдельные классы-провайдеры (AuthProvider, ChatProvider, TaskProvider), которые затем предоставляются дереву виджетов.

Производительность: Provider оптимизирован для минимизации ненужных перестроек виджетов.

Принцип использования:

Создается класс, наследуемый от ChangeNotifier (например, AuthProvider). Внутренние поля этого класса хранят состояние (например, \_isAuthenticated, \_currentUser). При изменении этих полей вызывается метод notifyListeners().

Экземпляр этого ChangeNotifier предоставляется дереву виджетов с помощью ChangeNotifierProvider в main.dart или выше по дереву относительно виджетов, которые будут его использовать.

В виджетах, которым нужно получить доступ к состоянию или вызвать методы провайдера, используются методы context.watch<T>() (подписывается на изменения и перестраивает виджет), context.read<T>() (получает провайдер без подписки) или виджет Consumer<T> (перестраивает только часть дерева).

Таким образом, при получении события login\_success от NetworkService, AuthProvider обновит свои поля \_isAuthenticated и \_currentUser, вызовет notifyListeners(), и виджеты, подписанные на AuthProvider (например, LoginScreen или HomeScreen), автоматически перестроятся, чтобы отразить новое состояние аутентификации. Аналогичный подход будет применен для управления списком сообщений в ChatProvider и списком задач в TaskProvider.

**7. Проектирование Пользовательского Интерфейса (UI) и Взаимодействия (UX)**

Пользовательский интерфейс (UI) и опыт взаимодействия (UX) играют ключевую роль в успехе любого мобильного приложения. Даже самое функциональное приложение не найдет отклика у пользователей, если оно неудобно, непонятно или неэстетично. Проектирование UI/UX для мобильных устройств требует учета специфики платформы, размера экрана, сенсорного ввода и паттернов поведения пользователей.

Основные принципы хорошего мобильного UX включают:

Простота: Интерфейс должен быть лаконичным, не перегруженным элементами. Пользователь должен легко находить нужные функции.

Интуитивность: Навигация и взаимодействие должны быть понятны без дополнительных инструкций, основываясь на привычных паттернах.

Отзывчивость: Приложение должно быстро реагировать на действия пользователя, предоставлять визуальную обратную связь (анимации нажатия, индикаторы загрузки).

Соответствие гайдлайнам: Приложение должно выглядеть и вести себя привычно для пользователей конкретной платформы (Android или iOS), следуя ее дизайнерским рекомендациям.

Для платформы Android основным руководством по дизайну является Material Design, разработанный Google. Он предлагает целостную систему визуального языка, принципов взаимодействия и компонентов для создания качественных цифровых продуктов. Ключевые концепции Material Design включают использование метафоры физического мира (поверхности, тени для создания иерархии), осмысленное применение цвета и типографики, понятную иконографию и плавные анимации для обратной связи.

Flutter предоставляет превосходную поддержку Material Design "из коробки". Фреймворк включает обширный набор готовых виджетов, реализующих компоненты Material Design:

Scaffold: Базовая структура экрана Material Design (включает AppBar, body, FloatingActionButton, Drawer, BottomNavigationBar и др.).

AppBar: Верхняя панель приложения с заголовком, иконками действий.

TextField: Поле для ввода текста с поддержкой стилей Material.

ElevatedButton, TextButton, OutlinedButton: Различные типы кнопок.

ListView, GridView: Виджеты для отображения прокручиваемых списков и сеток.

Card: Карточка с тенью для группировки контента.

Icon: Отображение иконок из библиотеки Material Icons.

BottomNavigationBar: Нижняя навигационная панель для переключения между основными разделами приложения.

Структура UI разрабатываемого мессенджера будет включать следующие основные экраны:

Экран Входа (LoginScreen): Поля для ввода имени пользователя и пароля, кнопка "Войти", ссылка на экран регистрации.

Экран Регистрации (RegisterScreen): Поля для ввода имени пользователя и пароля, кнопка "Зарегистрироваться".

Главный Экран (HomeScreen): После успешного входа. Будет содержать BottomNavigationBar для переключения между разделами "Чаты" и "Задачи". Возможно, будет отображать список пользователей/чатов (раздел "Чаты") или список задач (раздел "Задачи") в зависимости от выбранной вкладки. AppBar будет отображать имя текущего пользователя и кнопку выхода.

Экран Чата (ChatScreen): Открывается при выборе пользователя из списка. Отображает историю сообщений с выбранным пользователем (ListView.builder, возможно, перевернутый для отображения новых сообщений снизу), поле для ввода нового сообщения и кнопку отправки.

Экран Списка Задач (TaskListScreen): Отображает список активных задач пользователя (ListView.builder), кнопку для перехода на экран добавления новой задачи.

Экран Добавления Задачи (AddTaskScreen): Форма с полями для ввода названия задачи, описания, времени срабатывания (с использованием виджета выбора даты и времени) и смещения уведомления. Кнопка "Добавить задачу".

Фундаментальной концепцией Flutter является "всё есть виджет". Интерфейс строится путем композиции виджетов – от простых (текст, иконка, отступ) до сложных (экран, список, форма). Виджеты бывают двух основных типов: StatelessWidget (неизменяемое состояние, перестраивается только при изменении входных параметров) и StatefulWidget (имеет изменяемое внутреннее состояние, может перестраиваться вызовом setState()).

Для отображения динамически изменяемых списков (сообщения, пользователи, задачи) будет использоваться конструктор ListView.builder, который эффективно создает элементы списка по мере их появления на экране, что важно для производительности при работе с большими объемами данных.

Обработка пользовательского ввода будет реализована с помощью виджетов Form и TextFormField, которые предоставляют удобные средства для сбора данных и их валидации перед отправкой на сервер или обработкой в приложении. Навигация между экранами будет осуществляться с помощью встроенного в Flutter Navigator API, используя методы Navigator.push (открыть новый экран поверх текущего), Navigator.pop (закрыть текущий экран) и Navigator.pushReplacement или Navigator.pushAndRemoveUntil (для переходов, не позволяющих вернуться назад, например, после логина или логаута).

**8. Асинхронное Программирование и Обработка Данных**

Современные мобильные приложения интенсивно взаимодействуют с внешним миром (сеть, файловая система, база данных) и реагируют на действия пользователя. Эти операции часто являются длительными и не должны блокировать основной поток выполнения, отвечающий за отрисовку пользовательского интерфейса, чтобы приложение оставалось отзывчивым. Это достигается за счет использования асинхронного программирования.

Язык Dart имеет мощную и удобную встроенную поддержку асинхронности, основанную на модели Event Loop. Все события (ввод пользователя, таймеры, сетевые ответы) помещаются в очередь событий. Event Loop последовательно извлекает события из очереди и обрабатывает их. Длительные операции выполняются асинхронно, не блокируя Event Loop. Основными инструментами для работы с асинхронностью в Dart являются:

Future: Объект, представляющий результат асинхронной операции, который будет доступен в будущем (либо значение, либо ошибка).

async: Ключевое слово, помечающее функцию как асинхронную (она неявно возвращает Future).

await: Ключевое слово, используемое внутри async-функций для ожидания завершения Future. Оно приостанавливает выполнение async-функции до получения результата Future, но не блокирует основной поток и Event Loop.

Обработка ошибок в асинхронном коде выполняется с помощью стандартных блоков try-catch (при использовании await) или метода .catchError() у объекта Future.

Для работы с последовательностями асинхронных событий (например, непрерывный поток данных из TCP-сокета, пользовательские клики, изменения состояния) в Dart используется концепция Потоков (Stream). Stream представляет собой асинхронную последовательность данных. Для прослушивания потока используется метод stream.listen(), который принимает колбэки для обработки данных (onData), ошибок (onError) и завершения потока (onDone). Для создания собственных потоков используется класс StreamController. В данном проекте StreamController.broadcast() будет использоваться в NetworkService для трансляции входящих JSON-сообщений всем заинтересованным подписчикам (например, AuthProvider, ChatProvider). StreamSubscription, возвращаемый методом listen, используется для управления подпиской (пауза, возобновление, отмена).

Обработка данных включает несколько этапов. Во-первых, это парсинг JSON-строк, полученных от сервера, в объекты Dart (Map<String, dynamic>) с использованием jsonDecode. Во-вторых, это преобразование этих объектов в строго типизированные модели данных (Message, Task) с помощью фабричных конструкторов fromJson. Эти модели инкапсулируют структуру данных и могут содержать дополнительную логику, связанную с данными. В-третьих, это обработка специфичных типов данных, таких как дата и время. Сервер передает временные метки в стандартном формате ISO 8601 UTC (например, 2023-10-27T10:00:00Z). В Dart для работы с датой и временем используется класс DateTime. Метод DateTime.parse() позволяет преобразовать строку ISO 8601 в объект DateTime. Поскольку сервер отдает время в UTC, а пользователю удобнее видеть время в его локальном часовом поясе, используется метод .toLocal() для преобразования DateTime из UTC в локальное время устройства. Для форматирования DateTime в удобочитаемый вид (например, "14:35" или "26 окт. 2023 г.") будет использоваться пакет intl, предоставляющий широкие возможности для интернационализации и форматирования дат, времени и чисел.

**9. Вопросы Безопасности и Тестирования**

При разработке любого приложения, особенно связанного с передачей данных по сети, необходимо уделять внимание вопросам безопасности. В контексте разрабатываемого мессенджера основные аспекты безопасности связаны с аутентификацией и передачей данных. Сервер отвечает за проверку учетных данных пользователя (логин и пароль) при аутентификации. Клиентское приложение отправляет эти данные на сервер. Критически важным недостатком текущей архитектуры взаимодействия является отсутствие шифрования TCP-соединения. Это означает, что все передаваемые данные, включая логины, пароли и тексты сообщений, передаются по сети в открытом виде. В реальной сети это представляет серьезную уязвимость, так как злоумышленник, имеющий доступ к сетевому трафику (например, в публичной Wi-Fi сети), может легко перехватить и прочитать эту информацию. Для защиты данных в продуктовой среде абсолютно необходимо использовать шифрование транспортного уровня, например, инкапсулируя TCP-соединение в TLS/SSL. В рамках данной курсовой работы этот аспект рассматривается как известное ограничение и потенциальное направление для улучшения. Клиентская валидация вводимых данных (проверка на пустоту, возможно, формат) также является частью обеспечения качества, хотя и не заменяет серверную валидацию. Если бы приложение требовало хранения чувствительных данных на устройстве (например, токена сессии), следовало бы использовать безопасное хранилище, предоставляемое пакетом flutter\_secure\_storage, который использует Keychain на iOS и Keystore на Android.

Тестирование является неотъемлемой частью процесса разработки программного обеспечения, позволяя гарантировать корректность работы приложения, предотвращать регрессии (появление старых ошибок после внесения изменений) и упрощать рефакторинг кода. Фреймворк Flutter предоставляет развитую систему для написания различных видов тестов:

Unit-тесты: Тестируют наименьшие изолированные части кода – отдельные функции, методы или классы – без зависимостей от UI и внешних сервисов. Например, можно протестировать логику парсинга JSON в моделях данных, методы валидации ввода или чистую бизнес-логику в провайдерах состояния (с использованием моков для зависимостей). Для unit-тестов используется пакет test.

Widget-тесты: Тестируют отдельные виджеты Flutter. Они позволяют проверить, корректно ли виджет отображается, реагирует ли на взаимодействие (нажатия кнопок, ввод текста) и правильно ли управляет своим состоянием. Widget-тесты выполняются в специальной тестовой среде без необходимости запуска на реальном устройстве или эмуляторе. Используется пакет flutter\_test.

Integration-тесты: Тестируют приложение целиком или его значительные части, включая взаимодействие между различными компонентами (UI, провайдеры, сервисы) и работу с реальными или мокированными внешними зависимостями (сеть, БД). Integration-тесты выполняются непосредственно на устройстве или эмуляторе. Используется пакет integration\_test.

Помимо автоматизированных тестов, важную роль играет ручное тестирование и использование инструментов отладки. К ним относятся:

Логирование: Вывод отладочной информации в консоль. Рекомендуется использовать специализированные пакеты, такие как logger, предоставляющие более структурированный и информативный вывод по сравнению со стандартным print().

Точки останова (Breakpoints): Возможность приостановить выполнение кода в определенном месте и пошагово исследовать состояние переменных и поток выполнения в IDE.

Flutter DevTools: Мощный набор инструментов, запускаемый в браузере, который позволяет инспектировать дерево виджетов, анализировать производительность отрисовки и выполнения кода, отслеживать сетевые запросы, исследовать использование памяти и многое другое.

Комплексное применение различных видов тестирования и инструментов отладки позволяет создавать более надежные и качественные приложения.

**10. Заключение(теория)**

В ходе выполнения теоретической части данной курсовой работы был проведен всесторонний анализ задачи разработки клиентского Android-приложения для взаимодействия с существующим TCP-сервером мессенджера. Были рассмотрены актуальность темы, проанализированы существующие аналоги и их архитектурные особенности, что позволило выявить специфику разрабатываемого решения.

На основе сравнительного анализа платформ и подходов к мобильной разработке был сделан обоснованный выбор в пользу платформы Android и кроссплатформенного фреймворка Flutter. Преимущества Flutter, такие как высокая производительность, гибкость UI-инструментария, быстрая разработка благодаря Hot Reload и активное сообщество, делают его оптимальным выбором для данного проекта.

Была спроектирована многоуровневая архитектура приложения, обеспечивающая разделение ответственностей между слоями представления, управления состоянием и данных, что способствует созданию поддерживаемого и тестируемого кода. В качестве механизма управления состоянием приложения был выбран Provider в связке с ChangeNotifier как достаточно простое и эффективное решение для управления состоянием аутентификации, чатов и задач.

Детально рассмотрены особенности сетевого взаимодействия по протоколу TCP на мобильных устройствах с использованием библиотеки dart:io. Особое внимание уделено вопросам обработки потока данных, фрейминга сообщений с использованием разделителя \n, парсинга JSON и реализации логики автоматического переподключения. Также были затронуты принципы проектирования пользовательского интерфейса с использованием гайдлайнов Material Design и виджетов Flutter, и рассмотрены основы асинхронного программирования в Dart, необходимые для создания отзывчивого приложения. Затронуты важные аспекты безопасности, в частности, указана критическая необходимость шифрования канала связи в реальных условиях, и описаны подходы к тестированию Flutter-приложений.

Проведенное теоретическое исследование и принятые проектные решения формируют прочную основу для практической реализации клиентского приложения. Выбранные технологии и подходы позволяют эффективно решить поставленные задачи и разработать функциональное Android-приложение, соответствующее требованиям взаимодействия с заданным TCP-сервером. Следующим этапом работы является непосредственная разработка приложения на основе изложенных теоретических положений.