

СОДЕРЖАНИЕ

Введение.....	9
1 Обзор литературы.....	10
1.1 Обзор существующих аналогов.....	10
1.2 Zigbee	14
1.3 MQTT	14
1.4 Интегрированная среда разработки Android Studio.....	16
1.5 Архитектура приложений.....	17
1.6 Язык программирования Kotlin	20
1.7 Jetpack Compose	20
2 Системное проектирование	22
2.1 Блок MQTT клиента	22
2.2 Блок соединения с брокером MQTT.....	23
2.3 Блок коммуникации MQTT	24
2.4 Блок обработки и генерации сообщений	24
2.5 Блок базы данных	25
2.6 Блок моделей.....	25
2.7 Блок бизнес-логики	26
2.8 Блок представлений	26
2.9 Блок моделей представлений	27
3 Функциональное проектирование	28
3.1 Представления	28
3.2 Модели представлений	33
3.3 Модели.....	35
3.4 Хранение данных.....	38
3.5 MQTT и взаимодействие с брокером	42
3.6 Вспомогательные компоненты системы.....	44
4 Разработка программных модулей	46
4.1 Управление соединением с MQTT-брокером	46
4.2 Обработка данных полученных от MQTT-брокера	48
4.3 Отправка пользовательских данных в MQTT-брокер	50
4.4 Управление модулем светодиодной подсветки	51
4.5 Отображение общего списка устройств системы	53
4.6 Отображение списка комнат системы.....	55
4.7 Добавление новых комнат в систему	56
4.8 Управление режимом обнаружения и подключения новых устройств	57
4.9 Отображение детальной информации об устройстве.....	58
4.10 Привязка устройства к определенной комнате	59
5 Руководство пользователя	60
5.1 Руководство пользователя координатора	60
5.2 Руководство пользователя MQTT-брокера.....	64
5.3 Руководство пользователя мобильного приложения	65
5.4 Руководство программиста	70
6 Программа и методика испытаний	73

6.1 Автоматическое тестирование	73
6.2 Ручное тестирование	74
7 Технико-экономическое обоснование разработки и реализации на рынке Android-приложения для управления и мониторинга устройствами умного дома	79
7.1 Характеристика программного средства, разрабатываемого для реализации на рынке	79
7.2 Расчет инвестиций в разработку программного средства	79
7.3 Расчет экономического эффекта от реализации программного средства на рынке	82
7.4 Расчет показателей экономической эффективности разработки и реализации программного средства на рынке	83
7.5 Вывод об экономической целесообразности реализации проектного решения	84
Заключение	85
Приложение А	88
Приложение Б	89

ВВЕДЕНИЕ

Благодаря развитию передовых технологий повседневная жизнь человека становится удобнее и эффективнее. Развитие систем «умный дом» позволило автоматизировать большинство бытовых процессов, обеспечивая комфорт для пользователей. Современные решения позволяют управлять освещением, климатом, системами безопасности и другими элементами, создавая интеллектуальную и адаптивную среду.

Развитие универсальных стандартов связи значительно упростило взаимодействие устройств разных производителей, обеспечивая их совместимость и позволяя пользователям создавать гибкие, масштабируемые экосистемы для управления. В результате современные системы автоматизации становятся все более функциональными и доступными.

Несмотря на все преимущества, вопрос стоимости остается одним из ключевых факторов, ограничивающих массовое внедрение систем «умный дом». Высокие цены на оборудование, необходимость приобретения дополнительных шлюзов и контроллеров, а также расходы на установку и настройку могут стать серьезным барьером для пользователей. Кроме того, некоторые устройства требуют регулярного обновления или подписки на облачные сервисы, что увеличивает общую стоимость использования внедряемой системы.

Однако, помимо готовых коммерческих решений, все большую популярность приобретают DIY-устройства (Do It Yourself – «сделай сам»), позволяющие пользователям самостоятельно создавать и настраивать системы «умного дома» под свои нужды. Такой подход не только снижает затраты, но и дает возможность полной кастомизации, позволяя интегрировать исполнительные устройства в единую интеллектуальную сеть.

Целью данного дипломного проекта является разработка Android-приложения, которое обеспечит мониторинг и управление устройствами «умного дома», предоставляя пользователю интуитивно понятный интерфейс для эффективного контроля, настройки и автоматизации различных устройств.

Для достижения поставленной цели дипломного проекта важно разбить ее на конкретные задачи:

- прошивка и настройка шлюза «умного дома»;
- проектирование архитектуры системы «умный дом»;
- разработка Android-приложения для управление системой «умный дом»;
- тестирование и отладка разработанного приложения.

Данный дипломный проект выполнен мной лично, проверен на заимствования, процент оригинальности составляет xx% (отчет о проверке на заимствования прилагается).

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Обзор существующих аналогов

Одним из первых этапов разработки приложения для управления и мониторинга «умным домом» является определение перечня необходимых функций и выбор подходящих технологий. Для принятия обоснованных решений важно проанализировать преимущества и недостатки существующих решений на рынке.

Многие пользователи не имеют полного представления о возможностях современных систем «умного дома», их функционале и технических особенностях. Такие приложения могут включать широкий спектр функций – от управления освещением и климатом до мониторинга безопасности и автоматизации сценариев. В дальнейшем будут рассмотрены существующие решения, схожие по функционалу с разрабатываемым приложением.

1.1.1 Системы от «Xiaomi» предоставляет широкий ассортимент разнообразной техники, в том числе технику, которая отвечает требованиям «умного дома» [1]. В их ассортименте можно найти большое количество разнообразных устройств, что делает их продукцию универсальным решением для создания современного «умного дома».

Для удобного управления и взаимодействия с этими устройствами предоставляется специальное приложение, обеспечивающее полную совместимость устройств одной экосистемы и их простую настройку. Благодаря фокусу данного производителя на широкий потребительский рынок, все устройства легко подключаются и не требуют значительных временных затрат на настройку.

Для построения системы «умного дома» с устройствами Xiaomi и использованием протокола MQTT необходим централизованный хаб Xiaomi Smart Home Hub 2. Этот хаб выступает в роли шлюза между беспроводными устройствами, поддерживающими Zigbee 3.0, Bluetooth Mesh, Wi-Fi и Matter, обеспечивая их взаимодействие в единой экосистеме.

Xiaomi Smart Home Hub 2 подключается к сети через Ethernet или Wi-Fi и работает от розетки. В отличие от некоторых других хабов, он не оснащен встроенным аккумулятором, поэтому при отключении питания теряет связь с устройствами. Он поддерживает локальное управление через MQTT, что позволяет интегрировать его в собственные серверные решения, такие как HomeAssistant. Благодаря современному оборудованию хаб обеспечивает стабильную работу, минимальные задержки и возможность автоматизации без облачных сервисов.

Приложение Xiaomi Home является основным инструментом для управления устройствами «умного дома» от Xiaomi, а также совместимыми гаджетами других брендов, поддерживающих экосистему Mi Home. Оно доступно для Android и iOS, обеспечивая удобное удаленное взаимодействие с устройствами.

С помощью приложения можно добавлять устройства, управлять их настройками, создавать сценарии автоматизации и получать уведомления. Приложение поддерживает групповое управление, позволяя объединять устройства по комнатам и сценариям. Также реализована интеграция с голосовыми помощниками (для совместимых устройств).

Xiaomi Home позволяет настроить локальное управление для некоторых устройств, но большая часть автоматизаций выполняется через облачные серверы Xiaomi, что требует постоянного подключения к интернету. Интерфейс приложения интуитивно понятен, но скорость работы может зависеть от количества подключенных устройств и качества интернет-соединения.

Графический интерфейс приложения представлен на рисунке 1.1.

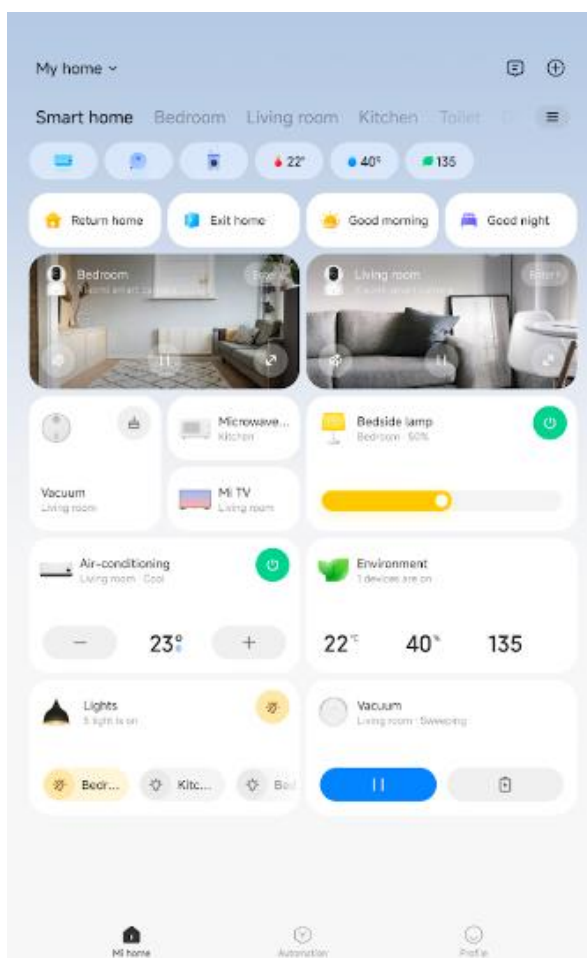


Рисунок 1.1 – Интерфейс приложения Xiaomi Home

Приложение Xiaomi Home предоставляет пользователям удобный способ управления устройствами умного дома от Xiaomi. Однако, на основе отзывов пользователей, можно выделить несколько недостатков:

1 Для корректной работы некоторых устройств и функций в приложении Xiaomi Home пользователям часто приходится выбирать определенный регион, например, Китай. Это связано с тем, что некоторые устройства могут быть доступны только в определенных регионах, а их функциональность

может отличаться в зависимости от выбранных настроек.

2 Для стабильной работы некоторых функций рекомендуется использовать английский язык. Это может быть неудобно для пользователей, предпочитающих русский интерфейс, так как требует дополнительной адаптации.

3 Приложение официально поддерживает русский язык, но некоторые разделы и плагины могут отображаться на китайском или английском. Это создает сложности при использовании и может привести к затруднениям в настройке устройств.

Таким образом приложение Xiaomi Home является удобным инструментом для управления устройствами умного дома, предлагая широкий функционал и интеграцию с экосистемой Xiaomi. Однако его использование может вызывать трудности из-за ограничений, связанных с региональными настройками. Несмотря на недостатки, приложение остается популярным благодаря своей универсальности, низкой стоимости и поддержке большого количества устройств.

1.1.2 Решения от компании «Яндекс» представляют собой экосистему, интегрированную с голосовым помощником Алисой. В ассортименте представлены устройства, такие как умные лампы, розетки, датчики движения и видеокамеры, а также шлюзы для взаимодействия с техникой сторонних производителей (например, Xiaomi, Philips Hue) через облачную интеграцию [2]. Основное преимущество системы – глубокая интеграция с сервисами Яндекса (Музыка, Погода, Расписания) и поддержка русского языка на всех уровнях взаимодействия.

В качестве шлюза чаще всего используется Яндекс Станция (умная колонка с голосовым управлением) или Яндекс Модуль, который подключается к сети через Wi-Fi и выступает мостом между устройствами умного дома и облачными сервисами. Шлюз поддерживает протоколы Wi-Fi и Bluetooth, но не Zigbee, что ограничивает совместимость с некоторыми устройствами. Управление через MQTT или локальные решения не предусмотрено – все команды обрабатываются через облако Яндекса. Это обеспечивает простоту настройки, но создает зависимость от интернет-соединения и серверов компании.

Приложение доступно для Android и iOS и служит центральным инструментом для управления устройствами. Ключевые функции данных приложений:

- добавление и группировка устройств по комнатам;
- создание сценариев автоматизации (например, «если датчик движения сработал, включить свет»);
- интеграция с голосовым помощником Алисой для управления через речь;
- уведомления о событиях (срабатывание датчиков, отключение устройств);
- совместимость с устройствами других брендов через облачные

плагины (например, TP-Link, Redmond).

Интерфейс приложения интуитивно понятен и полностью русифицирован (см. рисунок 1.2). Для настройки сценариев используется визуальный редактор, где пользователь может задавать условия и действия с помощью «перетаскивания» элементов.

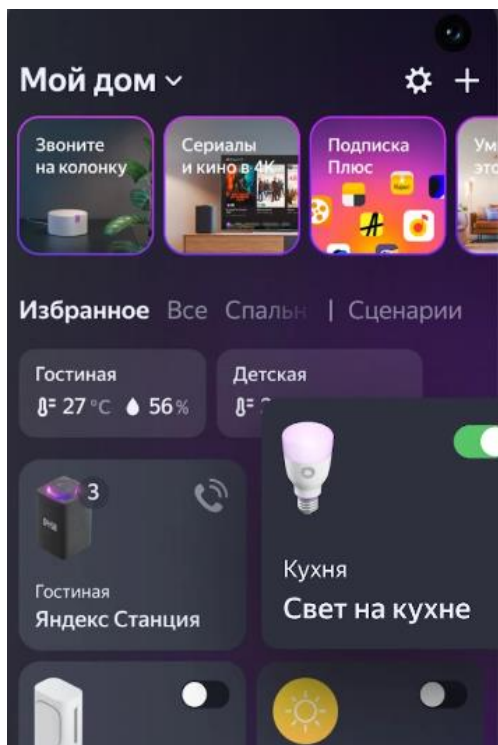


Рисунок 1.2 – Интерфейс приложения «Дом с Алисой»

Приложение «Яндекс.Умный дом» обеспечивает глубокую интеграцию с экосистемой сервисов Яндекса, что позволяет пользователям управлять устройствами через голосового помощника Алису и автоматически добавлять технику по QR-коду без ручного ввода данных. Локализация интерфейса, включая полную поддержку русского языка и голосовых команд, делает решение доступным для русскоязычной аудитории. Настройка устройств максимально упрощена: приложение не требует технических знаний, что идеально подходит для новичков. Кроме того, облачные сценарии автоматизации позволяют сохранять работоспособность функций даже при выключенном телефоне, так как логика обработки событий выполняется на серверах Яндекса.

Основным ограничением системы является сильная зависимость от облачных сервисов: большинство операций, включая управление устройствами и выполнение сценариев, требуют стабильного интернет-соединения, а локальное управление (например, через LAN) недоступно. Это создает риск потери контроля над умным домом при сбоях в сети. Кроме того, приложение поддерживает ограниченный набор протоколов – отсутствие Zigbee и MQTT сужает выбор совместимых устройств, делая экосистему закрытой.

Приложение «Яндекс.Умный дом» предлагает удобное решение для пользователей, которые ценят простоту и интеграцию с русскоязычными сервисами. Однако его зависимость от облака и закрытая экосистема ограничивают гибкость, что критично для продвинутых сценариев автоматизации. В отличие от Xiaomi, система Яндекса не поддерживает локальные протоколы (например, Zigbee), что делает ее менее универсальной.

1.2 Zigbee

Беспроводной протокол Zigbee [3], предназначенный для передачи данных между устройствами в сетях с низким энергопотреблением. Он работает на частоте 2,4 ГГц и поддерживает три типа устройств:

- координатор (главное устройство, управляющее сетью);
- роутер (передает сигналы другим устройствам);
- оконечное устройство (датчики, выключатели и другие узлы, потребляющие минимум энергии).

Благодаря сетевой топологии Mesh устройства могут передавать данные друг через друга, увеличивая дальность связи.

Zigbee активно применяется в системах умного дома, промышленной автоматизации, медицине и IoT-решениях. Его ключевые преимущества – низкое энергопотребление, высокая надежность за счет самовосстанавливающейся сети и совместимость с разными производителями. Однако у него есть и недостатки: ограниченная пропускная способность делает его неподходящим для передачи мультимедиа, а необходимость координатора может усложнять настройку сети. Кроме того, возможны помехи от Wi-Fi, так как оба протокола работают на одной частоте.

1.3 MQTT

Большинство устройств «умного дома» находятся в непосредственной близости друг от друга и работают в локальной сети, но для их работы в качестве IoT-устройств необходимо подключение к сети Интернет. Это позволит организовать мониторинг и управление устройствами из любой точки планеты при наличии интернет-доступа. Этот подход значительно облегчает взаимодействие человека с системой, поскольку сегодня мобильный интернет доступен практически повсюду.

В большинстве современных систем «умный дом» управление системой производится с использованием мобильного телефона с выходом в интернет, в редких случаях используются веб-приложения доступные как для телефонов, так и для персональных компьютеров с помощью веб-браузера. Такой подход делает мобильное устройство еще одним IoT-устройством.

В непосредственной близости для обмена данными между устройствами и координатором умного дома применяются такие протоколы, как Zigbee, Z-Wave и другие. Однако для связи координатора со смартфоном эти протоколы не подходят, так как они имеют ограниченный радиус действия, не

поддерживаются в большинстве мобильных устройств и требуют специального оборудования для приема и передачи данных. Для этих целей используется подключение к интернету с использованием протокола MQTT.

Протокол MQTT, разработанный на основе TCP/IP, оптимизирован для потоковой передачи данных в сетях с низкой пропускной способностью, что делает его особенно востребованным при создании Android-приложений для удаленного управления системой «умный дом». Его преимущества перед другими протоколами и ключевые особенности включают:

- поддержка сохраненных сообщений (retained message), что позволяет новым клиентам получать актуальную информацию сразу после подключения;
- для работы в условиях нестабильного подключения реализован механизм QoS (Quality of Service) который обеспечивает надежность доставки сообщений;
- имеет поддержку функционала LWT (Last Will and Testament, «последняя воля и завещание»), что используется для оповещения брокера о нештатном отключении клиента;
- минимальная задержка передачи данных, благодаря легковесности передаваемых сообщений.

Система связи, использующая протокол MQTT (см. рисунок 1.3), включает в себя издателей (publisher), сервер-брокер (MQTT broker) и одного или нескольких клиентов (subscribers). Издателю не требуется дополнительная настройка для определения числа или местоположения подписчиков, которые будут получать сообщения. Также подписчики не нуждаются в настройке для определения конкретного издателя. Издатель отправляет сообщения в темы (topic), откуда их могут читать клиенты, подписанные на данную тему. Издатель и подписчик не могут коммуницировать напрямую, поэтому они не знают о существовании друг друга.

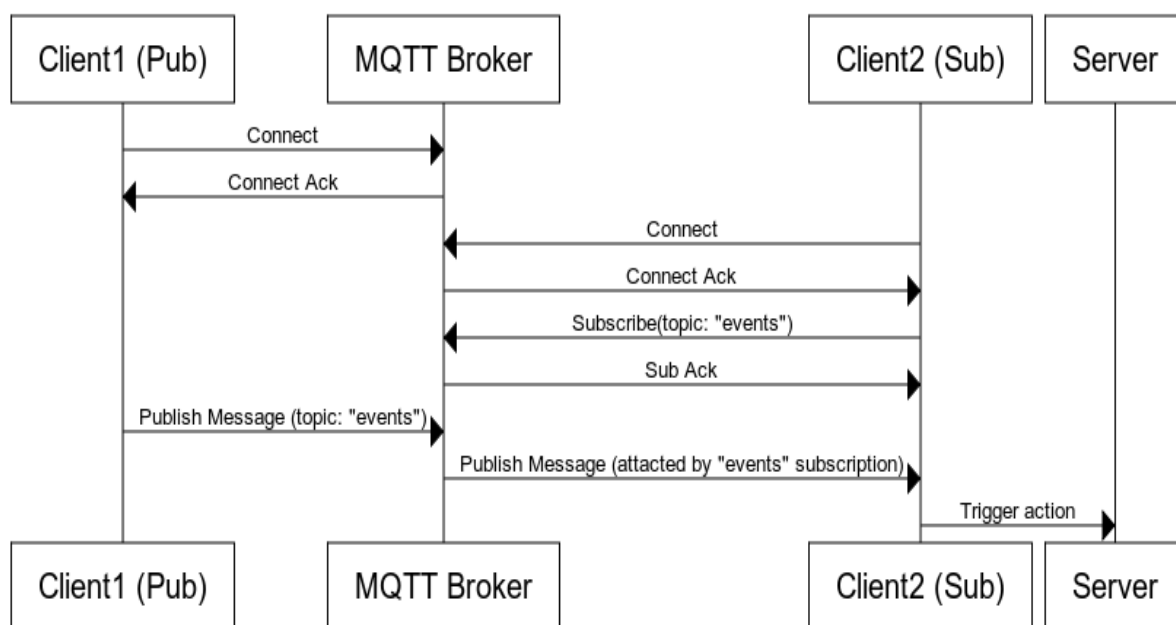


Рисунок 1.3 – Схема взаимодействия по протоколу MQTT

В MQTT данные передаются в виде `payload`, которые могут быть любыми байтовыми данными. Это означает, что информация может быть представлена в виде строки, JSON, двоичных данных или даже изображений – все зависит от того, что отправитель и получатель согласуют. Однако, в большинстве случаев, данные передаются в формате JSON, так как этот формат легко обрабатывается, обладает хорошей читаемостью и широко используется для обмена структурированными данными в приложениях, связанных с IoT и умными устройствами.

После отправки, данные хранятся в топиках, которые составлены из символов в кодировке UTF8 и имеют структуру, схожую с деревом файловых UNIX подобных операционных систем. Данный подход обеспечивает дополнительную читаемость для пользователей. В некоторых случаях устройства отправляют данные в топики, которые соответствуют их адресу, например:

- `home/kitchen/plug`;
- `home/restroom/light/ceiling`;
- `zigbee/0xA4C138DEB58ECDEC`.

Данная структура топиков изначально задана в устройстве, но может быть изменена пользователем при необходимости.

Организация топиков с использованием дружественных имен для устройств позволяет наглядно отслеживать передаваемые данные и упрощает разработку и отладку кода, не требуя запоминания числовых адресов расположения данных.

Для взаимодействия устройств с брокером предусмотрены различные типы сообщений, ниже перечислены основные типы для функционирования системы:

- `connect` – установка соединения клиента с брокером;
- `disconnect` – разрыв соединения клиента с брокером;
- `publish` – публикация данных клиента в топик;
- `subscribe` – подписка клиента на получение сообщений из топика.

Таким образом, протокол MQTT является отличным средством для обмена данными в системах «умный дом». Его простота и гибкость позволяют легко интегрировать устройства с различными уровнями взаимодействия. Возможность использования структуры топиков, которая напоминает файловую систему, делает обмен данными более удобным и читаемым для пользователя. MQTT также поддерживает различные уровни качества обслуживания (QoS), что обеспечивает надежность передачи сообщений даже в условиях нестабильной сети. Эти особенности делают MQTT идеальным выбором для создания масштабируемых и устойчивых решений в умных домах, промышленности и других областях.

1.4 Интегрированная среда разработки Android Studio

Android Studio включает в себя редакторы кода, визуальные инструменты для разработки пользовательских интерфейсов, отладчики и

профилировщики производительности, а также средства для автоматизации и управления проектами.

Можно устанавливать и отлаживать приложения как в эмуляторе, так и на физических устройствах, подключенных по USB или беспроводным способом. В режиме отладки Android Studio предоставляет подробную информацию о работе приложения, позволяя приостанавливать его выполнение, анализировать состояние переменных и инспектировать пользовательский интерфейс.

Среда также включает библиотеку инструментов для работы с базами данных, мультимедийными файлами, сетевыми протоколами и графикой. Функция Live Edit позволяет мгновенно просматривать изменения в пользовательском интерфейсе Jetpack Compose без необходимости полной перекомпиляции приложения. Это помогает разработчикам оперативно тестировать идеи и настраивать внешний вид, а также проверять адаптацию интерфейса для разных устройств, ориентаций экрана и цветовых схем.

1.5 Архитектура приложений

Разработка Android-приложений опирается на несколько архитектурных подходов, которые помогают организовать код, упростить поддержку и улучшить масштабируемость проекта. Архитектура определяет принципы разделения ответственности между компонентами приложения, что особенно важно в условиях сложных пользовательских интерфейсов и многопоточных операций.

Среди наиболее распространенных архитектур для Android можно выделить MVC (Model-View-Controller) и MVVM (Model-View-ViewModel). Эти модели помогают структурировать код так, чтобы интерфейс, логика обработки данных и бизнес-логика были четко разграничены, что снижает зависимость компонентов и делает приложение более гибким.

1.5.1 Model-View-Controller (MVC) – это архитектурный паттерн, который является основой для разработки приложений. Он обеспечивает разделение приложения на три основных взаимодействующих компонента: модель (Model), представление (View) и контроллер (Controller) представленных на рисунке 1.4.

Использование паттерна MVC позволяет создавать гибкие и масштабируемые приложения, где каждый компонент отвечает за свою часть функциональности.

Модель отвечает за управление данными и их обработку, включая реализацию бизнес-логики, проверку данных и другие связанные операции. Она не взаимодействует напрямую с представлением и контроллером, не имея доступа к их методам, переменным и состояниям. Взаимодействие с моделью осуществляется через заранее определенные интерфейсы, что позволяет полностью отделить ее от пользовательского интерфейса и сделать более универсальной.

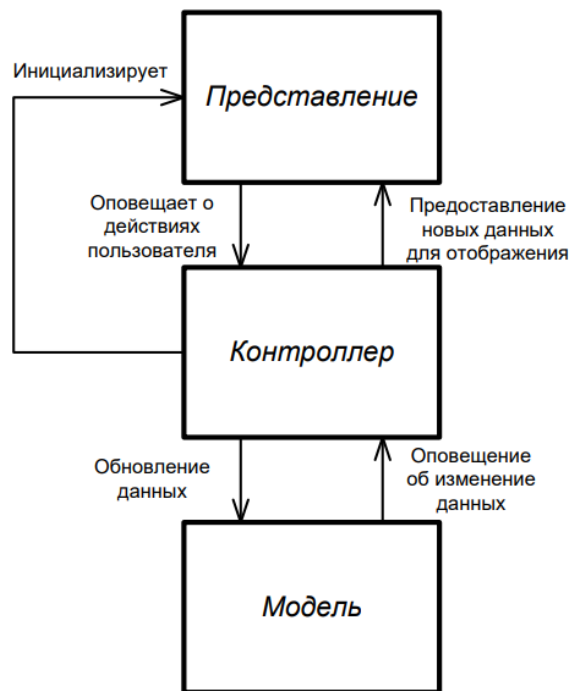


Рисунок 1.4 – Архитектурный паттерн MVC

Представление отображает данные пользователю и предоставляет возможность взаимодействия с приложением через графический интерфейс. В ней определяется внешний вид и макет интерфейса, это происходит в графическом редакторе. Представление не имеет представления о существовании модели и контроллера.

Контроллер является посредником между моделью и представлением. Он получает информацию от представления, обрабатывает ее и вносит соответствующие изменения в модель. Таким образом, контроллер обеспечивает взаимодействие и согласованность между моделью и представлением, делая возможным реализацию функциональности приложения. Чаще всего контроллер является наиболее наполненным исходным кодом элементом приложения.

В контексте разработки Android-приложений паттерн MVC может использоваться для структурирования кода и разделения ответственности между компонентами. Однако в Android этот паттерн встречается реже, так как система активностей и фрагментов уже включает элементы контроллера и представления.

Хотя MVC может быть использован в Android, чаще применяются другие архитектурные паттерны, такие как MVVM (Model-View-ViewModel), которые лучше соответствуют особенностям Android-разработки и обеспечивают лучшую разделяемость кода.

1.5.2 Model-View-ViewModel (MVVM) является схожим с MVC паттерном, используемым для разделения компонентов приложения на три основных уровня: Модель, Представление и Модель Представления (ViewModel).

Одним из недостатков MVVM является некоторая его избыточность для простых приложений, которые имеют небольшое количество логики и данных. MVVM требует больше кода, чем, например, простая парадигма Model-View-Controller. При этом эти недостатки нивелируются при разработке относительно больших приложений, так как MVVM предоставляет огромные возможности для повторного использования различных элементов Представления. На начальных этапах разработка приложения, использующего платформу MVVM, может быть затратной с точки зрения количества кода и времени, однако повторное использование и универсальность стандартизированных представлений упрощают поддержку и расширение проекта в будущем.

Структурная схема взаимодействия компонентов архитектурного паттерна MVVM представлена на рисунке ниже:

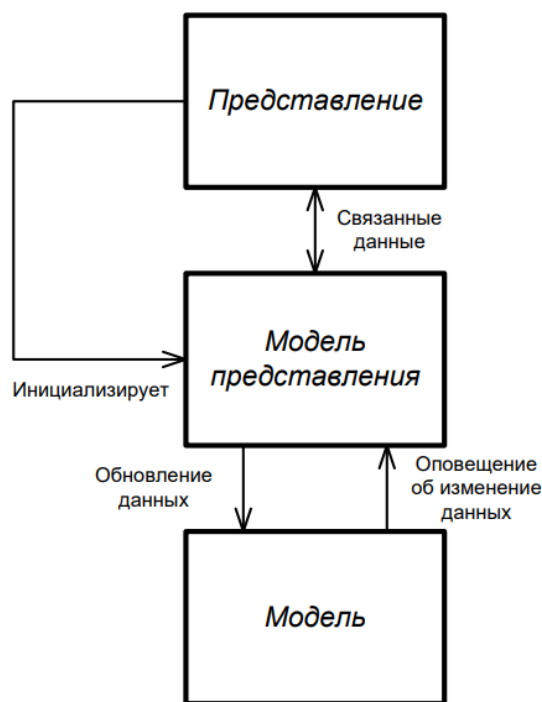


Рисунок 1.5 – Архитектурный паттерн MVVM

Модель, аналогична модели MVC, не имеет информации о представлении и модели представления.

Представление, частично похоже на MVC, отображает данные пользователю и обрабатывает пользовательский ввод. При этом в отличие от MVC, представление напрямую обращается к модели представления, последняя чаще всего является полем представления.

Модель представления является посредником между моделью и представлением и чем-то схожа с контроллером MVC. Модель имеет общее с Представлением данные, которые автоматически обновляются и модели представления и в самом представлении при изменении их как стороны пользователя, так и со стороны модели. В MVC же контроллер при обновлении данных запускает метод обновления Представления. В каком-то смысле

представление совмещает контроллер и представление MVC. Модель представления также может обслуживать несколько представлений одновременно.

1.6 Язык программирования Kotlin

Kotlin – это современный язык программирования со статической типизацией. Разработкой языка занимается компания JetBrains. Он ориентирован на работу с JVM (Java Virtual Machine) и является полностью совместимым с Java, что позволяет использовать библиотеки и фреймворки разработанных для Java без значительных изменений в кодовой базе. Kotlin имеет удобочитаемый и краткий синтаксис, обеспечивает повышенную надежность кода и предлагает удобные конструкции, которые способствуют сокращению числа ошибок в процессе разработки.

Одним из главных достоинств языка программирования Kotlin является его защищенность при работе с null-значениями. В отличие от Java, в Kotlin реализована строгая типизация, исключающая ошибки при попытке обратиться к объекту, значение которого является null. Кроме того, язык поддерживает функциональный подход, позволяя применять анонимные функции, усовершенствованные коллекции и другие удобные механизмы для обработки данных.

Kotlin активно используется в разработке Android-приложений и официально поддерживается Google как основной язык разработки приложения для операционных систем Android.

1.7 Jetpack Compose

Jetpack Compose – это набор инструментов от Google для разработки пользовательских интерфейсов для системы Android, основанный на декларативном подходе, что позволяет описывать пользовательский интерфейс (UI) с помощью функций, а не традиционных XML-разметок. Это упрощает разработку, делает код более читаемым и снижает вероятность ошибок.

Одно из ключевых достоинств Jetpack Compose – его высокая адаптивность и структурированность. Элементы интерфейса, представленные в виде Composable-функций, можно без труда повторно использовать, комбинировать и модифицировать без необходимости заново выстраивать всю структуру экрана. Помимо этого, Jetpack Compose хорошо сочетается с архитектурными подходами, такими как MVVM, что делает его удобным решением для разработки современных Android-приложений.

Compose полностью управляет состоянием UI, что позволяет разрабатывать динамические и отзывчивые интерфейсы без сложных манипуляций с View. Разработчики могут использовать State и MutableState для отслеживания изменений и автоматического обновления экрана. Это значительно упрощает работу с анимациями, пользовательскими действиями

и изменениями данных.

Еще одно важное преимущество – интеграция с существующими Android-компонентами. Jetpack Compose совместим с View-based UI, поэтому его можно постепенно внедрять в проекты, работающие на старых версиях Android. Кроме того, Google активно развивает Compose, добавляя новые возможности и улучшая производительность.

Данный фреймворк предоставляет мощные инструменты для отладки и предпросмотра интерфейсов. Благодаря функции Preview разработчики могут визуализировать компоненты прямо в Android Studio без запуска эмулятора или физического устройства. Это ускоряет итерации: изменения в коде мгновенно отражаются в превью, а поддержка тем, параметров и локалей позволяет тестировать адаптивность интерфейса в разных сценариях. Кроме того, встроенные инструменты вроде Layout Inspector помогают анализировать иерархию элементов, находить перерисовки и оптимизировать производительность, что особенно важно для сложных анимаций и динамических данных.

Jetpack Compose также поддерживает Material Design 3, что позволяет легко использовать современные UI-компоненты и стили. Это ускоряет процесс разработки и делает приложения более визуально привлекательными. В сочетании с другими библиотеками Jetpack (например, Navigation, Paging, Room) Compose становится мощным инструментом для создания надежных и масштабируемых Android-приложений.

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

После анализа предметной области разрабатываемого приложения были сформулированы основные требования, которые необходимо учитывать при его реализации. Поскольку приложение включает обширный функционал, включая работу с базой данных, обработку и отправку данных MQTT-брокеру, работа в локальной сети с использованием HTTP, возникает необходимость в проектировании структурной схемы программного средства.

Структура проекта состоит из следующих блоков:

- блок MQTT клиента;
- блок соединения с брокером;
- блок коммуникации MQTT;
- блок обработки и генерации сообщений;
- блок базы данных;
- блок моделей;
- блок бизнес-логики;
- блок моделей представления;
- блок представления.

Структура системы была организована таким образом, чтобы каждый блок выполнял строго определенную задачу и обеспечивал стабильную работу приложения в целом.

Далее будет рассмотрен принцип работы каждого из выделенных блоков, их основные функции, а также взаимодействие между ними. Взаимосвязь между основными блоками проекта отражена на структурной схеме ГУИР.400201.060 С1.

2.1 Блок MQTT клиента

Блок MQTT клиента отвечает за первичную настройку соединения, обеспечивая корректное взаимодействие с брокером. В рамках этой настройки задаются основные параметры подключения, включая адрес брокера, номер порта, уникальный идентификатор клиента, а также параметры безопасности, такие как логин и пароль.

Дополнительно можно задать интервал Keep Alive, который определяет, как часто клиент будет отправлять сигналы активности брокеру, предотвращая разрыв соединения. Также можно настроить механизм Last Will and Testament (LWT) – заранее заданное сообщение, которое брокер опубликует в случае неожиданного отключения клиента. Это позволяет другим участникам системы получать актуальную информацию о его состоянии.

Кроме того, при настройке соединения можно указать, должна ли сессия быть очищаемой (Clean Session). Если параметр включен, при каждом новом подключении клиент начинает работу с нуля, без сохраненных подписок. Если отключен, брокер запоминает подписки и недоставленные сообщения, обеспечивая более надежную связь при временных разрывах соединения.

При первом подключении системы «умного дома» данные для

подключения вводятся пользователем через блок представления. Вводимые параметры, такие как адрес брокера, логин, пароль и другие настройки, проверяются на корректность и сохраняются в базе данных только при успешном вводе и установлении соединения. Это обеспечивает их последующее использование и предотвращает сохранение некорректных данных. При следующем запуске приложения блок MQTT клиента загружает сохраненные параметры из базы данных и автоматически устанавливает соединение с брокером, исключая необходимость повторного ввода данных. Такой подход повышает удобство работы пользователя и снижает вероятность ошибок, связанных с ручной настройкой.

Благодаря этому механизму обеспечивается не только удобное управление подключением, но и стабильность работы системы, так как параметры сохраняются и могут быть использованы для восстановления соединения в случае сбоя.

2.2 Блок соединения с брокером MQTT

Блок соединения с MQTT-брокером отвечает за установление и поддержание стабильного соединения между клиентом и брокером. Его главная задача – инициировать подключение, следить за его состоянием и восстанавливать связь в случае разрыва.

При запуске системы блок загружает сохраненные параметры подключения, такие как адрес брокера, номер порта, логин, пароль, уникальный идентификатор клиента, а также дополнительные настройки, включая Keep Alive, Clean Session и Last Will and Testament (LWT).

Далее создается экземпляр MQTT-клиента с заданными параметрами, после чего он отправляет запрос на подключение к брокеру, используя аутентификацию по логину и паролю. Если соединение успешно установлено, блок уведомляет другие компоненты системы, в частности блок коммуникации MQTT, который затем подписывается на нужные топики и начинает обмен сообщениями.

Для поддержания соединения используется механизм Keep Alive, который позволяет клиенту периодически отправлять сигналы активности, предотвращая разрыв связи. Если же соединение все же прерывается, блок оперативно обнаруживает это и инициирует процедуру переподключения. При повторных неудачах используется механизм увеличения задержки между попытками восстановления связи, чтобы снизить нагрузку на используемую сеть и брокер.

При каждом успешном подключении блок соединения может сохранять актуальные параметры в базе данных, чтобы использовать их при следующих запусках системы. Это позволяет исключить необходимость повторного ввода данных вручную.

Таким образом, блок соединения с MQTT-брокером выполняет ключевую функцию в системе, обеспечивая надежное подключение и автоматическое восстановление связи при сбоях.

2.3 Блок коммуникации MQTT

Блок коммуникации MQTT играет ключевую роль в системе «умного дома», обеспечивая обмен сообщениями между приложением и удаленным брокером. Его основная задача – установление защищенного соединения, подписка на топики и публикация сообщений, что позволяет передавать данные о состоянии устройств и отправлять им управляющие команды.

После подключения к брокеру с использованием аутентификации блок подписывается на определенные топики, что дает возможность получать актуальную информацию о состоянии устройств. При поступлении сообщения от брокера блок передает его в блок обработки и генерации сообщений, который анализирует содержимое и определяет дальнейшие действия. Если данные содержат информацию о новом состоянии устройства, они передаются в пользовательский интерфейс для отображения актуальной информации.

Кроме приема сообщений, блок коммуникации выполняет отправку данных, обеспечивая управление устройствами. Когда пользователь изменяет параметры в приложении, например, включает освещение или задает температуру, блок обработки формирует соответствующую команду и передает ее в блок коммуникации, который публикует сообщение в нужный топик MQTT. Это позволяет системе обеспечивать двусторонний обмен данными, необходимый для стабильной работы «умного дома».

Для поддержания соединения блок включает механизмы обработки ошибок и восстановления связи. В случае разрыва соединения он уведомляет блок соединения с брокером, который предпринимает попытки восстановления. Также он контролирует целостность передаваемых данных, предотвращая некорректные сообщения и дублирование команд.

2.4 Блок обработки и генерации сообщений

Блок обработки и генерации сообщений играет самую важную роль в системе, обеспечивая корректное взаимодействие между приложением и устройствами «умного дома» через MQTT-протокол. Он выполняет обработку входящих сообщений от брокера, анализируя их содержимое, и формирует исходящие команды для управления устройствами. Основная цель этого блока – преобразование сырых данных в понятный формат, фильтрация лишней информации и передача обработанных данных в соответствующие компоненты системы.

При получении сообщения от MQTT-брокера блок обработки и генерации анализирует его, проверяет на наличие ошибок и определяет, к какому устройству оно относится. Если данные содержат информацию о состоянии устройства, например, текущую температуру, уровень освещенности или статус реле, они передаются в пользовательский интерфейс для отображения актуальной информации.

Кроме обработки входящих сообщений, этот блок также отвечает за генерацию исходящих данных. Когда пользователь в приложении изменяет

параметры работы устройства, блок обработки получает соответствующий запрос, преобразует его в формат MQTT-сообщения и передает в блок коммуникации для отправки брокеру. Это позволяет осуществлять двусторонний обмен данными между клиентом и устройствами «умного дома».

Дополнительно блок включает механизмы фильтрации и валидации данных, предотвращая отправку некорректных команд и обеспечивая корректную работу системы. В случае обнаружения ошибки он может отправлять уведомления в интерфейс пользователя или запускать механизмы повторной передачи сообщения. Все это делает блок обработки и генерации сообщений важным элементом системы, который обеспечивает стабильную и надежную работу обмена данными через MQTT.

2.5 Блок базы данных

Информация, получаемая из блока бизнес-логики после обработки сообщений от брокера, записывается в базу данных, где она структурируется в таблицах, предназначенных для хранения данных о системе и подключенных устройствах. В качестве хранилища используется SQLite с библиотекой Room, обеспечивающей удобную работу с базой данных через объектно-реляционное отображение (ORM).

Блок базы данных отвечает за сохранение параметров подключения к удаленному брокеру сообщений, включая адрес брокера, номер порта, учетные данные, уникальный идентификатор клиента, а также дополнительные настройки, такие как Keep Alive, Clean Session и Last Will and Testament. Это позволяет автоматически восстанавливать соединение после перезапуска приложения.

В таблицах хранятся сведения о подключенных устройствах, включая уникальный сетевой адрес Zigbee, последние полученные значения от MQTT-брокера, пользовательские настройки (например, дружественное имя для устройства), параметры управления (каналы переключения реле, регулировки освещения) и тип устройства. Для ускорения работы приложения и снижения нагрузки на сеть используется кеширование данных, позволяющее загружать актуальные параметры устройств без необходимости частых запросов к брокеру. Механизмы обновления и удаления данных реализованы через Room API с учетом каскадных изменений и целостности связей между таблицами.

2.6 Блок моделей

Блок моделей отвечает за структуру данных, используемых в приложении. Он содержит классы, описывающие подключенные устройства умного дома, а также параметры подключения к MQTT-брокеру. Модели обеспечивают единообразное представление данных, используемых в бизнес-логике, базе данных и пользовательском интерфейсе.

Данные об устройствах включают уникальный идентификатор, тип

устройства, текущее состояние, доступные команды и другие параметры, необходимые для взаимодействия. Также в моделях хранятся сведения о подключении к MQTT-брокеру, такие как адрес сервера, логин, пароль и настройки соединения.

Модели упрощают обработку и передачу данных между различными компонентами системы, обеспечивая их согласованность и целостность. Кроме того, они позволяют эффективно управлять состоянием устройств и взаимодействовать с MQTT-брокером для получения и отправки данных в реальном времени.

2.7 Блок бизнес-логики

Блок бизнес-логики играет ключевую роль в обеспечении стабильной работы приложения, связывая между собой все основные компоненты системы. Он отвечает за обработку данных, управление MQTT-сообщениями, взаимодействие с базой данных и передачу данных в модель представления. Вся логика работы с устройствами умного дома, включая обработку пользовательских команд, обновление состояний и выполнение автоматических сценариев, сосредоточена именно здесь.

Для повышения надежности блок бизнес-логики реализует механизмы обработки ошибок и повторной отправки сообщений в случае сбоев соединения. Он также контролирует целостность данных, предотвращая их дублирование или потерю. Благодаря такому подходу обеспечивается бесперебойная работа системы и точное выполнение всех команд пользователя.

2.8 Блок представлений

Блок представлений отвечает за отображение данных и взаимодействие пользователя с системой «умного дома». В проекте используется Jetpack Compose, который позволяет создавать гибкий и переиспользуемый UI, адаптируемый под разные устройства. Представления работают в тесной связке с ViewModel, которая управляет состоянием, обрабатывает события и передает данные в интерфейс. Это обеспечивает четкое разделение ответственности, где UI остается декларативным, а бизнес-логика сосредоточена во ViewModel.

Одним из важных аспектов является тестируемость представлений. ViewModel можно проверять отдельно, используя инструменты для работы с потоками данных, а UI тестируется с помощью Compose UI Testing. Это позволяет выявлять ошибки на ранних этапах разработки. Кроме того, в Jetpack Compose предусмотрены механизмы обработки ошибок, такие как отображение уведомлений и диалогов, что помогает улучшить пользовательский опыт. Благодаря такому подходу представления остаются простыми, удобными в сопровождении и эффективно взаимодействуют с бизнес-логикой приложения.

2.9 Блок моделей представлений

В Jetpack Compose корневым представлением является композиционная функция (Composable), с которой начинается работа приложения. Это представление содержит ссылки на другие представления, формируя иерархическую структуру интерфейса. При этом всегда возможна навигация назад, а также циклические переходы, когда одно представление может ссылаться на себя.

В Jetpack Compose любое отображаемое на экране содержимое является представлением (Composable). Например, кнопка – это отдельное представление, описанное декларативным способом, а содержащий ее список также является представлением. Таким образом, одно представление может включать в себя множество других, а благодаря механизму параметров при инициализации обеспечивается гибкость и переиспользуемость компонентов.

Представления должны работать одинаково на всех устройствах Android и адаптироваться под разные размеры экранов. Для этого используется механизм адаптивного пользовательского интерфейса, обеспечивающий корректное отображение элементов на смартфонах, планшетах и других устройствах.

Представление взаимодействует с моделью представления (ViewModel), получая из нее данные и отправляя события на обработку. Однако допускается выполнение простых вычислений непосредственно в представлениях. В некоторых случаях, когда представление содержит только статические данные или не требует сложной логики, оно может работать напрямую с моделью данных без использования ViewModel. Это особенно актуально для простых компонентов, таких как кнопки или отдельные элементы списка.

Представления также могут взаимодействовать с Composition Local, получая данные из общего контекста, что упрощает управление состоянием и передачу зависимостей. Функциональность представлений в системе «умного дома»:

- 1 Пользовательский опыт. Важно учитывать удобство использования интерфейса: логичную навигацию, интуитивные элементы управления и четкую обратную связь при взаимодействии с системой.

- 2 Визуализация данных. Jetpack Compose позволяет создавать динамические и адаптивные интерфейсы для отображения данных. Это могут быть графики, диаграммы, карточки или таблицы, отображающие текущее состояние устройств системы «умного дома».

- 3 Элементы управления, такие как Button, Switch, Slider, TextField, позволяют пользователю управлять устройствами «умного дома». Использование состояний (State) и событий (Event) обеспечивает интерактивность и отзывчивость интерфейса.

Таким образом, использование Jetpack Compose в разработке интерфейса приложения «умного дома» позволяет создавать гибкие, адаптивные и удобные представления, обеспечивая качественный пользовательский опыт.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

Функциональное проектирование делает упор на создании корректно работающего приложения, определяя его ключевые возможности и структуру. В этом разделе рассматриваются логические блоки системы, их классы, методы и выполняемые функции. Также представлены диаграмма классов на чертеже ГУИР.400201.060 РР.1 и диаграмма последовательности на чертеже ГУИР.400201.060 РР.2.

3.1 Представления

Разработка приложения производится с использованием фреймворка Jetpack Compose, который основан на декларативном и функциональном подходе к созданию пользовательского интерфейса. В рамках данного подхода представления строятся с помощью функций-компонентов (Composable), которые возвращают описание UI-элементов. Каждый экран приложения представлен в виде набора таких функций, которые принимают параметры, управляют состоянием и отрисовывают элементы интерфейса.

3.1.1 Класс `AppNavHost` отвечает за управление стеком навигации, позволяя пользователю перемещаться между экранами и возвращаться назад без потери состояния. `NavController` используется для выполнения переходов и передачи параметров между экранами, что обеспечивает динамическую и удобную навигацию.

Каждый маршрут внутри `NavHost` связан с соответствующим экраном, а передача `ViewModel` в `composable()` гарантирует, что данные остаются актуальными при смене экранов. Такой подход позволяет эффективно управлять состоянием приложения и упрощает работу с пользовательским интерфейсом.

Основные маршруты:

1 Экран авторизации (`Routes.AUTH_SCREEN`) – первый экран, если пользователь не авторизован. Передает `sensorsViewModel` и `devicesViewModel` в `AuthorizationScreen` для инициализации обработчика сообщений `MQTTMessageHandler` и управления подключением к MQTT-брокеру.

2 Главный экран (`Routes.HOME_SCREEN`) – основной интерфейс пользователя для управления системой «умный дом». Здесь отображаются списки доступных устройств и комнат, полученные из `devicesViewModel` и `roomsViewModel`.

3 Экран деталей устройства (`Routes.DEVICE_DETAILS_SCREEN`) – получает `BackStackEntry` для обработки параметров навигации и передает `devicesViewModel` и `roomsViewModel` для работы с конкретным устройством.

4 Экран деталей комнаты (`Routes.ROOM_DETAILS_SCREEN`) – также

использует `backStackEntry`, передает `navHostController` и `ViewModel`, чтобы загружать информацию о комнате и ее устройствах.

Данный класс обеспечивает централизованное управление навигацией, передавая общие `ViewModel` в экраны, что позволяет эффективно работать с данными без дублирования их экземпляров. Это улучшает производительность приложения и упрощает управление состоянием, так как модели представлений остаются активными при переходах между экранами.

3.1.2 Функция `AuthorizationScreen` отвечает за графический интерфейс экрана авторизации в приложении, предназначенном для управления системой «умный дом». Она предоставляет пользователю возможность ввести учетные данные MQTT-брокера, необходимые для установления соединения с сервером.

Основные компоненты:

1 Элемент `Image` используется для отображения логотипа приложения посередине в верхней части экрана.

2 Форма `BrokerInputForm` включает поля для ввода адреса (URI), порта, имени пользователя и пароля, а также кнопку (`Button`), вызывающую функцию `onAddBroker` для сохранения введенных данных и добавления нового брокера в приложение.

3 Список `BrokerList` отображает последний MQTT-брокер, к которому производилось подключение, с помощью `BrokerItem`, позволяя повторно подключиться (`onLogin`) или удалить запись (`onDelete`).

4 Компонент `BrokerItem` оформляет информацию о брокере в `Card`, включая `Text` для отображения URI, порта, а также (если указаны) логина и замаскированного пароля. Внизу расположены две кнопки: `Button` для подключения к брокеру и `OutlinedButton` для удаления записи.

Данный интерфейс предоставляет возможность добавить информацию для подключения к новому MQTT-брокеру, используя соответствующие учетные данные, а при повторном входе – без ввода данных подключиться к последнему использованному брокеру или выбрать другой из списка сохраненных.

3.1.3 `HomeScreen` является основным экраном приложения, предоставляющий удобный интерфейс для навигации между ключевыми разделами: устройства, комнаты и настройки. Пользователь может легко переключаться между вкладками, что обеспечивает быстрый доступ к необходимым данным без задержек и перезагрузки.

Функциональные особенности:

- горизонтальная навигация реализована через `LazyRow`, где каждая вкладка представлена компонентом `TabButton`, визуально выделяющим активный раздел;

- состояние активной вкладки отслеживается через переменную

`selectedTab`, а `previousTab` используется для анимации плавных переходов;

- динамическое обновление контента осуществляется с помощью `AnimatedContent`, который меняет отображаемый экран в зависимости от текущей вкладки;

- анимации переходов включают эффекты `slideInHorizontally`, `slideOutHorizontally`, `fadeIn` и `fadeOut`, создавая плавный и интуитивный пользовательский опыт;

- адаптация интерфейса под различные размеры экранов: если ширины экрана недостаточно, `LazyRow` позволяет прокручивать вкладки;

- оптимизация производительности, позволяющая избежать лишних перерисовок и снизить нагрузку на устройство, что особенно важно для слабых смартфонов.

Благодаря этим особенностям `HomeScreen` обеспечивает удобную и быструю работу с основным функционалом приложения.

3.1.4 Интерфейс `DevicesScreen` отвечает за отображение списка устройств, доступных в системе, и предоставляет пользователю возможность управления их состоянием.

Основные компоненты экрана:

- `Scaffold` – используется в качестве контейнера, обеспечивающего корректное размещение элементов и управление отступами;

- `Column` – главный контейнер, содержащий все элементы экрана и обеспечивающий вертикальное расположение компонентов;

- вертикальная прокрутка `verticalScroll()` позволяет комфортно просматривать список устройств при их большом количестве;

- `DeviceCard` – карточка устройства, содержащая изображение, название, тип устройства и его текущее состояние с возможностью управления.

Для загрузки данных используется `devicesViewModel.devices`, который с помощью `collectAsState()` получает актуальный список всех устройств. Фильтрация по типу выполняется с помощью функции `getDevicesByTypeFlow(type)`, где в качестве аргумента указывается тип устройства, а актуальные состояния устройств отслеживаются в `DeviceState.devicesData`, что позволяет оперативно обновлять информацию.

Динамическое обновление интерфейса обеспечивается автоматическим реагированием UI на изменения в данных `devicesViewModel.devices`, благодаря чему состояние устройств остается актуальным без необходимости ручного обновления. Текущее состояние каждого устройства извлекается из `deviceState`, что гарантирует мгновенное отображение изменений.

3.1.5 `SettingsScreen` – это функция, описывающая экран настроек, отвечающий за управление дополнительными параметрами и средствами в системе умного дома. В частности, здесь реализовано управление некоторыми настройками шлюза, включая режим поиска устройств.

В данном представлении реализован функционал:

- включение и отключение режима обнаружения новых устройств «умного дома»;

- отображение времени работы режима обнаружения;

- управление настройками подсветки координатора.

Экран настроек состоит из следующих компонентов:

- основной контейнер (`Column`) – обеспечивает расположение элементов в столбец с отступами между ними;

- карточки (`Card`) – используется для отображения различных функциональных компонентов системы;

- переключатель (`Switch`) – позволяет производить управление переключаемых параметров координатора;

- текстовые элементы (`Text`) – отображают информацию о работе координатора.

Функция `onDiscoverySwitchChanged(enabled: Boolean)` отвечает за управление режимом обнаружения новых устройств. При передаче `true` вызывается функция `startDiscovery()`, иначе – `stopDiscovery()`, прекращающая поиск.

Функция `LaunchedEffect(discoveryState)` асинхронно отслеживает изменение состояния поиска (`discoveryState`). Если поиск включен, переменная `remainingTime` инициализируется значением 255, соответствующее 255 секундам, затем в цикле каждую секунду значение уменьшается на единицу функцией управления задержками `delay(1000)`, пока не достигнет нуля. Если поиск отключается, значение `remainingTime` устанавливает значение 0.

3.1.6 Функция `RoomsScreen` реализует функционал управления системой группирования устройств по их территориальному расположению в помещении. Данный пользовательский интерфейс позволяет просматривать список существующих комнат, отображать количество устройств в каждой из них и добавлять новые комнаты в систему.

В данной функции описаны следующие атрибуты:

- `showDialog` – переменная состояния, которая отвечает за отображение диалогового окна добавления новой комнаты: если `true`, отображается диалоговое окно `AddRoomDialog`, иначе компонент скрыт;

- `rooms` – коллекция комнат, полученная из `roomsViewModel`, используемая для построения списка комнат в `LazyColumn`.

Для добавления новой комнаты используется компонент

`FloatingActionButton` для отображения кнопки, которая при нажатии изменяет состояние `showDialog` на `true`, вызывая отображение `AddRoomDialog`.

После ввода названия комнаты и подтверждения действий, вызывается функция `roomsViewModel.addRoom(roomName)`, которая добавляет новую комнату в список. После этого состояние отображения диалогового окна устанавливается в `false`.

3.1.7 Интерфейс `DeviceDetailScreen` представляет собой пользовательский интерфейс для просмотра подробной информации о всех данных предоставляемыми IoT-устройством системы. Он включает в себя элементы для изменения дружественного имени просматриваемого устройства, назначения его в определенную комнату и просмотра текущих параметров устройства.

Атрибуты, описанные в интерфейсе:

- `deviceId` – идентификатор текущего устройства, получаемый из `backStackEntry` в результате навигации между экранами приложения;
- `deviceData` – информация о состоянии текущего устройства, получаемого из объекта, отвечающего за временное хранение показаний устройств;
- `rooms` – список всех комнат, для возможности переназначения отношения к какой-либо комнате;
- `expanded` – переменная состояния для управление выпадающим списком при выборе комнат;
- `showEditDialog` – состояние, управляющее отображением диалогового окна для редактирования дружественного имени устройства;
- `selectedRoom` – переменная состояния для хранения информации о выбранной комнате.

Основные графические компоненты интерфейса:

- `DeviceTitle` – компонент, отображающий текущее имя устройства и кнопку для редактирования имени устройства;
- `EditDialog` – всплывающее окно для редактирования информации;
- `ExposedDropDownMenuBox` – это компонент, реализующий раскрывающееся меню для выбора комнат;
- `ExposedDropDownMenu` – компонент выпадающего списка для выбора комнаты;
- `DropDownMenuItem` – элементы списка выпадающего меню;
- `DeviceDetails` – `composable`-функция, используемая для отображения текущего состояния устройства.

Для реализации функционала в данном интерфейсе используются следующие функции:

- `onEditClick` – `callback` функция для открытия диалогового окна `EditDialog` при изменении имени устройства;
- `updateDeviceName(deviceId, newName)` – функция, которая

вызывается при подтверждении нового имени устройства, обновляя его в `DevicesViewModel`;

– `assignRoomToDevice(deviceId, roomId?)` – функция, которая выполняет привязку устройства к выбранной комнате или снимает ее при выборе опции «Selected Room».

Таким образом, интерфейс `DeviceDetailScreen` предоставляет пользователю удобный способ просмотра данных об устройстве, а также позволяет изменять его имя и привязку к комнате.

3.2 Модели представлений

Взаимодействие пользовательского интерфейса с данными и бизнес-логикой осуществляется с использованием архитектурного компонента `ViewModel`. `ViewModel` отвечает за хранение и управление состоянием экрана, обеспечивая его устойчивость к изменениям конфигурации, таким как поворот экрана. Он предоставляет данные в `Composable`-функции представлений через механизмы хранения состояний, а также содержит методы для обработки пользовательских действий.

Для каждого представления реализован соответствующий фабричный класс `ViewModelProvider.Factory`, который отвечает за создание экземпляров классов с передачей необходимых зависимостей, обеспечивая инъекцию зависимостей и соблюдение принципов инверсии управления.

3.2.1 Класс `AuthorizationViewModel` реализует логику управления данными и состоянием экрана авторизации.

Основные атрибуты класса:

`1 _brokers` – это приватное свойство типа `MutableState<List<Broker>>`, хранящее текущий список брокеров. Оно используется для внутреннего управления состоянием и позволяет изменять данные внутри `ViewModel`.

`2 brokers` – публичное свойство типа `State<List<Broker>>`, предоставляющее доступ к списку брокеров только для чтения. Оно обеспечивает реактивное обновление UI при изменении данных (например, при добавлении или удалении брокеров).

Методы класса:

`1 addBroker()` асинхронно добавляет нового MQTT-брокера в локальную базу данных и обновляет UI. Метод выполняется в корутине через `viewModelScope.launch`, создает объект `Broker` с переданными `serverUri`, `serverPort`, `user` и `password` (поддерживая анонимное подключение), затем сохраняет его в базе данных с помощью `brokerDao.insert()`, после этого вызывается метод `loadBrokers()`, чтобы загрузить обновленный список брокеров, что автоматически обновляет UI благодаря реактивному состоянию `_brokers`.

`2 deleteBroker()` асинхронно удаляет информацию о брокере из

базы данных и обновляет UI. Он выполняется в корутине, сначала отключает MQTT-клиент с помощью `disconnect()`, затем удаляет брокера через `deleteBroker(broker)`, после чего вызывает `loadBrokers()`.

3 `handleLogin()` устанавливает текущий брокер, создает MQTT-клиент и выполняет подключение. Он обновляет `BrokerState` с `broker.id`, затем переинициализирует клиент `MQTTClient` с указанным брокером и обработчиком сообщений. Если подключение успешно, вызывается функция `onSuccess()`. Далее асинхронно с использованием `viewModelScope.launch(Dispatchers.IO)` клиент подписывается на необходимые для работы топики.

3.2.2 `DevicesViewModel` это модель представления, которая служит для управления устройствами умного дома, обеспечивая их загрузку, обновление и взаимодействие с MQTT брокером.

Атрибут `_devices` – это `MutableStateFlow`, хранящий список устройств и используемый для управления их состоянием. Он инициализируется пустым списком и изменяется внутри `viewModelScope.launch`, что позволяет выполнять загрузку устройств из базы данных в фоновом режиме и не блокировать основной поток выполнения программы. Для получения данных используется атрибут `devices` типа `StateFlow`.

Методы, используемые для работы с данными:

1 `loadDevices(brokerId: Int)` – асинхронно загружает устройства, связанные с брокером, подписываясь на `Flow` из базы данных. При обновлении данных обновляет `_devices`, что автоматически отражается в пользовательском интерфейсе.

2 `getDeviceIdByIeeeAddr(addr: String, callback: (Int?) -> Unit)` – получает ID устройства по его адресу и передает результат в `callback`.

3 `getDevicesByTypeFlow(type: String)` – возвращает `StateFlow` с устройствами указанного типа. Фильтрует `_devices` на основе команд, полученных через `Flow`.

4 `getDevicesByRoomIdFlow(roomId: Int)` – аналогично предыдущему методу фильтрует устройства по их типу, но дополнительно учитывает идентификатор комнаты `roomId`, обновляя `resultFlow`.

5 `addDeviceIfNotExists(device: Device)` – добавляет устройство в локальное хранилище, если его там нет. Проверяет наличие устройства по его физическому адресу `ieeeAddr`, при отсутствии добавляет новую запись с информацией об устройстве.

6 `updateDeviceName(deviceId: Int, newName: String)` – изменяет имя устройства, обновляя его в базе и списке `_devices`.

7 `addCommandIfNotExists(command: Command)` – добавляет команду в базу данных, если ее еще нет, проверяя по `commandTopic`.

8 `sendCommandToMqtt(topic: String, command: String)` – публикует команду управления устройством в заданный топик.

9 `onToggle(topic: String, state: Boolean)` – функция для обработки управляющих команд переключателей.

10 `onValueChange(topic: String, value: Int)` – функция для обработки управляющих команд устройств с регулируемыми параметрами в диапазоне от 0 до 100.

11 `onSelectChange(topic: String, option: String)` – функция для обработки управляющих команд устройств с выбором доступных опций.

12 `assignRoomToDevice(deviceId: Int, roomId: Long?)` – обновляет `roomId` устройства в базе.

3.3 Модели

В данном подразделе представлены таблицы моделей данных и связи между ними. Типы данных указаны в нотации Kotlin, как они описаны в исходном коде. Опциональность значений обозначается вопросительным знаком после типа данных.

Для создания сущности и соответствующей ей таблицы в базе данных используется аннотация `@Entity`. В ее параметрах указывается служебная информация для библиотеки Room, такая как:

- `tableName` – название таблицы;
- `foreignKeys` – внешний ключ, определяющий связи между таблицами;
- `indices` – список индексов, ускоряющих поиск данных при большом объеме информации.

Описание модели состоит из двух частей:

- описание атрибутов таблицы – перечень столбцов и их характеристики;
- описание связей таблицы – указание внешних ключей, индексов и каскадных правил.

3.3.1 Класс `Device` представляет собой сущность базы данных, представленной в таблице `devices`, предназначенную для хранения информации об устройствах в системе «умного дома».

Описание атрибутов таблицы:

1 `id: Int` – первичный ключ таблицы, используемый для уникальной идентификации каждой записи. Генерируется автоматически при создании новой записи благодаря аннотации `@PrimaryKey(autoGenerate = true)`.

2 `ieeeAddr: String` – уникальный физический адрес устройства, используемый для его идентификации в сети. Как правило, соответствует MAC-адресу или другому идентификатору, назначенному на аппаратном

уровне.

3 `friendlyName: String` – читаемое название устройства, которое пользователь может задать вручную. Это название отображается в интерфейсе и упрощает взаимодействие с устройством, так как заменяет сложные идентификаторы удобным именем.

4 `modelId: String` – уникальный идентификатор модели устройства, назначаемый производителем. Используется для определения типа устройства, его функциональных возможностей и совместимости с другими компонентами системы.

5 `topic: String` – MQTT-топик, в который публикуются сообщения о состоянии устройства и через который осуществляется его управление. Позволяет организовать обмен данными между устройством и сервером, используя брокер MQTT.

6 `roomId: Long?` – идентификатор комнаты, в которой находится устройство. Если значение `null`, это означает, что устройство не привязано ни к одной конкретной комнате. Используется для логической группировки устройств по помещениям.

7 `brokerId: Int` – идентификатор MQTT-брокера, через которого осуществляется обмен данными. Позволяет системе поддерживать работу с несколькими брокерами и маршрутизировать сообщения к нужному серверу.

Связи между таблицами определены с помощью внешних ключей. Поле `brokerId` связано с `id` таблицы `Broker`, и при удалении брокера все связанные устройства также удаляются (с параметром `onDelete = ForeignKey.CASCADE`). Параметр `roomId` связан с `id` таблицы `RoomEntity`, и, если комната удаляется, у всех устройств, связанных с этой комнатой, значение `roomId` становится `null` (с параметром `onDelete = ForeignKey.SET_NULL`).

Объект компаньон в классе `Device` используется для создания нового устройства с автоматическим формированием топика. Топик генерируется на основе физического адреса устройства в формате «zigbee/0x<ieeeAddr>».

Кроме того, в таблице используется индексация по полю `roomId`, что значительно ускоряет поиск устройств, принадлежащих определенной комнате.

3.3.2 Класс `Broker` представляет собой сущность базы данных, отображаемую в таблице `brokers`, предназначенную для хранения информации о MQTT-брокерах, используемых в системе «умного дома».

Поля класса `Broker`:

- `id: Int` – первичный ключ таблицы с автоматической генерацией;
- `serverUri: String` – URI MQTT-брокера, который используется для подключения к нему;
- `serverPort: Int` – порт, через который осуществляется подключение к брокеру;

– `user: String?` – имя пользователя для аутентификации при подключении к брокеру (может быть `null`, если аутентификация не требуется);

– `password: String?` – пароль пользователя для подключения к брокеру (может быть `null`, если аутентификация не требуется).

Поле `id` используется в таблице `devices` как внешний ключ (`brokerId`). Если брокер удаляется, все связанные с ним устройства также удаляются, что обеспечивается параметром `onDelete = ForeignKey.CASCADE` в определении внешнего ключа.

В таблице `brokers` можно хранить несколько брокеров, что позволяет системе «умного дома» работать с разными MQTT-серверами.

3.3.3 Класс `Command` представляет собой сущность базы данных, отображаемую в таблице `commands`, предназначенную для хранения информации о командах управления устройствами в системе «умного дома».

Атрибуты таблицы:

– `id: Int` – первичный ключ таблицы;

– `deviceId: Int` – внешний ключ, связывающий команду с устройством из таблицы `devices`. При удалении устройства все связанные с ним команды также удаляются;

– `commandTopic: String` – MQTT-топик, в который отправляются команды для управления устройством;

– `payloadOn: String?` – полезная нагрузка для включения устройства (может быть `null`, если команда не предназначена для переключателя);

– `payloadOff: String?` – полезная нагрузка для выключения устройства (может быть `null`, если команда не предназначена для переключателя);

– `options: Map<String, String>?` – список доступных опций для команд выбора (например, вариантов переключения режимов устройства), для хранения в базе данных с использованием `MapTypeConvertor`;

– `commandTemplate: String?` – шаблон команды, который может быть использован для формирования сложных сообщений в MQTT (может быть `null`, если команда не требует шаблонизации);

– `commandType: String` – тип команды (например, `switch`, `select`, `custom`), определяющий логику ее обработки.

Поле `deviceId` связано с идентификатором таблицы `devices`. При удалении устройства все связанные с ним команды автоматически удаляются (`onDelete = ForeignKey.CASCADE`).

Таблица `commands` позволяет хранить команды управления устройствами умного дома, поддерживая как простые команды (например, включение и выключение устройств), так и более сложные (например, выбор режима работы устройства).

3.3.4 Класс `RoomEntity` представляет собой сущность, информация о которой хранится в таблице `rooms`.

Поля сущности:

- `id: Long` – идентификатор, выступающий в роли первичного ключа таблицы;

- `name: String` – название комнаты, которое задается пользователем для удобного восприятия в системе.

Поле `id` используется в таблице `devices` как внешний ключ (`roomId`). Если комната удаляется, у всех связанных с ней устройств значение `roomId` принимает значение `null` (`onDelete = ForeignKey.SET_NULL`), что позволяет избежать удаления устройств из системы при удалении комнаты.

Таблица `rooms` позволяет группировать устройства по местоположению в помещении.

3.4 Хранение данных

В приложении используется локальная база данных на основе `SQLite`, управляемая через библиотеку `Room`. Это позволяет эффективно хранить и обрабатывать информацию обеспечивая удобный доступ к данным и их целостность.

Взаимодействие с хранимой информацией осуществляется через DAO-интерфейсы. Помимо постоянного хранения данных в базе, в приложении используются `singleton`-объекты, отвечающие за временное хранение состояний системы. Их основная задача – обеспечение быстрого доступа к актуальной информации без необходимости выполнения запросов к базе данных. Эти состояния существуют только во время работы приложения и не сохраняются в базе, поскольку их значения могут динамически изменяться в зависимости от текущего сеанса работы системы.

3.4.1 Класс `RoomLocalDatabase` является основной точкой доступа к базе данных `SQLite`, реализованной с использованием библиотеки `Room`.

Атрибуты класса базы данных задаются в качестве параметров аннотации `@Database`, в которой указываются:

- `entities` – список моделей используемых в базе данных (`Broker`, `Device`, `Command`);

- `version` – версия базы данных;

- `exportSchema` – флаг экспорта схемы, установленный в значение `false`, что означает создание таблиц производится на основе моделей, описанных в приложении.

Методы класса:

- `brokerDAO()`, `deviceDAO()`, `commandDAO()`, `roomEntityDAO()` – абстрактные методы, предоставляющие доступ к DAO-объектам для работы с таблицами `brokers`, `devices`, `commands` и `rooms` соответственно;

– `getInstance()` – статический метод, реализующий порождающий паттерн Singleton для создания и получения единственного экземпляра базы данных, используя `Room.databaseBuilder`.

3.4.2 Класс `BrokerState` определен с помощью ключевого слова `object`, что делает его singleton-объектом – он создается один раз и существует в единственном экземпляре в рамках всего приложения.

Этот класс предназначен для хранения идентификатора, текущего MQTT-брокера, который используется в данный момент.

Основные компоненты данного класса:

– `_brokerId` – приватное поле типа `MutableStateFlow<Int?>`, хранящее идентификатор брокера и позволяющее его изменять внутри методов класса;

– `brokerId` – публичное свойство типа `StateFlow<Int?>`, предоставляющее неизменяемый поток данных, содержащий идентификатор текущего брокера;

– `setBrokerId(id: Int)` – метод, обновляющий идентификатор брокера, устанавливая новое значение в `_brokerId`.

3.4.3 Класс `DeviceState` так же определен с помощью ключевого слова `object`, что делает его singleton-объектом.

Данный класс используется для хранения состояний устройств «умного дома», содержащего актуальные данные, передаваемые устройствами.

Атрибуты класса:

– `_devicesData` – приватное поле типа `MutableStateFlow<Map<Int, Map<String, Any>>>`, хранящее данные всех устройств, где ключом верхнего уровня является `deviceId`, а значением – параметры устройств, представленных в виде структуры `Map`;

– `devicesData` – публичное свойство типа `StateFlow<Map<Int, Map<String, Any>>>`, предоставляющее неизменяемый поток данных с текущими состояниями устройств;

– `updateDeviceData(deviceId: Int, payload: String)` – метод, обновляющий данные конкретного устройства, принимая идентификатор устройства `deviceId` и строку в формате JSON `payload`;

– `parseJson(json: String)` – метод, преобразующий JSON-строку в структуру `Map<String, Any>` используемую для хранения состояния устройства;

– `getDeviceValue(deviceId: Int, key: String)` – метод, возвращающий конкретное значение из состояния устройства по его идентификатору и заданному ключу `key`.

3.4.4 Класс `DiscoveryState` определен ключевым словом `object`. Данный класс используется для управления процессом обнаружения и

подключения устройств в систему «умный дом». Он содержит методы для запуска и остановки процесса обнаружения, а также отслеживает его текущее состояние.

Атрибуты класса:

1 `_isDiscoveryActive` – поле `MutableStateFlow<Boolean>`, которое хранит текущее состояние процесса обнаружения устройств. Если значение равно `true`, это означает, что процесс обнаружения активен, если `false` — процесс неактивен.

2 `isDiscoveryActive` – атрибут типа `StateFlow<Boolean>`, которое предоставляет доступ к текущему состоянию процесса обнаружения устройств. Это свойство является неизменяемым, что позволяет внешним компонентам подписываться на изменения состояния, но не изменять его напрямую.

3 `resetJob` – приватное поле типа `Job?` которое хранит ссылку на корутину, отвечающую за автоматическое завершение процесса обнаружения через определенный промежуток времени. Это поле может быть `null`, если процесс обнаружения неактивен.

Методы, описанные в классе:

1 Метод `startDiscovery()` устанавливает значение `true` для атрибута `_isDiscoveryActive`, что указывает на начало процесса обнаружения устройств. Он отправляет MQTT-сообщение в топик `DISCOVERY_TOPIC` со значением `true`, чтобы разрешить устройствам Zigbee подключаться к сети. Если существует предыдущая корутина, она отменяется. Затем запускается новая корутина, которая через `DISCOVERY_TIME` автоматически завершает процесс обнаружения, устанавливая `_isDiscoveryActive` в `false`.

2 `stopDiscovery()` устанавливает значение `_isDiscoveryActive` в `false`, что указывает на завершение процесса обнаружения. Он отправляет MQTT-сообщение в аналогичный топик со значением `false`, чтобы запретить устройствам Zigbee подключаться к сети. Если текущая корутина существует, она отменяется.

3.4.5 Класс `DeviceDAO` представляет собой DAO-интерфейс (Data Access Object) для работы с таблицей устройств в базе данных. Он определен с помощью аннотации `@Dao` и содержит методы для взаимодействия с данными.

Основные методы:

– `getDevicesByBroker(brokerId: Int)` – получает список информации об устройствах, связанных с указанным брокером используя его ID;

– `getDevicesByBrokerFlow(brokerId: Int)` – возвращает поток (`Flow`), содержащий информацию об устройствах для заданного брокера по его идентификационному номеру;

- `getAllDevices()` – загружает информацию о всех устройствах, хранящихся в базе данных;
- `getDeviceByIeeeAddr(ieeeAddr: String)` – ищет устройство с использованием его физического адреса (IEEE адрес), возвращает `null`, если устройство не найдено;
- `getDeviceById(deviceId: Int)` – получает устройство по его `deviceId`;
- `insertDevice(device: Device)` – добавление устройства в базу данных, игнорируя конфликты. Возвращает `deviceId` добавленной записи или `-1`, если устройство уже существует;
- `getDevicesByRoomIdFlow(roomId: Int)` – возвращает поток `Flow`, содержащий список устройств, принадлежащих указанной комнате (`roomId`);
- `updateDevice(device: Device)` – обновляет информацию об устройстве в базе данных;
- `getDeviceCountForRoom(roomId: Long)` – получает количество устройств, связанных с заданной комнатой (`roomId`).

3.4.6 Класс `BrokerDAO` является DAO-интерфейсом для взаимодействия приложения с таблицей, содержащей информацию о брокерах.

Методы класса:

- `insert(broker: Broker)` – добавляет брокера в базу или обновляет его при конфликте;
- `getAllBrokers()` – получает список всех брокеров;
- `getLastBroker()` – получает последнего добавленного брокера (по убыванию ID), возвращает `null`, если брокеров нет;
- `deleteBroker(broker: Broker)` – удаляет запись содержащую информацию об указанном брокере.

3.4.7 Класс `CommandDAO` представляет собой интерфейс, обеспечивающий доступ к данным команд, хранящимся в базе.

Методы, реализуемые в данном интерфейсе:

- `getCommandByCommandTopic(commandTopic: String)` – получает команду по заданной теме (`commandTopic`), возвращает `null`, если такой команды нет;
- `getSwitchCommandByDeviceId(deviceId: Int)` – ищет команду типа «switch» по `deviceId`, возвращает `null`, если команда не найдена;
- `insertCommand(command: Command)` – добавляет новую команду в базу или обновляет существующую при конфликте;
- `getCommandsByTypeFlow(type: String)` – возвращает `Flow` со списком команд заданного типа.

3.4.8 `RoomEntityDAO` – класс, предназначенный для управления данными о комнатах в базе данных.

Доступные функции для работы с данными:

- `getAllRooms()` – возвращает поток (`Flow`) со списком всех комнат;
- `insertRoom(room: RoomEntity)` – добавляет новую комнату или обновляет существующую при конфликте.

3.4.9 Класс `MapTypeConverter` используется в базе данных `Room` для преобразования объектов типа `Map<String, String>` в строковый формат `JSON` и обратно. Это необходимо, так как `SQLite`, на котором основана `Room`, не поддерживает хранение сложных структур данных, таких как `Map`. `SQLite` работает с примитивными типами, поэтому для сохранения структур ключ-значение их нужно преобразовывать в строку.

В данном классе используется класс `Gson` для сериализации (`fromMap`) и десериализации (`toMap`) `Map<String, String>` в `JSON`. Метод `fromMap` превращает карту в строку `JSON`, которая может быть сохранена в базе данных как `TEXT`. Метод `toMap` выполняет обратную операцию — восстанавливает карту из `JSON`, что позволяет использовать ее в коде как обычный объект `Map<String, String>`. Такой подход позволяет удобно работать с `Map` в `Room`, несмотря на ограничения `SQLite`.

3.5 MQTT и взаимодействие с брокером

В данном разделе рассматривается организация взаимодействия с `MQTT`-брокером в разрабатываемом приложении.

3.5.1 Интерфейс `MQTTMessaging` определяет методы `subscribe` и `publish`, которые используются клиентом для взаимодействия с брокером. Метод подписки `subscribe` принимает в качестве параметра `MQTT`-топик, на который необходимо подписаться.

Метод для публикации сообщений, в свою очередь, отправляет заданное сообщение `payload` в указанный `MQTT`-топик с возможностью выбора уровня качества обслуживания `qos` и флага сохраненного сообщения `retained`.

Реализуя этот интерфейс, класс описывает логику подписки на `MQTT`-топики и отправки сообщений, обеспечивая взаимодействие с брокером.

3.5.2 Интерфейс `MQTTConnection` определяет методы `connect` и `disconnect`, используемые для управления подключением к `MQTT`-брокеру.

Метод `connect` выполняет установку соединения с брокером и возвращает `Boolean`, указывающий на успешность подключения.

Метод `disconnect`, в свою очередь, разрывает текущее соединение с брокером.

Реализуя этот интерфейс, класс описывает логику установки и завершения соединения, обеспечивая стабильное взаимодействие с MQTT-брокером.

3.5.3 Класс `MQTTClient` представляет собой реализацию MQTT-клиента для взаимодействия с брокером. Данный класс реализует интерфейсы `MQTTMessaging` и `MQTTConnection`. Он позволяет настроить соединение с учетом параметров аутентификации, обрабатывать входящие сообщения через callback-функции и управлять подписками.

Атрибуты класса:

- `mqtClient: MqttClient?` – объект клиента MQTT;
- `broker: Broker?` – содержит информацию о брокере;
- `messageHandler: MQTTMessageHandler?` – обработчик

входящих сообщений.

Методы, реализованные в классе:

- `initialize(Broker, MQTTMessageHandler)` – выполняет инициализацию клиента, устанавливая параметры подключения к брокеру и назначая обработчик сообщений;
- `reinitialize(Broker, MQTTMessageHandler)` – завершает текущее соединение, перенастраивает параметры подключения и повторно инициализирует клиента с новым брокером;
- `connect(): Boolean` – выполняет подключение к брокеру с учетом учетных данных. Возвращает `true` в случае успешного соединения и `false` при возникновении ошибки;
- `subscribe(topic: String)` – подписывает клиента на указанный MQTT-топик и передает полученные сообщения в `messageHandler`;
- `publish(topic: String, payload: String, qos: Int, retained: Boolean)` – отправляет сообщение в указанный топик с заданным уровнем качества обслуживания (QoS) и возможностью хранения последнего значения (`retained`);
- `disconnect()` – разрывает соединение с MQTT-брокером и освобождает ресурсы.

3.5.4 Класс `MQTTMessageHandler` отвечает за обработку входящих MQTT-сообщений и взаимодействие с моделями данных приложения.

Метод `handleMessage(topic: String, payload: String)` является основной функцией для обработки входящих MQTT-сообщений. Анализирует `topic` сообщения и вызывает соответствующий обработчик в зависимости от его типа. Это позволяет корректно маршрутизировать данные в системе.

Метод `handleDeviceStateMessage(topic: String, payload: String)` отвечает за обновление состояния устройств. Разбирает данные, полученные в сообщении, и обновляет соответствующую запись в модели состояния устройств. Может использоваться для синхронизации данных между физическими устройствами и интерфейсом пользователя.

Метод `handleDeviceCommandMessage(topic: String, payload: String)` предназначен для обработки команд, отправленных устройствам. Проверяет, существует ли уже такая команда в базе данных, и, если нет, сохраняет ее. Это позволяет избежать дублирования команд и обеспечивает корректное управление устройствами.

Метод `handleDeviceListMessage(payload: String)` выполняет обработку JSON-данных, содержащих список доступных устройств. Если какое-либо устройство еще не добавлено в систему, оно записывается в базу данных. Этот механизм позволяет автоматически обновлять список доступных устройств в приложении.

Данный класс обеспечивает корректную маршрутизацию MQTT сообщений и взаимодействие с внутренними структурами данных приложения.

3.5.5 Класс `Topics` содержит predefined константы, представляющие MQTT-топики, с которыми взаимодействует приложение. Эти топики используются для обмена данными с MQTT-брокером и управления устройствами умного дома.

Атрибуты:

- `DISCOVERY_TOPIC` – топик для включения режима сопряжения новых Zigbee-устройств;
- `DEVICE_STATE_TOPIC` – шаблон топиков, содержащих информацию о состоянии устройств;
- `DEVICE_COMMANDS_TOPIC` – базовый топик для получения информации о командах для управления устройствами;
- `DEVICE_LIST_TOPIC` – топик для получения списка доступных устройств.

3.6 Вспомогательные компоненты системы

3.6.1 В файле-разметке `AndroidManifest.xml` заданы необходимые разрешения, которые приложение запрашивает перед использованием определенных функций. Ниже приведен список указанных разрешений и их назначение:

1 `INTERNET` – разрешает приложению доступ в интернет. Требуется для работы с сетевыми запросами по протоколу MQTT.

2 `ACCESS_NETWORK_STATE` – позволяет приложению запрашивать текущее состояние о подключении к сети.

3 `POST_NOTIFICATIONS` – предоставляет приложению доступ для

отправки уведомлений.

Так же в данном файле задаются ключевые параметры приложения. Устанавливаются иконки, название и тема, что определяет внешний вид и стиль интерфейса. Указываются настройки резервного копирования, позволяющие сохранять данные при переустановке или переносе на другое устройство.

3.6.2 Класс `Logger` предоставляет удобный способ ведения логов в приложении. Реализован с использованием порождающего паттерна `Singleton` и использует `Log.i()` для вывода информационных сообщений.

Функции для логирования:

- `log(className: KClass<*>, message: String)` – логирует сообщение, используя имя класса в качестве тега;
- `log(tag: String, message: String)` – логирует сообщение с указанным тегом.

Этот класс упрощает отладку, позволяя быстро идентифицировать источник сообщений в логах.

3.6.3 Класс `Constants` представляет собой объект, содержащий константы, используемые в приложении.

Основные значения, представленные в виде констант:

- типы устройств: `SWITCH_TYPE`, `SELECT_TYPE`, `DIMMER_TYPE`;
- список вкладок интерфейса: `TABS_LIST` включает «Devices», «Rooms», «Settings»;
- настройки для системы обнаружения устройств: `DISCOVERY_ENABLE`, `DISCOVERY_DISABLE`, `DISCOVERY_TIME`.

Этот класс обеспечивает централизованное хранение значений, упрощая доступ и поддержку кода.

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

Разработка данного приложения производилась в интегрированной среде разработки Android Studio с использованием языка программирования Kotlin и фреймворка для разработки графического интерфейса Jetpack Compose.

В приложении реализуется следующий функционал:

- управление соединением с MQTT-брокером;
- обработка данных, полученных от MQTT-брокера;
- отправка пользовательских данных в MQTT-брокер;
- управление модулем светодиодной подсветки;
- отображение общего списка устройств системы;
- отображение списка комнат системы;
- добавление новых комнат в систему;
- управление режимом обнаружения и подключения новых устройств;
- отображение полной информации об устройстве;
- привязка устройства к определенной комнате.

4.1 Управление соединением с MQTT-брокером

За функционал управления соединением отвечает пользовательский интерфейс `AuthorizationScreen` и класс клиента MQTT-брокера `MQTTClient`.

Установка нового соединения состоит из двух основных этапов:

- получение данных об используемом брокере;
- создание экземпляра клиента и подключение к брокеру.

Данные об используемом брокере могут быть получены несколькими способами:

- посредством пользовательского ввода;
- из сохраненных данных в базе данных.

4.1.1 В первом случае это может быть ввод данных об подключении через форму `BrokerInputForm` на экране `AuthorizationScreen`, где пользователь указывает адрес сервера, порт брокера, имя пользователя и пароль (если на сервере настроена аутентификация пользователей). Введенные данные должны пройти проверку на валидность, чтобы убедиться, что все параметры корректны (например, правильный формат адреса и порта). После успешной валидации и подключения к брокеру, данные сохраняются в базу данных, чтобы при следующих запусках приложения пользователь мог автоматически подключиться, без необходимости повторного ввода данных. Валидация и добавление информации в базу данных представлена в функции, описанной ниже:

```
fun addBroker(serverUri: String, serverPort: Int, user:
String?, password: String?) {
    if (serverUri.isBlank()) return
```



```

        val uriPattern = "^(http|https|mqtt|mqtts)://[a-zA-Z0-9.-]+(:[0-9]+)?".toRegex()
        if (!uriPattern.matches(serverUri)) return
        if (serverPort !in 1..65535) return

        viewModelScope.launch {
            val broker = Broker(
                serverUri = serverUri,
                serverPort = serverPort,
                user = user,
                password = password
            )
            brokerDao.insert(broker)
            loadBrokers()
        }
    }
}

```

Метод `brokerDao.insert(broker)` реализован следующим образом:

```

@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun insert(broker: Broker)

```

В данном методе происходит вставка нового объекта `Broker` в базу данных. `@Insert(onConflict = OnConflictStrategy.REPLACE)` – данная аннотация означает, что если в таблице уже существует запись с таким же первичным ключом, она будет заменена новой. Это поведение удобно для обновления настроек брокера без необходимости выполнения дополнительной операции удаления. Так же метод помечен ключевым словом `suspend`, что означает его асинхронное выполнение. Для вызова этого метода требуется использовать корутину.

4.1.2 Если пользователь ранее уже вводил корректные данные для подключения, приложение может автоматически извлечь эти данные из базы данных и отобразить на экране авторизации в виде карточки с информацией о подключении.

```

@Composable
fun BrokerList(
    brokers: List<Broker>,
    onDelete: (Broker) -> Unit,
    onLogin: (Broker) -> Unit
) {
    brokers.lastOrNull()?.let { broker ->
        Text("Recently used broker:", style =
            MaterialTheme.typography.labelMedium)
        Spacer(modifier = Modifier.height(1.dp))
        BrokerItem(
            broker = broker,
            onDelete = { onDelete(broker) },
            onLogin = { onLogin(broker) }
        )
    }
}

```

В этом случае, пользователь не будет вынужден снова вводить

параметры вручную, что значительно ускоряет процесс подключения. Приложение предоставит пользователю информацию о последнем использованном брокере, и при наличии интернет-соединения по нажатию кнопки входа в систему произойдет подключение к брокеру.

При нажатии кнопки входа вызывается следующая функция:

```
fun handleLogin(broker: Broker, onSuccess: () -> Unit) {
    BrokerState.setBrokerId(broker.id)
    val mqttClient = MQTTClient.reinitialize(broker,
messageHandler)
    val isSuccess = mqttClient.connect()
    if (isSuccess) {
        onSuccess()
        viewModelScope.launch(Dispatchers.IO) {
            mqttClient.subscribe(Topics.SUBSCRIBE_DEVICE_LIST_TOPIC)
            mqttClient.subscribe(Topics.SUBSCRIBE_COMMANDS_TOPIC)
            mqttClient.subscribe(Topics.SUBSCRIBE_DEVICE_STATE_TOPIC)
            mqttClient.subscribe(Topics.SUBSCRIBE_LED_STATE_TOPIC)
        }
    }
}
```

В этом методе идет проверка подключения к брокеру, и в случае успешного подключения производится асинхронная подписка на ключевые темы для корректного управления и мониторинга системой «умный дом» посредством использования мобильного приложения. Подписка подразумевает собой установку соответствующего callback метода, который автоматически вызывается при получении сообщений по указанным темам. Этот метод обрабатывает входящие данные и передает их в систему для дальнейшей обработки или отображения в интерфейсе мобильного приложения.

4.2 Обработка данных полученных от MQTT-брокера

После принятия сообщения данные следует обработать корректным образом, так как они приходят в «сыром виде». При инициализации клиента MQTT-брокера определяется класс MQTTMessageHandler, который будет обрабатывать приходящие в темы сообщения.

```
fun handleMessage(topic: String, payload: String) {
    Log.i("MQTTHandler", "Обрабатываем сообщение: $payload с
топика: $topic")
    when {
        topic.startsWith(Topics.DEVICE_STATE_TOPIC) ->
handleDeviceStateMessage(topic, payload)
        topic.startsWith(Topics.DEVICE_COMMANDS_TOPIC) ->
handleDeviceCommandMessage(topic, payload)
        topic.startsWith(Topics.DEVICE_LIST_TOPIC) ->
handleDeviceListMessage(payload)
        topic.startsWith(Topics.LED_STATE_TOPIC) ->
handleLEDState(payload)
        else -> Log.i("MQTTHandler", "Необрабатываемый
топик: $topic")
    }
}
```

Данный метод выполняет роль оркестратора, принимая входящие сообщения и, в зависимости от темы, делегирует их обработку соответствующим методам. Этот подход предоставляет возможность для вариативной обработки событий в системе «умного дома».

4.2.1 Обработка состояний устройств происходит в методе, представленном ниже:

```
private fun handleDeviceStateMessage(topic: String,
payload: String) {
    val ieeeAddr = extractIeeeAddrFromTopic(topic)
    val deviceId = devicesViewModel.devices.value.find {
        it.ieeeAddr == ieeeAddr }?.id

    if (deviceId != null) {
        DeviceState.updateDeviceData(deviceId, payload)
    } else {
        println("Устройство с IEEE Addr $ieeeAddr не
найден")
    }
}
```

При получении сообщения из информационной темы устройства осуществляется извлечение его физического адреса из имени темы. На основе этого адреса определяется внутренний идентификатор устройства в приложении, после чего производится обновление текущего состояния устройства во временном хранилище данных приложения.

4.2.2 Для получения актуализированной информации о подключенных устройствах, в случае их сопряжения с использованием графического интерфейса координатора используется следующий метод:

```
private fun handleDeviceListMessage(payload: String) {
    try {
        val jsonObject = JSONObject(payload)
        for (key in jsonObject.keys()) {
            val deviceJson = jsonObject
                .optJSONObject(key) ?: continue
            val ieeeAddr = deviceJson.optString("ieeeAddr")
            val friendlyName = deviceJson
                .optString("friendly_name")
            val modelId = deviceJson
                .optString("ModelId")
            val device = Device.create(
                ieeeAddr = ieeeAddr,
                friendlyName = friendlyName,
                modelId = modelId,
                roomId = null,
                brokerId = BrokerState.brokerId.value ?: -1
            )
            devicesViewModel.addDeviceIfNotExists(device)
        }
    } catch (e: Exception) {
        Log.e("DEVICE", "Ошибка обработки списка устройств:
${e.message}")
    }
}
```

Здесь происходит первоначальная обработка информации, получаемой от брокера, с преобразованием в формат JSON. Далее выполняется обработка каждого ключа и извлекаются данные об устройстве. Если устройство еще не добавлено в систему, оно создается и добавляется в базу данных с использованием метода `addDeviceIfNotExists`.

4.2.3 Метод `handleDeviceCommandMessage` отвечает за обработку сообщений, содержащих команды для управления устройствами системы «умный дом». В этом методе извлекается информация о команде управления устройством, которая используется для формирования объекта, необходимого для создания записи в базе данных.

```
val jsonObject = JSONObject(payload)
val commandTopic = jsonObject
    .optString("command_topic")
    .takeIf { it.isNotBlank() } ?: return
val payloadOn = jsonObject.optString("payload_on", null)
val payloadOff = jsonObject.optString("payload_off", null)
val commandTemplate = jsonObject
    .optString("command_template", null)
val options = jsonObject.optJSONArray("options")
    ?.let { array -> (
        0 until array.length()).associate { index ->
            array.getString(index) to
            array.getString(index)
        }
    } ?: emptyMap()
val commandType = extractCommandTypeFromTopic(topic)
val deviceIeeeAddr = extractIeeeAddrFromTopic(topic)
```

Сначала происходит извлечение данных из входного сообщения в формате JSON. Затем определяется тип команды с помощью функции `extractCommandTypeFromTopic` и извлекается уникальный адрес устройства с помощью метода `extractIeeeAddrFromTopic`, что позволяет точно идентифицировать, какое устройство должно быть управляемым. После формирования объекта данные сохраняются в базу данных и привязываются к устройству соответствующему данному физическому адресу, что позволяет точно идентифицировать, какое устройство в приложении должно отображаться как управляемое.

4.3 Отправка пользовательских данных в MQTT-брокер

Для управления системой «умный дом» в приложении реализован механизм отправки пользовательских команд через MQTT-брокер. Когда пользователь взаимодействует с интерактивными элементами графического интерфейса – например, включает свет – приложение интерпретирует это как намерение изменить состояние соответствующего устройства. Для реализации

такого поведения формируется и отправляется MQTT-сообщение с помощью функции, предназначенной для обработки команд управления переключаемыми устройствами:

```
fun onToggle(deviceId: Int, state: Boolean) {
    viewModelScope.launch {
        val cmd = db.commandDAO()
            .getSwitchCommandByDeviceId(
                deviceId)

        val newState = if (!state) cmd?.payloadOff else
cmd?.payloadOn
        cmd?.let {
            if (newState != null) {
                sendCommandToMqtt(it.commandTopic, newState)
            }
        }
    }
}
```

В функции запускается корутина, внутри которой из локальной базы данных извлекается объект команды, связанный с конкретным устройством. На основе переданного состояния определяется, какое значение должно быть отправлено в брокер. Таким образом, приложение формирует MQTT-сообщение, соответствующее синтаксису Home Assistant, и отправляет его по указанному в объекте команды топику.

Для непосредственной отправки сообщения используется метод `publish`, определенный в интерфейсе `MQTTMessaging` и реализованный в классе `MQTTClient`.

```
override fun publish(topic: String, payload: String, qos:
Int, retained: Boolean) {
    try {
        val message =
            MqttMessage(payload.toByteArray()).apply {
                this.qos = qos
                this.isRetained = retained
            }
        mqttClient?.publish(topic, message)
        Log.i("MQTT", "Отправлено сообщение: $payload в топи
к: $topic")
    } catch (e: MqttException) {
        Log.e("MQTT", "Ошибка отправки сообщения: ${e.reason
Code} - ${e.message}")
    }
}
```

Внутри метода создается объект `MqttMessage`, которому задаются параметры качества обслуживания и флаг сохранения сообщения. После этого сформированное сообщение публикуется в указанный топик с помощью сущности `mqttClient`.

4.4 Управление модулем светодиодной подсветки

Для управления светодиодной подсветкой в приложении используется

два метода, в зависимости от выбранного пользователем:

- автоматический;
- ручная настройка.

4.4.1 Для автоматического режима подсветки используется следующий метод:

```
fun setAUTOMode() {
    viewModelScope.launch {
        val ledAutoStatus = ""
        {
            "state": "OFF",
            "brightness": 255,
            "color": {
                "r": 255,
                "g": 255,
                "b": 255
            },
            "color_mode": "rgb",
            "mode": "auto"
        }
        """.trimIndent()

        MQTTClient.publish(Topics.LED_SET_STATE_TOPIC,
            ledAutoStatus)
    }
}
```

В корутине создается JSON-объект, описывающий параметры подсветки в автоматическом режиме. Ключевое значение `mode` устанавливается в `auto`, что указывает устройству перейти в автоматический режим управления подсветкой. В этом режиме поведение светодиодов зависит от текущего состояния устройства: например, при активации режима сопряжения подсветка начинает мигать зеленым, сигнализируя о готовности к подключению.

4.4.2 Для настройки подсветки по пользовательским параметрам пользователю предоставляется возможность вручную отрегулировать яркость и интенсивность каждого цвета подсветки с помощью отдельных карточек-интерфейсов:

```
BrigtnessCard(
    title = "Brightness",
    color = Color(brightness / 255f, brightness / 255f,
        brightness / 255f),
    value = brightness,
    onValueChange = { LEDState.setBrightness(it) },
    onValueChangeFinished = { ledScreenViewModel
        .sendLEDStatus() })
```

Данная карточка отвечает за регулировку яркости светодиодной подсветки. При изменении значения на ползунке в режиме ручной настройки изменяется цвет индикатора яркости – небольшого визуального элемента внутри карточки. Этот элемент служит для наглядного отображения текущего

уровня яркости: от черного цвета, символизирующего минимально допустимый уровень подсветки, до белого – соответствующего максимальной яркости. Обновление значения яркости происходит в функции, передаваемой в параметре `onValueChange`, а отправку команды координатору – только после завершения действия пользователя с использованием функции `onValueChangeFinished`.

Для настройки интенсивности каждого цветового канала (красного, зеленого и синего) используется отдельная карточка:

```
ColorsCard(
    red = red,
    green = green,
    blue = blue,
    onRedChange = { LEDState.setRed(it) },
    onGreenChange = { LEDState.setGreen(it) },
    onBlueChange = { LEDState.setBlue(it) },
    onValueChangeFinished = {
        ledScreenViewModel.sendLEDStatus()
    }
)
```

Значения цветовых компонентов изменяются в режиме реального времени, а итоговое состояние модуля подсветки координатора передается устройству также только по завершении регулировки.

4.5 Отображение общего списка устройств системы

Полный список устройств системы отображается в соответствующем разделе интерфейса с использованием `DevicesScreen`. В данном компоненте происходит чтение потока, содержащего информацию о всех активных устройствах системы:

```
val switchDevices by devicesViewModel
    .getDevicesByTypeFlow("switch").collectAsState()
val selectDevices by devicesViewModel
    .getDevicesByTypeFlow("select").collectAsState()
val sliderDevices by devicesViewModel
    .getDevicesByTypeFlow("slider").collectAsState()
val devicesWithoutCommands by devicesViewModel
    .getDevicesWithoutCommandsFlow().collectAsState()
```

Для отображения устройств используется `DeviceCard`. Данный компонент является универсальным и позволяет динамически отображать различные элементы управления, такие как переключатели, ползунки и выпадающие списки, в зависимости от типа устройства и его состояния. Каждое устройство может быть связано с различными параметрами, такими как яркость, состояние включения/выключения или выбор параметра из списка.

4.5.1 В случае отображения устройства с возможностью переключения состояния, например, включения и выключения:

```

"switch" -> {
    val switchColors = SwitchDefaults.colors(
        checkedThumbColor = Color.White,
        checkedTrackColor = Color(0xFF4CAF50),
        uncheckedThumbColor = Color.White,
        uncheckedTrackColor = Color(0xFFFF44336))
    Switch(
        checked = checked,
        onCheckedChange = {
            checked = it
            onToggle?.invoke(it)
        },
        colors = switchColors
    )
}

```

4.5.2 Для устройств с возможностью регулирования значения, например, яркости освещения, значение может быть представлено в диапазоне от 0 до 100. В случае, если параметр устройства не находится в пределах этого диапазона, его значение делится на 100 равных промежутков. Каждый из этих промежутков затем используется для вычисления корректного значения в пределах от 0 до 100, что позволяет точно управлять параметром, даже если он изначально имеет другой диапазон. Эта логика представлена в коде ниже.

```

"slider" -> {
    val sliderColors = SliderDefaults.colors(
        thumbColor = Color(0xFF8A9F9B),
        activeTrackColor = Color(0xFFA6B6A9))
    Slider(
        value = sliderValue,
        onValueChange = {
            sliderValue = it
            onSliderChange?.invoke(it)
        },
        valueRange = 0f..100f,
        colors = sliderColors
    )
}

```

4.5.3 Для устройств с возможностью выбора параметра из списка, например, режим работы устройства, где значение может быть представлено в виде заранее определенного набора опций, используется следующий элемент управления:

```

"select" -> ExposedDropDownMenuBox(
    expanded = expanded,
    onExpandedChange = {
        expanded = it }) {
    TextField(
        value = selectedOption,
        onValueChange = {},
        readOnly = true,
        modifier = Modifier.menuAnchor(),
        label = { Text("Выберите") },
        trailingIcon = {
            ExposedDropDownMenuDefaults
                .TrailingIcon(expanded = expanded) })
    ExposedDropDownMenu(
        expanded = expanded,
        onDismissRequest = { expanded = false }) {
        options.forEach {
            option ->

```



```

        DropdownMenuItem(
            text = { Text(option) },
            onClick = {
                selectedOption = option
                expanded = false
                onSelectChange?.invoke(option) }) }) })
    }
}

```

4.6 Отображение списка комнат системы

Для получения списка комнат системы необходимо перейти на вторую вкладку главного экрана приложения. После нажатия на кнопку «Rooms» откроется интерфейс, который отобразит карточки комнат, каждая из которых будет содержать название и количество устройств, связанных с данной комнатой. За данный функционал отвечает следующий код:

```

items(rooms) { room ->
    var deviceCount by remember {
        mutableIntStateOf(0) }
    LaunchedEffect(room.id) {
        roomsViewModel.getDeviceCount(room.id) {
            count -> deviceCount = count
        }
    }
    RoomCard(
        roomId = room.id.toInt(),
        roomName = room.name,
        deviceCount = deviceCount,
        navHostController = navHostController,
        onDelete = { roomId ->
            roomsViewModel.deleteRoom(roomId)
        }
    )
}

```

Для каждой комнаты мы создаем карточку, используя Composable-компонент RoomCard. В этот компонент передается информация об комнате, а также функции для навигации, которые позволяют перейти на новый экран с информацией об устройствах, относящихся к этой комнате. Также реализована функция для удаления комнаты, которая вызывается при нажатии на соответствующую кнопку внутри карточки.

Внутри компонента формируется карточка при нажатии на которую пользователя перенаправляет на экран информации об устройствах, относящихся к выбранной комнате. Для этого используется следующий код:

```

Card(
    shape = RoundedCornerShape(16.dp),
    modifier = Modifier
        .fillMaxWidth()
        .padding(8.dp)
        .clickable { navHostController
            .navigate("room details/$roomId") },
    elevation = CardDefaults
        .cardElevation(6.dp),
    colors = CardDefaults.cardColors(
        containerColor = Color(0xFFE3F2FD)
    )
)

```

Отображение устройств, привязанных к выбранной комнате, реализованно аналогично отображению всех устройств, однако при этом накладывается дополнительный фильтр, учитывающий привязку устройства к комнате. Пример фильтрации для устройств с режимом «переключения» состояния:

```
val switchDevices by devicesViewModel
    .getDevicesByTypeFlow("switch")
    .collectAsState()

val roomSwitchDevices = switchDevices
    .filter { it -> it.id in roomDevices.map {
        it.id } }
```

После фильтрации на экране RoomDetailsScreen отобразятся все устройства, связанные с этой комнатой.

4.7 Добавление новых комнат в систему

На экране RoomsScreen реализован функционал добавления новой комнаты с помощью плавающей кнопки (FloatingActionButton), расположенной в правом нижнем углу интерфейса. Кнопка оформлена в соответствии с Material Design 3 и содержит иконку «плюс», обозначающую действие добавления.

```
FloatingActionButton(
    onClick = { showDialog = true },
    containerColor =
MaterialTheme.colorScheme.primary,
    modifier = Modifier
        .align(Alignment.BottomEnd)
        .padding(16.dp)
) {
    Icon(imageVector = Icons.Default.Add,
contentDescription = "Добавить комнату")
}
```

При нажатии на кнопку состояние отображения диалогового окна showDialog становится true, и на экране отображается диалоговое окно AddRoomDialog. Это диалоговое окно позволяет пользователю ввести название новой комнаты.

```
if (showDialog) {
    AddRoomDialog(
        onDismiss = { showDialog = false },
        onConfirm = { roomName ->
            roomsViewModel.addRoom(roomName)
            showDialog = false
            Log.i("Rooms", "Комната добавлена: $roomName")}
    )
}
```

При заполнении информации в диалоговом окне предусмотрены два

действия:

- отмена – закрытие диалогового окна без сохранения данных;
- подтвердить – введенное имя комнаты передается в `roomsViewModel.addRoom`, после чего диалог закрывается и начинается процесс сохранения комнаты в базе данных.

Функция `addRoom` во `ViewModel` отвечает за асинхронное создание и сохранение новой комнаты в базе данных:

```
fun addRoom(name: String) {  
    viewModelScope.launch {  
        db.roomEntityDAO().insertRoom(  
            RoomEntity(name = name))  
        })  
}
```

При вызове этого метода:

- создается объект `RoomEntity` с указанным именем;
- выполняется вставка в базу данных с помощью DAO (`insertRoom`);
- операция запускается внутри `viewModelScope.launch`, что обеспечивает асинхронное выполнение без блокировки основного потока интерфейса.

4.8 Управление режимом обнаружения и подключения новых устройств

На экране настроек приложения реализован элемент управления режимом обнаружения новых устройств – переключатель `Switch`, размещенный внутри компонента `Card`. Данный переключатель предоставляет пользователю возможность вручную активировать или деактивировать режим «обнаружения устройств» в системе умного дома.

```
Switch(  
    checked = discoveryState,  
    onCheckedChange = { onSwitchChanged(it) },  
    colors = SwitchDefaults.colors(  
        checkedThumbColor = Color.White,  
        checkedTrackColor = Color(0xFF4CAF50),  
        uncheckedThumbColor = Color.White,  
        uncheckedTrackColor = Color(0xFFF44336)  
    )  
)
```

Изменение состояния переключателя вызывает функцию `onSwitchChanged`, которая в зависимости от значения параметра `enabled` инициирует соответствующее действие:

```
fun onSwitchChanged(enabled: Boolean) {  
    if (enabled) {  
        DiscoveryState.startDiscovery()  
    } else {  
        DiscoveryState.stopDiscovery()  
    }  
}
```

Функциональность включения и выключения режима обнаружения реализована в объекте `DiscoveryState`.

```
fun startDiscovery() {
    _isDiscoveryActive.value = true
    MQTTClient.publish(
        Topics.DISCOVERY_TOPIC,
        Constants.DISCOVERY_ENABLE
    )

    resetJob?.cancel()

    resetJob = CoroutineScope(Dispatchers.Default)
        .launch {
            delay(Constants.DISCOVERY_TIME)
            _isDiscoveryActive.value = false
        }
}
```

Данная функция отвечает за активацию режима обнаружения устройств. Она устанавливает значение флага `_isDiscoveryActive` в `true`, публикует соответствующее сообщение в тему `DISCOVERY_TOPIC`, а также запускает корутину, которая запускает таймер на 255 секунд. При этом, если ранее уже был запущен таймер, он отменяется, чтобы избежать конфликтов и повторных срабатываний.

4.9 Отображение детальной информации об устройстве

Для просмотра подробной информации о состоянии устройства и показаниях датчиков необходимо нажать на карточку соответствующего устройства. В компонент `DeviceCard` передается `NavHostController`.

```
DeviceCard(
    deviceId = device.id,
    imageRes = R.drawable.mqtt_logo,
    name = device.friendlyName,
    type = Constants.SWITCH_TYPE,
    value = (deviceData?.get("state") as? String)?
        .toBooleanState() ?: false,
    navController = navHostController,
    onToggle = { state ->
        devicesViewModel.onToggle(device.id, state)
    }
)
```

Обработка навигации осуществляется внутри компонента `DeviceCard` с использованием модификатора `clickable`, с помощью которого при нажатии осуществляется переход на экран с деталями: вызывается навигация на маршрут `device_details/$deviceId`, где `deviceId` – идентификатор выбранного устройства.

```
Card(
    shape = RoundedCornerShape(16.dp),
    modifier = Modifier
        .fillMaxWidth()
        .padding(8.dp)
)
```

```

        .clickable {
            navController.navigate("device_details/$deviceId") },
            elevation = CardDefaults.cardElevation(6.dp)
        )
    )
}

```

После перехода на соответствующий маршрут отображается экран с подробной информацией об устройстве. На экране доступно изменение дружественного имени устройства и выпадающее меню для привязки к комнате, а также отображается текущая привязанная комната. Ниже выводится информация о всех доступных значениях датчиков, а также служебные данные устройства, такие как `last_seen`. Все значения получаются из временного хранилища `DeviceState`.

4.10 Привязка устройства к определенной комнате

Привязка устройства к комнате подразумевает наличие заранее созданной комнаты в системе. Пользователь может выбрать одну из доступных комнат из выпадающего списка на экране подробной информации об устройстве.

```

ExposedDropdownMenu(
    expanded = expanded,
    onDismissRequest = { expanded = false }
) {
    DropdownMenuItem(
        text = { Text("Select room") },
        onClick = {
            selectedRoom = "Select room"
            expanded = false
            devicesViewModel
                .assignRoomToDevice(deviceId, null)
        }
    )
    rooms.forEach { room ->
        DropdownMenuItem(
            text = { Text(room.name) },
            onClick = {
                selectedRoom = room.name
                expanded = false
                devicesViewModel
                    .assignRoomToDevice(
                        deviceId, room.id)
            }
        )
    }
}

```

Данный компонент отвечает за отображение выпадающего меню с доступными комнатами. Он реализован с использованием `ExposedDropdownMenu`, внутри которого каждый элемент представляет одну из заранее созданных комнат. При выборе комнаты обновляется локальное состояние `selectedRoom`, закрывается меню и вызывается метод `assignRoomToDevice` у `devicesViewModel`, передающий идентификатор выбранной комнаты для привязки устройства.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

В данном разделе описано руководство по установке и взаимодействию с системой «умный дом». Это руководство разделено на три части по настройке, поскольку в рамках дипломного проекта используется как ранее реализованная аппаратная часть с прошивкой, так и разрабатываемое программная часть:

1 Руководство пользователя координатора. Этот раздел описывает процесс подготовки устройства координатора к работе в системе «умный дом». Здесь представлены инструкции по прошивке координатора, подключению к источнику питания и сети, а также первоначальной настройке для корректной работы с приложением.

2 Руководство пользователя MQTT-брокера. В этом разделе описывается настройка серверной части системы, выполняющей функцию брокера сообщений. Приведены инструкции по установке и конфигурации брокера, настройке параметров подключения и безопасности.

3 Руководство пользователя мобильного приложения. Этот раздел посвящен использованию мобильного приложения для управления системой. Здесь будут представлены инструкции по загрузке, установке и запуску приложения. Так же будут рассмотрены основные функции приложения, включая добавление системы в приложение, просмотр информации об устройствах, а также управление системными модулями.

5.1 Руководство пользователя координатора

5.1.1 После первого включения координатора, при условии успешной прошивки микроконтроллера ESP32 и радиомодуля E72, можно приступить к его первоначальной настройке.

Для этого необходимо подключить питание через разъем microUSB. После подачи питания координатор автоматически перейдет в режим точки доступа, что позволяет выполнить начальную настройку доступа в интернет без необходимости перепрошивки устройства. В доступных сетях Wi-Fi появится точка доступа с названием формата «SLS XXXX» (см рисунок 5.1).

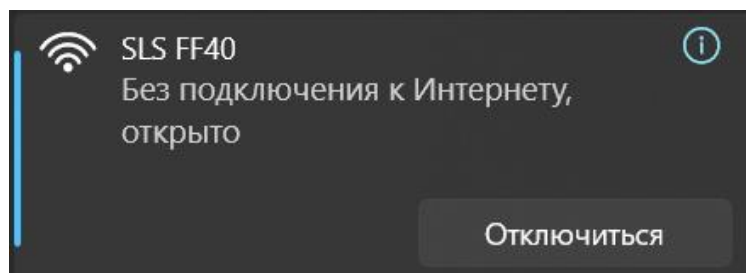


Рисунок 5.1 – Wi-Fi для подключения к координатору

Для подключения к координатору подойдет любое устройство, поддерживающее Wi-Fi не ниже версии 4. После подключения к сети

необходимо открыть веб-браузер и перейти на локальный сервер шлюза по адресу «http://192.168.100.1», где и осуществляется дальнейшая настройка подключения координатора к интернету. Интерфейс подключения к сети изображен на рисунке 5.2.

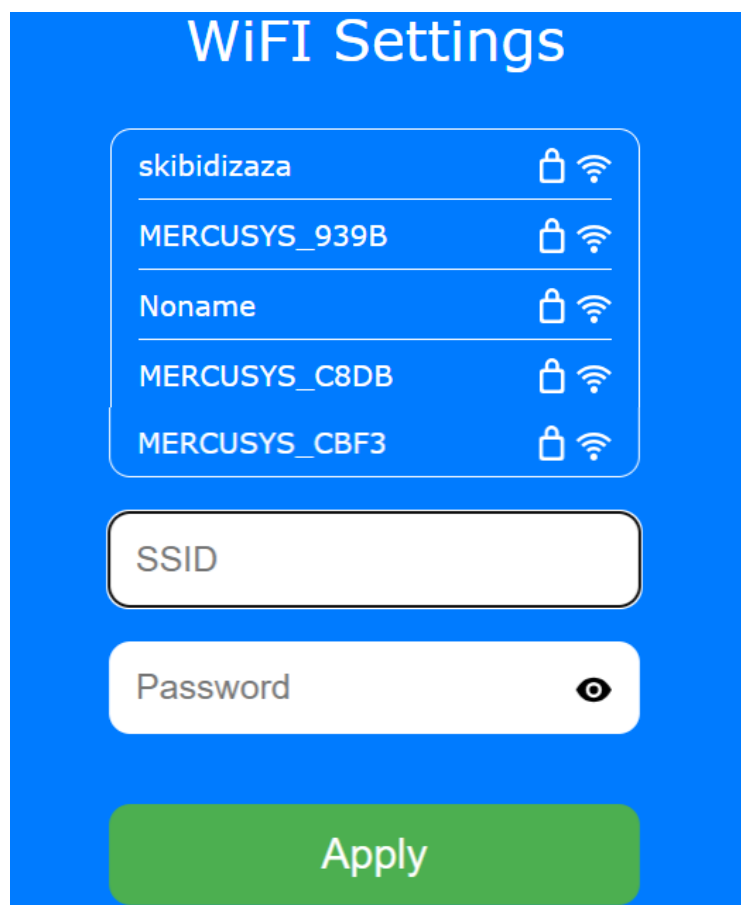


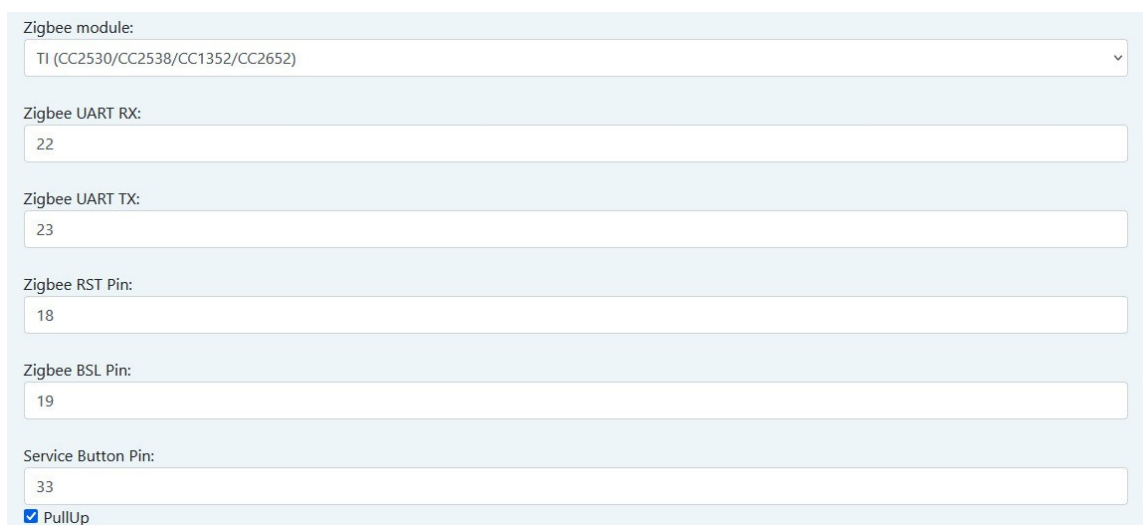
Рисунок 5.2 – Интерфейс подключения к Wi-Fi

В открывшейся веб-странице необходимо выбрать сеть Wi-Fi из списка, отображающего доступные в данный момент сети, которые видит координатор. Также предусмотрена возможность ручного ввода названия сети (SSID) и пароля, если нужная сеть не отображается в списке.

После подключения к сети Wi-Fi необходимо произвести полную перезагрузку устройства, отключив и повторно подключив его через разъем microUSB. После перезагрузки устройство автоматически перейдет в режим координатора с собственным веб-сервером, доступным по IP-адресу, который будет выдан маршрутизатором. Узнать IP-адрес координатора можно через веб-интерфейс маршрутизатора в списке подключенных устройств или с помощью сетевых сканеров, таких как nmap с использованием команды `nmap -sn 192.168.0.1/24`, где необходимо указать адрес локальной подсети.

5.1.2 Для корректной работы координатора необходимо произвести настройку номеров контактов подключения модуля радиосвязи к модулю микроконтроллера. Для этого необходимо перейти во вкладку Settings →

Hardware. Откроется интерфейс настройки периферии, который необходимо заполнить следующим образом, как показано на рисунке 5.3.

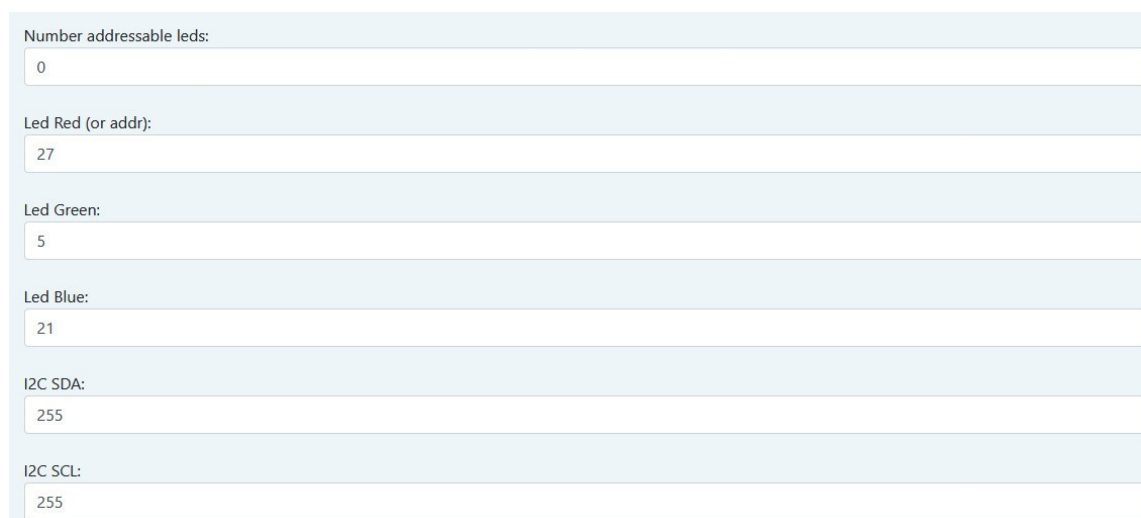


The screenshot shows a configuration form for a Zigbee module. It includes several input fields and a checkbox, all set with default values:

- Zigbee module:** A dropdown menu showing "TI (CC2530/CC2538/CC1352/CC2652)".
- Zigbee UART RX:** An input field containing the number "22".
- Zigbee UART TX:** An input field containing the number "23".
- Zigbee RST Pin:** An input field containing the number "18".
- Zigbee BSL Pin:** An input field containing the number "19".
- Service Button Pin:** An input field containing the number "33".
- PullUp:** A checkbox that is checked.

Рисунок 5.3 – Интерфейс аппаратной конфигурации модуля радиосвязи

Для настройки модуля светодиодной подсветки необходимо заполнить информацию в форме ниже. Данные для настройки представлены на рисунке 5.4. После ввода данных необходимо нажать кнопку «Save» для сохранения параметров.



The screenshot shows a configuration form for an LED module. It includes several input fields, all set with default values:

- Number addressable leds:** An input field containing the number "0".
- Led Red (or addr):** An input field containing the number "27".
- Led Green:** An input field containing the number "5".
- Led Blue:** An input field containing the number "21".
- I2C SDA:** An input field containing the number "255".
- I2C SCL:** An input field containing the number "255".

Рисунок 5.4 – Интерфейс аппаратной конфигурации модуля светодиодной подсветки

После настройки конфигурации необходимо выполнить перезагрузку устройства с помощью соответствующей кнопки в графическом интерфейсе. При успешной настройке и корректном подключении модуль радиосвязи будет обнаружен, и его статус отобразится как Status: 9 OK, как показано на рисунке 5.5. В случае ошибки или некорректной конфигурации в этом же поле будет выведено сообщение с указанием типа ошибки.

Zigbee info

Devices: 2
ChipType: TI ZStack
IeeeAddr: 0x00124B00259063CF
Revision: 20240710
PanId: 0x1234
Channel: 11
DeviceState: 9 [**OK**]

Рисунок 5.5 – Успешное подключение модуля радиосвязи

5.1.3 После успешной настройки радиомодуля необходимо подключить координатор к брокеру MQTT. Для этого необходимо перейти во вкладку Settings → Link → MQTT Setup. В открывшемся окне необходимо выставить чек-бокс Enable в активное состояние. Ниже необходимо заполнить поля, содержащие информацию о брокере. Поля «User» и «Password» необходимо заполнить в случае настроенной авторизации. В поле «System prefix topic» необходимо ввести значение «zigbee». Поля состояний «Retain States» и «Home Assistant MQTT Discovery» перевести в активное состояние. Интерфейс с введенными данными должен выглядеть так, как показано на рисунке 5.6.

MQTT Setup

☒ Enable

Server:
134.17.15.236

Port:
1883

User (if authorization is required on MQTT server):
(leave blank to ignore MQTT authorization)

Password:

Topic:
zigbee

☒ Retain states
☐ MQTT Retain lwt
☒ Home Assistant MQTT Discovery
☒ Add FriendlyName to Entity Name

Discovery topic (default: homeassistant):
homeassistant

Save

Рисунок 5.6 – Интерфейс настройки подключения к MQTT брокеру

Если данные для подключения указаны верно, то на главном экране веб-интерфейса координатора появится окно «States», в котором будет отображен статус подключения.

5.2 Руководство пользователя MQTT-брокера

5.2.1 Для корректной работы брокера MQTT сообщений необходима операционная система на базе Linux удовлетворяющая следующим условиям:

- операционная система: дистрибутив на базе ядра Linux версии 3.10 и выше;
- процессор: 1-ядерный, с тактовой частотой не менее 2,6 ГГц;
- оперативная память: не менее 1 ГБ;
- свободное место на диске: не менее 12 ГБ;
- наличие стабильного подключения к сети Интернет со скоростью не ниже 1 Мбит/с;
- поддержка виртуализации.

5.2.2 Для непрерывной и стабильной работы брокера сообщений используется инструмент контейнеризации Docker. Для установки необходимого программного пакета следует воспользоваться официальной инструкцией [6], соответствующей используемой операционной системе. Выбор инструкции в соответствии с платформой представлен на рисунке 5.7

Supported platforms

Platform	x86_64 / amd64	arm64 / aarch64	arm (32-bit)	ppc64le	s390x
CentOS	✓	✓		✓	
Debian	✓	✓	✓	✓	
Fedora	✓	✓		✓	
Raspberry Pi OS (32-bit)			✓		
RHEL	✓	✓			✓
SLES					✓
Ubuntu	✓	✓	✓	✓	✓
Binaries	✓	✓	✓		

Рисунок 5.7 – Выбор инструкции в соответствии с используемой платформой для установки Docker

После успешной установки Docker Engine, необходимо установить плагин Docker Compose [7] в соответствии с операционной системой. Установка плагина производится с использованием пакетного менеджера.

5.2.3 После установки инструментов контейнеризации необходимо подготовить следующую структуру проекта:

- каталог log – используется для хранения логов брокера;
- каталог config – используется для хранения конфигурационного файла брокера mosquitto.conf и файла с учетными данными для авторизации

пользователей password.txt.

Содержимое конфигурационного файла mosquitto.conf:

```
listener 1883
persistence true
persistence_location /mosquitto/data/
log_dest file /mosquitto/log/mosquitto.log
```

В корне проекта необходимо создать конфигурационный файл docker-compose.yml со следующим содержимым:

```
version: '3.9'

services:
  mosquitto:
    image: eclipse-mosquitto:latest
    container_name: mqtt-broker
    restart: unless-stopped
    ports:
      - "1883:1883"
      - "9001:9001"
    volumes:
      - ./config/mosquitto.conf:
          /mosquitto/config/mosquitto.conf:ro
      - ./config/password.txt:
          /mosquitto/config/password.txt:ro
      - data:/mosquitto/data
      - ./log:/mosquitto/log

volumes:
  data:
    name: mqtt-broker-data
```

5.2.4 В готовом проекте с использованием терминала необходимо запустить проект, используя Docker Compose и следующую команду:

```
$ docker-compose up -d
```

После выполнения команды будет загружен образ брокера, контейнер запустится в фоновом режиме и станет доступен для подключения по порту 1883. Для корректного подключения из внешней сети может потребоваться дополнительная настройка сетевого оборудования.

5.3 Руководство пользователя мобильного приложения

5.3.1 Для корректной работы мобильного приложения необходимо соблюдение следующих условий:

- наличие стабильного подключения к сети Интернет (Wi-Fi или мобильная передача данных) со скоростью не ниже 1 Мбит/с;
- операционная система Android версии 8.0 или выше.

5.3.2 Для установки мобильного приложения необходимо запустить исполняемый файл с расширением .apk на устройстве. После запуска

установки следует подтвердить действие и дождаться завершения стандартной процедуры установки.

В случае, если система заблокирует установку, необходимо разрешить установку приложений из неизвестных источников. Для этого может потребоваться перейти в настройки устройства и активировать соответствующую опцию, которая обычно находится в разделе «Безопасность» или «Для разработчиков».

Процесс установки может незначительно отличаться в зависимости от версии операционной системы Android или модели устройства.

5.3.3 После завершения установки приложение готово к использованию. Для запуска необходимо нажать на иконку с соответствующим названием, отображенную на рабочем столе устройства показанную на рисунке 5.8.

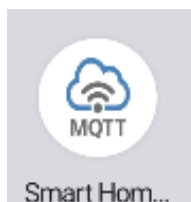


Рисунок 5.8 – Установленное приложение

После запуска откроется начальный экран приложения (см. рисунок 5.9), содержащий окно авторизации и подключения к MQTT-брокеру системы «умный дом».

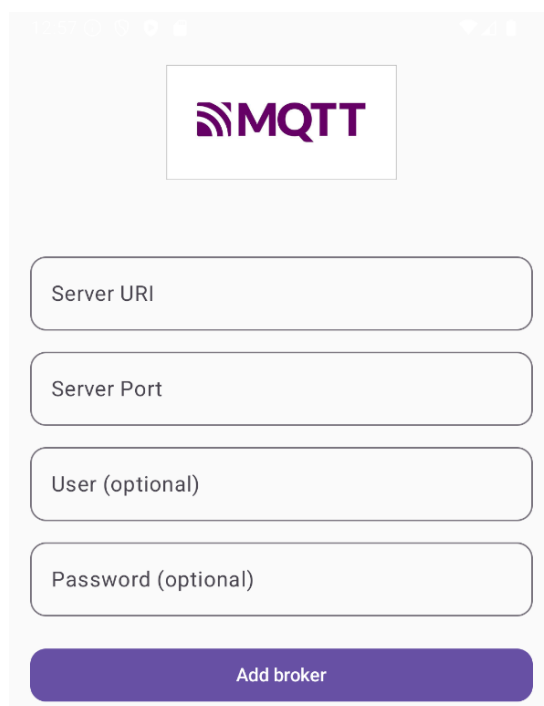


Рисунок 5.9 – Экран авторизации приложения

5.3.4 На экране авторизации отображается форма для ввода данных, необходимых для подключения к системе «умный дом». Форма содержит четыре поля:

- IP-адрес брокера – адрес сервера, на котором работает MQTT-брокер;
- порт – номер порта, используемого для соединения с брокером;
- имя пользователя – логин для авторизации (если требуется);
- пароль – пароль для авторизации (если требуется).

После заполнения обязательных полей пользователь инициирует проверку валидности данных, нажав кнопку добавления брокера. Приложение выполняет попытку установить соединение с указанным MQTT-брокером. В случае успешного подключения информация о системе сохраняется в локальную базу данных устройства, после чего на главном экране, под формой авторизации, появляется соответствующая карточка (см. рисунок 5.10) с информацией о системе и следующими элементами управления:

- кнопка «войти» – для перехода к управлению системой;
- кнопка «удалить» – для удаления записи о доступном подключении.

The image shows a mobile application interface for MQTT broker authorization. At the top, there is a logo with the text "MQTT". Below the logo are four input fields: "Server URI", "Server Port", "User (optional)", and "Password (optional)". Below these fields is a purple button labeled "Add broker". Underneath the button is a section titled "Recently used broker:". This section contains a light purple card with the text "URI: yahor.monster" and "Port: 1883". At the bottom of this card are two buttons: a purple "Login" button and a light purple "Delete" button.

Рисунок 5.10 – Экран авторизации при успешном подключении

На этом этапе пользователю необходимо нажать на кнопку «войти» расположенное в карточке с информацией о доступном брокере.

5.3.5 После успешного входа в систему пользователь попадает на главный экран управления системой умного дома. На этом экране отображаются все устройства, подключенные к координатору, с возможностью просмотра информации и управления. В случае отсутствия подключенных устройств, то на экране будет отображаться надпись «No connected devices» (см. рисунок 5.11).

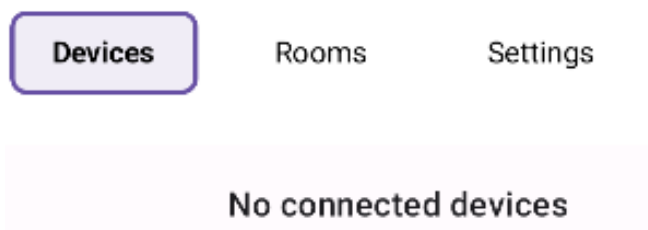


Рисунок 5.11 – Главный экран приложения

5.3.6 Для добавления новых устройств необходимо ознакомиться со списком поддерживаемых устройств на официальном сайте поставщика программного обеспечения SLS Smart Home [5]. Если устройство присутствует в списке поддерживаемых, необходимо перейти во вкладку «Settings», где доступна функция обнаружения и добавления новых устройств. (см. рисунок 5.12).

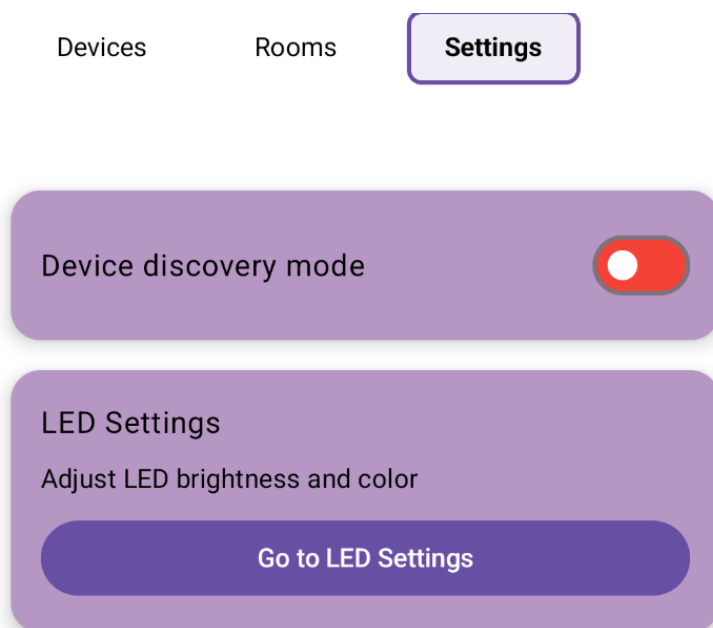


Рисунок 5.12 – Экран настроек

Для включения режима обнаружения необходимо установить значение переключателя в положение «Включено» – в этом случае он подсветится зеленым цветом. После активации на карточке отобразится обратный отсчет времени (см. рисунок 5.13), в течение которого система будет искать новые устройства и устанавливать с ними соединение для обмена информацией.

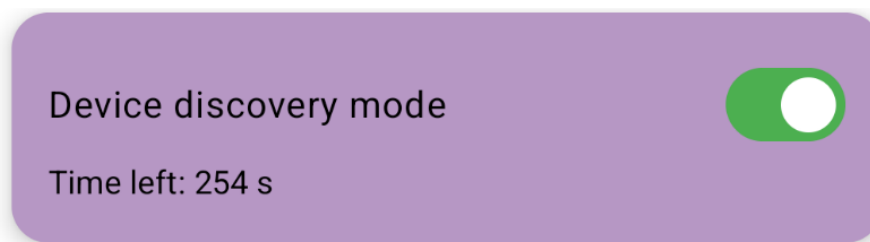


Рисунок 5.13 – Включенный режим обнаружения и сопряжения устройств

После активации режима обнаружения в мобильном приложении необходимо на оконечном устройстве выполнить действия для перевода его в режим сопряжения. Конкретный способ активации режима сопряжения зависит от типа устройства и его производителя – например, может потребоваться зажать кнопку на устройстве или выполнить определенную комбинацию действий, указанную в инструкции к устройству.

После выполнения вышеописанных действий устройство, как правило, появляется на главном экране приложения в течение нескольких секунд.

5.3.7 После успешного добавления устройства оно становится доступным для взаимодействия через мобильное приложение (см. рисунок 5.14).

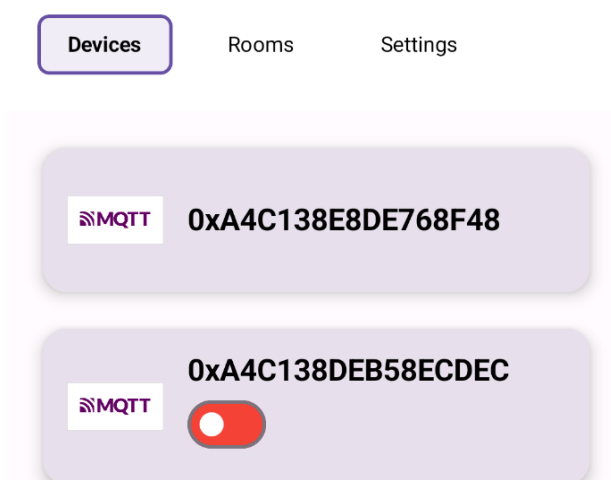


Рисунок 5.14 – Главный экран приложения после подключения устройства

В случае, если устройство поддерживает управление, на его карточке в главном экране отображается соответствующий элемент управления. Как показано на рисунке 5.13, для розетки доступен переключатель состояния – при его нажатии розетка включается или выключается, а ее текущее состояние визуально отображается в приложении с помощью изменения цвета элемента управления.

Для просмотра дополнительной информации о конкретном устройстве пользователь может нажать на его карточку. После этого откроется экран с детализированной информацией, где можно изменить название устройства в

приложении, привязать его к комнате для удобной территориальной группировки, а также просмотреть доступные параметры устройства. Пример такого экрана приведен на рисунке 5.15.

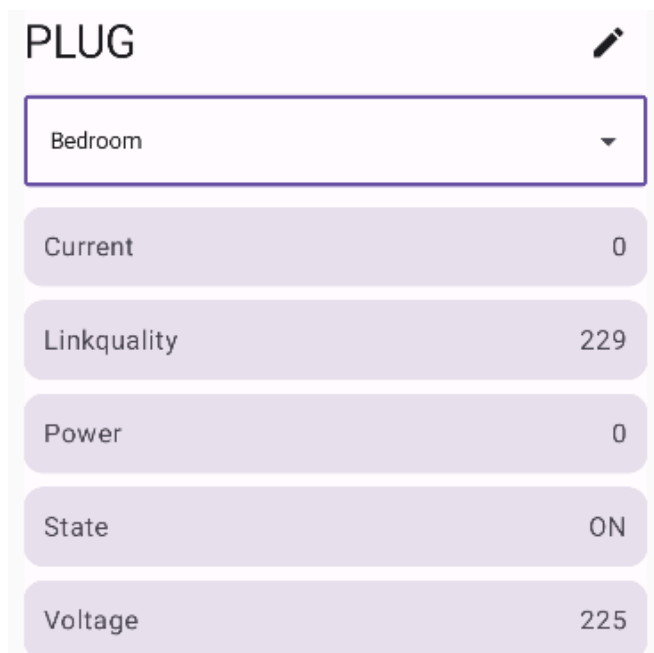


Рисунок 5.15 – Экран информации об устройстве

На данном экране пользователь может изменить привязку устройства к комнате, а также задать дружественное имя устройства в приложении для его более удобной идентификации.

5.4 Руководство программиста

Для сборки приложения, а также внесения изменений в его исходный код, необходимо использовать среду разработки Android Studio. Рекомендуемые системные требования для стабильной работы Android Studio следующие:

- операционная система: Windows 10 и выше;
- процессор: 4-ядерный, с тактовой частотой 2,6 ГГц;
- оперативная память: не менее 8 ГБ;
- свободное место на диске: не менее 15 ГБ.

Перед началом работы необходимо установить Android Studio, загрузить необходимые SDK и инструменты, а также импортировать проект в среду разработки.

5.4.1 Для установки IDE Android Studio необходимо перейти на официальный сайт разработчиков программного обеспечения [8] и на главной странице нажать «Download Android Studio» (см. рисунок 5.16) после чего начнется загрузка ПО. По завершению загрузки необходимо запустить исполняемый файл установщика.

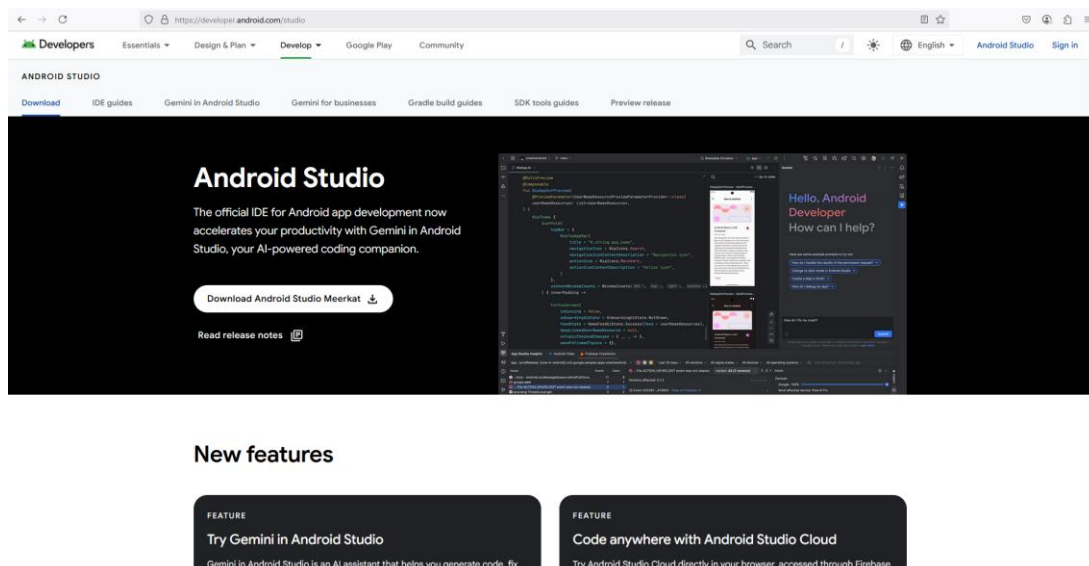


Рисунок 5.16 – Страница скачивания IDE Android Studio

В процессе подготовки среды разработки необходимо установить Android Studio — основную интегрированную среду для написания и отладки Android-приложений. Также требуется установить компонент Android Virtual Device (AVD), который позволяет создавать и запускать виртуальные устройства с различными конфигурациями для тестирования работы приложения без физического смартфона. Установка и настройка этих компонентов показана на рисунке 5.17.

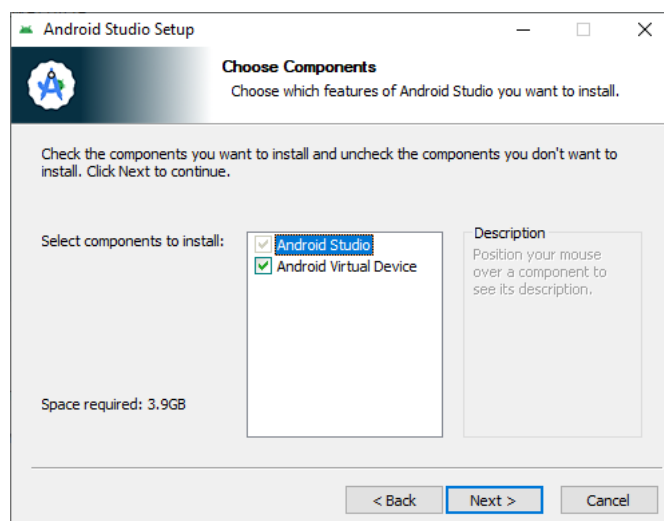


Рисунок 5.17 – Установка необходимых компонентов приложения

После завершения установки запустите Android Studio и при появлении окна настройки проекта выберите стандартные параметры (см. рисунок 5.18): это включает установку Android SDK и необходимых зависимостей, конфигурацию Gradle и подключение рекомендуемых плагинов для работы с разными версиями Android; после загрузки всех компонентов среда разработки будет готова к использованию.

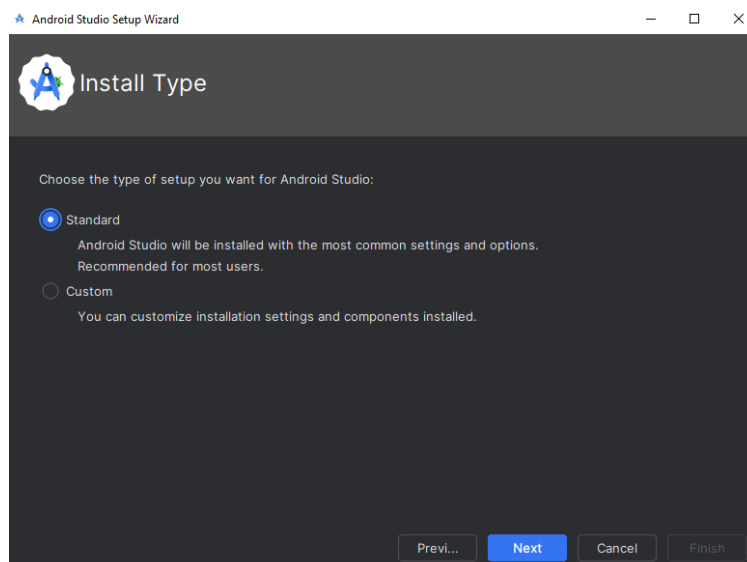


Рисунок 5.18 – Выбор режима установки приложения

5.4.2 Для работы с проектом необходимо открыть корневой каталог проекта «ha-application». На главном экране Android Studio нажмите кнопку «Open Project», затем в открывшемся диалоговом окне выберите папку с проектом и нажмите ОК. После этого проект откроется в Android Studio, как показано на рисунке 5.19.

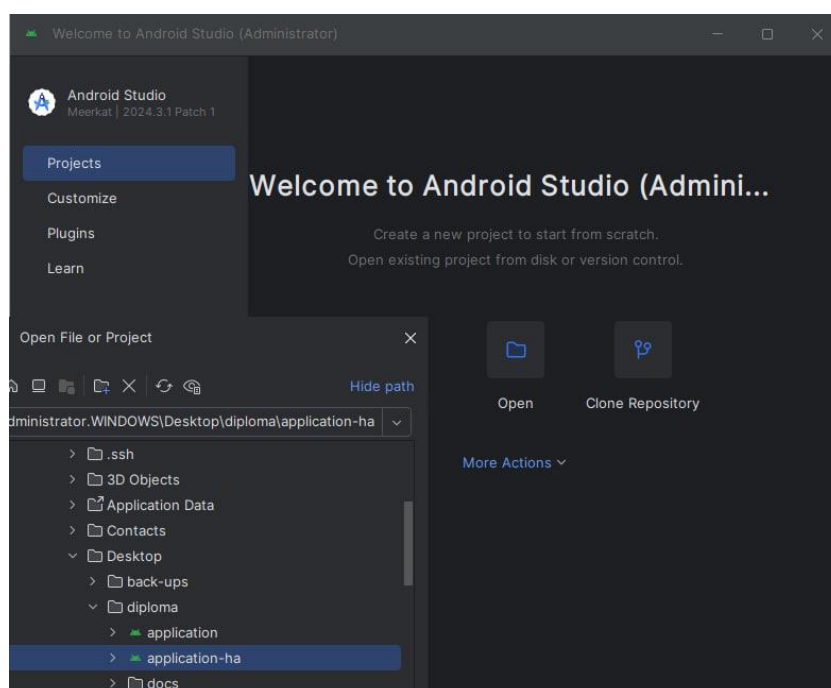


Рисунок 5.19 – Пример открытия проекта

После того как Android Studio завершит индексацию файлов и загрузку зависимостей проекта, можно приступать к запуску и внесению изменений в проект.

6 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

Этот раздел описывает программу и методику испытаний приложения. Целью тестирования является проверка правильности работы основных функций приложения, стабильности работы в различных условиях и соответствия требованиям, предъявляемым к пользовательскому интерфейсу и взаимодействию с устройствами по протоколу MQTT.

6.1 Автоматическое тестирование

Первый этап тестирования приложения – это автоматическое тестирование, направленное на проверку корректности работы отдельных компонентов на уровне исходного кода. Такие тесты позволяют выявить ошибки в логике до сборки и запуска приложения, что упрощает отладку и ускоряет процесс разработки.

В рамках автоматического тестирования производится проверка исходных файлов проекта, включая их соответствие стандартам кодирования, а также проверка версий используемых библиотек и зависимостей для выявления возможных несовместимостей.

6.1.1 Начало тестирования приложения включает проверку компонентов на уровне исходного кода. В качестве сборщика проекта используется Gradle, который выполняет следующие функции:

- проверка исходного кода на наличие логических или синтаксических ошибок;
- управление зависимостями проекта, включая проверку наличия библиотек, их версий и совместимости с другими библиотеками проекта;
- сборка проекта и компиляция исходного кода в единый исполняемый файл приложения;
- автоматический запуск unit-тестов.

Использование Gradle в качестве инструмента сборки проекта так же позволяет создавать детализированные отчёты о результатах тестирования и сборки приложения и результатов проверки стиля кода, что помогает улучшить качество приложения на протяжении всего жизненного цикла разработки.

6.1.2 На следующем этапе было проведено Unit-тестирование ключевых функциональных блоков приложения. Основное внимание уделялось проверке корректности подключения приложения к брокеру MQTT, а также обеспечению стабильного взаимодействия с базой данных.

Для этого были разработаны тесты, охватывающие следующий функционал:

- возможность подключения к MQTT брокеру;
- возможность публикации MQTT сообщений в топик;
- возможность подписки и получения сообщений из топика;

– корректность выполнения операций чтения, записи, изменения и удаления в базе данных.

Результаты тестирования представлены в таблице 6.1.

Таблица 6.1 – Результаты Unit-тестирования

Тестируемый функционал	Краткое описание теста	Результат тестирования
Подключение к MQTT брокеру	Проверка успешного подключения к MQTT брокеру с валидными данными	Успешно
Публикация MQTT сообщения	Проверка успешной публикации MQTT сообщений в топик «test/topic»	Успешно
Подписка на MQTT сообщения	Проверка возможности подписки на получение сообщений	Успешно
Получение MQTT сообщения	Проверка успешного получения MQTT сообщения из топика «test/topic»	Успешно
Проверка подключения к базе данных	Проверка успешного подключения к базе данных Room	Успешно
Проверка операций записи в базу данных	Проверка возможности добавления записей в таблицу Brokers	Успешно
Проверка операций чтения из базы данных	Проверка возможности получения данных из таблицы Brokers	Успешно
Проверка операций изменения в базе данных	Проверка возможности изменения данных в таблице Brokers	Успешно
Проверка операций удаления в базе данных	Проверка возможности удаления данных из таблицы Brokers	Успешно

По результатам Unit-тестирования был успешно проверен весь функционал, связанный с получением, отправкой и обработкой данных в системе умного дома.

6.2 Ручное тестирование

На втором этапе проводилось ручное тестирование приложения, направленное на проверку его работы в условиях, приближенных к реальному использованию. Тестирование выполнялось как на виртуальном Android-устройстве (AVD), так и на физическом смартфоне, что позволило оценить поведение интерфейса, стабильность соединения и корректность обработки пользовательских действий.

Особое внимание уделялось взаимодействию приложения с реальными устройствами умного дома, подключенными в сеть Zigbee. В ходе тестирования использовались такие устройства, как:

- умная zigbee розетка _TZ3000_zloso4jk (TS011F) от производителя Tuuya [9];
- датчик температуры и влажности _TZ3000_flyghyj (TS0201) от производителя Tuuya [10].

Каждый тест предполагает, что все функции, кроме тестируемой, работают корректно. Каждый тест-кейс включает описание шагов, которые необходимо выполнить для проверки конкретной функциональности приложения.

6.2.1 В таблице 6.2 представлено описание тест-кейсов, разработанных для проведения всестороннего испытания модуля авторизации пользователя и подключения приложения к MQTT-брокеру. Тест-кейсы охватывают основные сценарии использования модуля.

Таблица 6.2 – Описание тест-кейсов модуля авторизации и подключения

Наименование тест-кейса	Выполняемые шаги	Ожидаемый результат
Ввод не валидных данных для подключения	1 Ввести существующий адрес брокера 2 Ввести отрицательное значение порта для подключения к брокеру 3 Нажать кнопку «Add Broker»	Toast-уведомление о некорректном вводе данных
Ввод валидных данных для подключения	1 Ввести существующий адрес брокера 2 Ввести значение порта для подключения к брокеру 3 Нажать кнопку «Add Broker»	Добавление карточки для входа в систему на экране авторизации под формой ввода данных для подключения
Вход в систему	1 Нажать кнопку «Login» на карточке последнего используемого брокера	Отображение главного экрана системы
Удаление данных о последнем используемом брокере	1 Нажать кнопку «Delete» на карточке последнего используемого брокера	Удаление карточки последнего используемого брокера с экрана авторизации

Приведённые тест-кейсы обеспечивают всестороннюю проверку функционирования модуля авторизации и подключения к MQTT-брокеру, включая обработку некорректного ввода, успешное подключение, вход в систему и управление сохранёнными данными о брокерах.

6.2.2 В таблице 6.3 представлены тест-кейсы для проверки функционала отвечающего за взаимодействие с устройствами умного дома. Эти тесты проверяют основные аспекты для работы с физическими

устройствами, включая корректность отправки команд и получение текущего статуса устройств.

Таблица 6.3 – Описание тест-кейсов модуля устройств

Наименование тест-кейса	Выполняемые шаги	Ожидаемый результат
Просмотр списка подключенных устройств	1 Открыть вкладку «Devices»	Отображение списка подключенных устройств к системе
Просмотр детальной информации о конкретном устройстве	1 Открыть вкладку «Devices» 2 Нажать на карточку устройства	Отображение дружественного имени устройства, параметры привязки устройства к комнате, текущие параметры устройства
Изменение дружественного имени устройства в системе	1 Открыть вкладку «Devices» 2 Нажать на карточку устройства 3 Нажать на иконку карандаша 4 Ввести новое имя устройства в форме 5 Нажать кнопку сохранить	Обновленное отображение нового имени устройства во вкладке детальной информации об устройстве и главном экране со списком устройств
Обновление параметра привязки устройства к комнате	1 Открыть вкладку «Devices» 2 Нажать на карточку устройства 3 Нажать на выпадающий список комнат 4 Выбрать новую комнату для привязки устройства	Обновленное отображение комнаты в карточке устройства, устройство перемещено во вкладку новой комнаты на экране «Rooms»
Изменение состояния управляемого устройства	1 Открыть вкладку «Devices» 2 На карточке умной розетки «Plug» нажать на соответствующий виджет	Изменение отображения виджета управления и режима работы умной розетки согласно команде: включено или выключено

По результатам тестирования модуля взаимодействия с устройствами умного дома была подтверждена корректная работа всех основных функций. Проверено правильное отображение списка подключённых устройств и детальной информации о них, а также надёжность управления состоянием устройств. В процессе тестирования успешно выполнены операции по изменению настроек, таким как переименование устройств и обновление привязки к комнатам. Все тесты показали отсутствие ошибок и стабильную работу системы в условиях реального использования.

6.2.3 В таблице 6.4 представлено описание тестовых кейсов, разработанных для проверки функционала распределения устройств по комнатам внутри приложения. Эти тесты направлены на проверку

корректности выполнения операций добавления, удаления и изменения привязки устройств к различным комнатам, а также на обеспечение правильного обновления данных в интерфейсе приложения при изменении конфигурации помещений.

Таблица 6.4 – Описание тест-кейсов модуля комнат

Наименование тест-кейса	Выполняемые шаги	Ожидаемый результат
Добавление новой комнаты в систему	1 Открыть вкладку «Rooms» 2 Нажать на иконку «плюс» в нижнем правом углу экрана 3 Заполнить форму с названием новой комнаты 4 Нажать на кнопку добавить	Добавление карточки новой комнаты во вкладке «Rooms»
Удаление комнаты без привязанных устройств из системы	1 Открыть вкладку «Rooms» 2 Нажать на иконку «крестик» на карточке комнаты к которой привязано 0 устройств	Удаление выбранной карточки комнаты
Удаление комнаты с привязанными устройствами из системы	1 Открыть вкладку «Rooms» 2 Нажать на иконку «крестик» на карточке комнаты к которой привязано 1 и более устройств	Удаление выбранной карточки комнаты, при этом привязка у устройств, ранее связанных с этой комнатой, изменяется на «не привязано»

По результатам тестирования функционала распределения устройств по комнатам все ключевые операции, такие как добавление и удаление комнат, были успешно выполнены. Система корректно обрабатывала удаление комнат с привязанными и без привязанных устройствами

6.2.4 В таблице 6.5 представлен процесс тестирования модуля настроек приложения, включая проверку работы с модулем LED-подсветки и системой обнаружения устройств.

Таблица 6.5 – Описание тест-кейсов модуля настроек

Наименование тест-кейса	Выполняемые шаги	Ожидаемый результат
1	2	3
Добавление устройства в систему	1 Открыть вкладку «Settings» 2 Нажать на переключатель карточки «Enable Discovery Mode» 3 Включить режим сопряжения на устройстве	Отображение добавленного устройства во вкладке «Devices»

Продолжение таблицы 6.5

1	2	3
Включение автоматического режима настройки LED модуля	1 Открыть вкладку «Settings» 2 На карточке «LED Settings» нажать на кнопку «Go to LED Settings» 3 В открывшейся вкладке нажать на выпадающий список и выбрать режим «AUTO»	Отключение модуля подсветки, активность только в случае изменение состояний координатора, например, включение режима обнаружения устройств
Включение ручного режима настройки LED модуля	1 Открыть вкладку «Settings» 2 На карточке «LED Settings» нажать на кнопку «Go to LED Settings» 3 В открывшейся вкладке нажать на выпадающий список и выбрать режим «MANUAL»	Включение модуля подсветки с последними используемыми параметрами в ручном режиме, добавление ползунков для изменения яркости светодиодов и интенсивности каждого из цветов RGB
Изменение параметров подсветки LED модуля в ручном режиме	1 Открыть вкладку «Settings» 2 На карточке «LED Settings» нажать на кнопку «Go to LED Settings» 3 Включить режим «MANUAL» 4 Передвинуть все ползунки в крайнее правое положение	Включение модуля LED с максимальной яркостью и интенсивности цветов

По результатам тестирования модуль настроек приложения работал корректно: режим обнаружения устройств добавлял новые устройства в систему, автоматический режим LED-подсветки («AUTO») активировался при изменении состояния координатора, а ручной режим («MANUAL») обеспечивал точную регулировку яркости и цвета без ошибок.

6.3 Результаты тестирования

В результате проведённого тестирования приложение продемонстрировало стабильную и корректную работу во всех ключевых функциональных областях. Все тесты, включая проверки взаимодействия с устройствами, работу с комнатами, настройки и подсветку LED-модуля, были успешно пройдены без выявления критических ошибок. Таким образом, приложение соответствует заявленным требованиям и готово к использованию в рабочей среде.

7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ И РЕАЛИЗАЦИИ НА РЫНКЕ ANDROID- ПРИЛОЖЕНИЯ ДЛЯ УПРАВЛЕНИЯ И МОНИТОРИНГА УСТРОЙСТВАМИ УМНОГО ДОМА

7.1 Характеристика программного средства, разрабатываемого для реализации на рынке

Созданный дипломный проект представляет собой нативное приложение для операционной системы Android, которое позволяет пользователям управлять устройствами «умного дома» через протокол MQTT для DIY систем.

Целью разработки проекта является упрощение управления устройствами «умного дома» и обработки данных о их состоянии. Приложение предназначено для пользователей, позволяя им эффективно контролировать устройства, настраивать автоматизации и оперативно получать информацию об их работе.

Целевой аудиторией данного приложения являются пользователи систем «умного дома», которым необходим удобный инструмент для управления устройствами через протокол MQTT. Также потенциальными пользователями могут быть энтузиасты DIY-решений, использующие платформы, такие как HomeAssistant, а также владельцы экосистем Aqara, Xiaomi, TuYa и Яндекс.

На момент разработки существует большое количество решений для управления устройствами «умного дома», однако большинство из них ориентированы на конкретные экосистемы, такие как Aqara, Xiaomi, TuYa и Яндекс. Существующие приложения часто имеют ограниченный функционал и не поддерживают интеграцию с DIY-системами.

Планируется распространение приложения через Google Play с возможностью монетизации, включая бесплатную и расширенную платную версии.

7.2 Расчет инвестиций в разработку программного средства

7.2.1 Расчет зарплат на основную заработную плату разработчиков производится исходя из количества людей, которые занимаются разработкой программного продукта, месячной зарплаты каждого участника процесса разработки и сложности выполняемой ими работы. Затраты на основную заработную плату рассчитаны по формуле:

$$Z_o = K_{\text{пр}} \sum_{i=1}^n Z_{\text{чи}} \cdot t_i, \quad (7.1)$$

где n — количество исполнителей, занятых разработкой конкретного ПО;

$K_{пр}$ – коэффициент, учитывающий процент премий;

$Z_{ч,i}$ – часовая заработная плата i -го исполнителя, р.;

t_i – трудоемкость работ, выполняемых i -м исполнителем, ч.

Разработкой всего приложения занимается инженер-программист, обязанности тестирования приложения лежат на инженере-тестировщике. Задачами инженера-программиста, который занимается являются создание модели данных, графического интерфейса, связи между моделью данных и графическим интерфейсом. Инженер-тестировщик занимается выявлением неработоспособных частей приложения.

Месячная заработная плата основана на медианных показателях для Junior инженера-программиста за 2024 год по Республике Беларусь, которая составляет примерно 807 долларов США в месяц, а для Junior инженера-тестировщика – 466 долларов США [11]. По состоянию на 21 февраля 2025 года, 1 доллар США по курсу Национального Банка Республики Беларусь составляет 3,2239 белорусских рубля.

В перерасчете на белорусские рубли месячные оклады для инженера-программиста и инженера-тестировщика составляют 2601,68 и 1502,33 белорусских рублей соответственно.

Часовой оклад исполнителей высчитывается путем деления месячного оклада на количество рабочих часов в месяце, то есть 160 часов.

За количество рабочих часов в месяце для инженера-программиста и инженера-тестировщика принято соответственно 196 и 32 часа. Коэффициент премии приравнивается к единице, так как она входит в общую сумму заработной платы. Затраты на заработную плату приведены в таблице:

Таблица 7.1 – Затраты на основную заработную плату сотрудников

Категория исполнителя	Месячный оклад, р	Часовой оклад, р	Трудоемкость работ, ч	Итого, р
Инженер-программист	2601,68	16,26	196	3186,96
Инженер-тестировщик	1502,33	9,39	32	300,46
Итого				3487,42
Премия и стимулирующие зарплаты (0%)				0
Общие затраты на основную заработную плату разработчиков				3487,42

7.2.2 Расчет затрат на дополнительную заработную плату разработчиков, предусмотренных законодательством о труде, осуществляется по формуле 7.2:

$$Z_d = \frac{Z_o \cdot N_d}{100\%}, \quad (7.2)$$

где Z_o – затраты на основную заработную плату, р.;

N_d – норматив дополнительной заработной платы.

Норматив дополнительной заработной платы был принят равным 10%.

Размер дополнительной заработной составил:

$$З_д = \frac{З_о \cdot Н_д}{100\%} = \frac{3\,487,42 \cdot 10}{100\%} = 348,724 \text{ р.}$$

7.2.3 Расчет отчислений на социальные нужды производился в соответствии с действующими законодательными актами по формуле 7.3:

$$Р_{соц} = \frac{(З_о + З_д) \cdot Н_{соц}}{100\%}, \quad (7.3)$$

где $Н_{соц}$ – норматив отчислений от фонда оплаты труда.

Норматив отчислений от фонда оплаты труда дополнительной заработной платы был принят равным 35%.

Размер дополнительной заработной платы составил:

$$Р_{соц} = \frac{(З_о + З_д) \cdot Н_{соц}}{100\%} = \frac{(3\,487,42 + 348,724) \cdot 35}{100\%} = 1\,342,65 \text{ р.}$$

7.2.4 Расчет затрат на прочие расходы был произведен по формуле 7.4:

$$Р_{пр} = \frac{З_о \cdot Н_{пр}}{100\%}, \quad (7.4)$$

где $Н_{пр}$ – норматив прочих расходов.

Норматив прочих расходов был принят равным 30%.

Размер прочих расходов составил:

$$Р_{пз} = \frac{З_о \cdot Н_{пр}}{100\%} = \frac{3\,487,42 \cdot 30}{100\%} = 1\,046,226 \text{ р.}$$

7.2.5 Расчет расходов на реализацию рассчитан по формуле 7.5:

$$Р_{пр} = \frac{З_о \cdot Н_p}{100\%} \quad (7.5)$$

где $Н_p$ – норматив расходов на реализацию

Норматив расходов на реализацию был принят 3%.

Размер расходов на реализацию составил:

$$Р_{пз} = \frac{З_о \cdot Н_p}{100\%} = \frac{3\,487,42 \cdot 3}{100\%} = 104,6226 \text{ р.}$$

Полная сумма затрат на разработку программного средства представлена в таблице 7.2.

Таблица 7.2 – Полная сумма затрат на разработку программного средства

Наименование статьи затрат	Сумма, р.
Основная заработная плата разработчиков	3 487,42
Дополнительная заработная плата разработчиков	348,72
Отчисления на социальные нужды	1 342,65
Прочие расходы	1 046,22
Расходы на реализацию	104,622
Общая сумма инвестиций (затрат) на разработку	6 329,64

7.3 Расчет экономического эффекта от реализации программного средства на рынке

Для расчета экономического эффекта организации-разработчика программного средства, а именно чистой прибыли, необходимо знать такие параметры как объем продаж, цену реализации и затраты на разработку.

Соответственно необходимо создать обоснование возможного объема продаж, количество проданных лицензий расширенной версии программного средства, купленного пользователями.

В Беларуси проживает около 9,2 миллиона человек, из которых примерно 8,48 миллионов являются активными интернет-пользователями [12]. По данным [13], доля пользователей Android среди мобильных ОС в Беларуси на 2024 год составляет 83%, что делает платформу наиболее популярной среди владельцев смартфонов.

Учитывая, что использование систем «умный дом» требует наличия совместимых устройств, примем, что 3% от общего числа активных интернет-пользователей уже обладают устройствами умного дома. Из них 80% (будут использовать приложения аналоги от производителей устройств, а оставшиеся 20% (42 200 человек) установят разработанное программное средство. Из них 5000 пользователей приобретут расширенную версию программного обеспечения.

С учетом цены на расширенную версию приложения, которая составляет 2,99 долларов США, и с учетом обменного курса доллара к белорусскому рублю, отпускная стоимость программного средства составит примерно 9,62 белорусских рубля.

Для расчета прироста чистой прибыли необходимо учесть налог на добавленную стоимость, который высчитывается по следующей формуле:

$$\text{НДС} = \frac{C_{\text{отп}} \cdot N \cdot H_{\text{д.с}}}{100\% + H_{\text{д.с}}}, \quad (7.6)$$

где N – количество копий программного продукта, реализуемое за год, шт.;

$C_{\text{отп}}$ – отпускная цена копии программного средства, р.;

$H_{\text{д.с}}$ – ставка налога на добавленную стоимость, %.

Ставка налога на добавленную стоимость по состоянию на 15 апреля

2024 года, в соответствии с действующим законодательством Республики Беларусь, составляет 20%. Используя данное значение ставки налога, посчитаем НДС:

$$\text{НДС} = \frac{9,62 \cdot 5\,000 \cdot 20\%}{100\% + 20\%} = 8\,016 \text{ р.}$$

Посчитав налог на добавленную стоимость, можно рассчитать прирост чистой прибыли, которую получит разработчик от продажи программного продукта. Для этого используется формулу:

$$\Delta\Pi_{\text{ч}}^{\text{р}} = (\text{Ц}_{\text{отп}} \cdot N - \text{НДС}) \cdot \text{Р}_{\text{пр}} \cdot \left(1 - \frac{\text{Н}_{\text{п}}}{100}\right), \quad (7.7)$$

где N – количество копий программного продукта, реализуемое за год, шт.;

$\text{Ц}_{\text{отп}}$ – отпускная цена копии программного средства, р.;

НДС – сумма налога на добавленную стоимость, р.;

$\text{Н}_{\text{п}}$ – ставка налога на прибыль, %;

$\text{Р}_{\text{пр}}$ – рентабельность продаж копий.

Ставка налога на прибыль, согласно действующему законодательству с 1 января 2025 равна 20%. Рентабельность продаж копий взята в размере 30%. Зная ставку налога и рентабельность продаж копий (лицензий), рассчитывается прирост чистой прибыли для разработчика:

$$\Delta\Pi_{\text{ч}}^{\text{р}} = (9,62 \cdot 5\,000 - 8\,016) \cdot 30\% \cdot \left(1 - \frac{20}{100}\right) = 9\,620 \text{ р.}$$

7.4 Расчет показателей экономической эффективности разработки и реализации программного средства на рынке

Для того, чтобы оценить экономическую эффективность разработки и реализации программного средства на рынке, необходимо рассмотреть результат сравнения затрат на разработку данного программного продукта, а также полученный прирост чистой прибыли за год после внедрения программного обеспечения.

Сумма затрат на разработку меньше суммы годового экономического эффекта, поэтому можно сделать вывод, что такие инвестиции окупятся менее, чем за один год.

Таким образом, оценка экономической эффективности инвестиций производится при помощи расчета рентабельности инвестиций (Return on Investment, ROI). Формула для расчета ROI:

$$\text{ROI} = \frac{\Delta\Pi_{\text{ч}}^{\text{р}} - \text{З}_{\text{р}}}{\text{З}_{\text{р}}} \cdot 100\%, \quad (7.8)$$

где $\Delta\Pi_{\text{ч}}^p$ – прирост чистой прибыли, полученной от реализации программного средства на рынке информационных технологий, р.;

Z_p – затраты на разработку и реализацию программного средства, р.

$$ROI = \frac{9\,620 - 8\,016}{8\,016} \cdot 100\% = 20\%$$

7.5 Вывод об экономической целесообразности реализации проектного решения

Проведенные расчеты технико-экономического обоснования позволяют сделать предварительный вывод о целесообразности разработки программного продукта для умного дома. Общая сумма затрат на его разработку и реализацию составила 6 329,64 белорусских рублей, а отпускная цена установлена на уровне 9,62 белорусских рублей.

Прогнозируемый прирост чистой прибыли за год, основанный на предполагаемом объеме продаж в размере 5 000 расширенных версий в год, составляет 9 620 белорусских рублей. Рентабельность инвестиций за год оценивается в 20%.

Такие результаты говорят о том, что разработка данного программного продукта является перспективной и имеет экономическое обоснование. Однако, необходимо учитывать возможные риски, связанные с конкуренцией на рынке и возможной недостаточной оценкой продукта со стороны потребителей.

Такой показатель рентабельности может быть приемлем для стабильного бизнеса, но для увеличения доходности следует рассмотреть стратегии оптимизации затрат, повышения ценности продукта для пользователей и расширения рынка сбыта. Дополнительные инвестиции в маркетинг, улучшение функционала и внедрение подписочных моделей могут способствовать росту рентабельности в будущем.

ЗАКЛЮЧЕНИЕ

В рамках дипломного проекта было разработано мобильное приложение для операционной системы Android, предназначенное для мониторинга состояния и управления устройствами умного дома.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] «Умный дом» Xiaomi [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://www.mi.com/ru/smart-home/>. – Дата доступа: 12.02.2025.
- [2] Умный дом с Алисой – комфортный и безопасный [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://alice.yandex.ru/smart-home>. – Дата доступа: 13.02.2025.
- [3] The ZigBee Protocol [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://www.netguru.com/blog/the-zigbee-protocol>. – Дата доступа: 13.02.2025.
- [4] MQTT: The Standart for IoT Messaging [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://mqtt.org>. – Дата доступа: 14.02.2025.
- [5] Supported Devices [Электронный ресурс]. – Электронные данные. – Режим доступа: https://slsys.io/en/action/supported_devices. – Дата доступа: 15.04.2025.
- [6] Install Docker Engine [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://docs.docker.com/engine/install/>. – Дата доступа: 18.04.2025.
- [7] Install the Docker Compose [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://docs.docker.com/compose/install/linux/>. – Дата доступа: 18.02.2025.
- [8] Android Studio [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://developer.android.com/studio>. – Дата доступа: 19.02.2025.
- [9] TS011F Plug [Электронный ресурс]. – Электронные данные. – Режим доступа: https://slsys.io/en/action/supported_devices?device=66. – Дата доступа: 22.04.2025.
- [10] TS0201_TH [Электронный ресурс]. – Электронные данные. – Режим доступа: https://slsys.io/en/action/supported_devices?device=239. – Дата доступа: 22.04.2025.
- [11] Зарплата в IT | dev.by [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://salaries.devby.io/>. – Дата доступа: 23.02.2025.
- [12] Какими смартфонами чаще всего пользуются Белорусы [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://bvn.by/2024/04/15/kakimi-smartfonami-chashhe-vsego-polzujutsja-belorusy>. – Дата доступа: 23.02.2025.
- [13] Статистика интернета и соцсетей на 2024 год – цифры и тренды в Беларуси [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://myfin.by/article/tekhnologii/tiktok-nabiraet-popularnost-u-belorusov-issledovanie>. – Дата доступа: 23.02.2025.
- [14] MQTT [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://en.wikipedia.org/wiki/MQTT>. – Дата доступа: 14.02.2025.
- [15] MQTT Broker: How It Works, Popular Options, and Quickstart

[Электронный ресурс]. – Электронные данные. – Режим доступа: <https://www.emqx.com/en/blog/the-ultimate-guide-to-mqtt-broker-comparison>. – Дата доступа: 15.02.2025.

[16] Eclipse Mosquitto – An open source MQTT broker [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://mosquitto.org/>. – Дата доступа: 15.02.2025.

[18] Comparing Software Architecture Patterns MVC Vs. MVVM Vs. MVP [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://www.masaischool.com/blog/comparing-software-architecture-patterns>. – Дата доступа: 18.02.2025.

[19] Что такое MVVM-шаблон: как работает, преимущества и примеры использования [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://gitverse.ru/blog/articles/development/116-что-такое-mvvm-shablon-kak-rabotaet-preimushchestva-i-primery-ispolzovaniya>. – Дата доступа: 18.02.2025.

[20] Что такое MVC: рассказываем простыми словами [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://ru.hexlet.io/blog/posts/что-такое-mvc-rasskazyvaem-prostymi-slovami>. – Дата доступа: 18.02.2025.

[21] Kotlin Programming Language [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://kotlinlang.org/>. – Дата доступа: 19.02.2025.

[22] Build Better apps faster with Jetpack Compose [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://developer.android.com/compose>. – Дата доступа: 19.02.2025.

[23] Eclipse Paho [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://github.com/eclipse-paho>. – Дата доступа: 10.03.2025.

ПРИЛОЖЕНИЕ А
(обязательное)

Структурная схема

ПРИЛОЖЕНИЕ Б
(обязательное)

Диаграмма классов