МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»

Высшая школа электроники и компьютерных наук
Кафедра «Электронные вычислительные машины»

ДОПУСТИТЬ К ЗАЩИТЕ
Заведующий кафедрой ЭВМ
_____ Д.В. Топольский
«___»_____ 2025 г.

Разработка программно-аппаратного комплекса управления компонентами «умного дома»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ
ЮУРГУ-090301.2025.405 ПЗ ВКР

Руководитель работы,
к. пед. н., доцент каф. ЭВМ
_____ Ю.Г. Плаксина
«___»_____ 2025 г.

Автор работы,
студент группы КЭ-405
_____ Е.Е. Загребнев
«___»_____ 2025 г.

Нормоконтролёр,
ст. преподаватель каф. ЭВМ
_____ С.В. Сяськов
«___»_____ 2025 г.

Челябинск-2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«Южно-Уральский государственный университет
(национальный исследовательский университет)»
Высшая школа электроники и компьютерных наук
Кафедра «Электронные вычислительные машины»

УТВЕРЖДАЮ
Заведующий кафедрой ЭВМ
_____ Д.В. Топольский
«___»_____2025 г.

ЗАДАНИЕ

на выпускную квалификационную работу бакалавра

студенту группы КЭ-405
Загребневу Егору Евгеньевичу
обучающемуся по направлению
09.03.01 «Информатика и вычислительная техника»

1. **Тема работы:** «Программно-аппаратный комплекс управления компонентами "умного дома"» утверждена приказом по университету от «21» апреля 2025 г. №648-13/12 (приложение №9).

2. **Срок сдачи студентом законченной работы:** 1 июня 2025 г.

3. **Исходные данные к работе:**

Функциональные требования:

1. Управление устройствами:

1.1. Возможность изменения параметров исполнительного устройства через веб-интерфейс.

1.2. Поддержка отправки команд через MQTT-брокер.

1.3. Отображение текущего состояния устройства в реальном времени.

2. Взаимодействие с MQTT-брокером:

2.1. Подключение устройства на базе микроконтроллера ESP32 к брокеру с использованием SSL/TLS.

2.2. Публикация состояния устройства в топик Mosquitto /home/device/<id>/state.

2.3. Публикация управляющих команд в топик Mosquitto /home/device/<id>/control.

3. Веб-интерфейс:

3.1. Веб-интерфейс должен поддерживать авторизацию пользователя со своей уникальной учётной записью.

3.2. Веб-интерфейс должен разграничивать устройства пользователей по признаку принадлежности.

3.3. Хранение пользовательских данных осуществляется при помощи базы данных PostgreSQL.

3.4. Графический интерфейс разрабатывается при помощи системы компонентов Chakra UI.

3.5. Серверная часть разрабатывается при помощи языка программирования Python.

Нефункциональные требования:

1. Производительность:

1.1. Время отклика на команду устройства «умного дома» не более 600 мс при штатной нагрузке (до 5 команд в секунду).

2. Надёжность:

2.1. Автоматическое восстановление соединения устройства с сервером не более, чем через 10 с после сбоя.

3. Безопасность:

3.1. Аутентификация пользователя через уникальный логин и пароль.

3.2. Ограничение доступа по IP-адресу.

4. Развёртывание:

4.1. Поддержка работы серверного модуля в условиях контейнеризации средствами Docker.

4. **Перечень подлежащих разработке вопросов:**

1. Аналитический обзор научно-технической, нормативной и методической литературы.
2. Проектирование архитектуры комплекса управления компонентами «умного дома».
3. Разработка серверного программного модуля и программного модуля для устройства «умного дома».
4. Проведение тестирования программных модулей.

5. **Дата выдачи задания:** 2 декабря 2024 г.

Руководитель работы _____/ *Ю. Г. Плаксина* /

Студент _____/ *Е. Е. Загребнев* /

КАЛЕНДАРНЫЙ ПЛАН

Этап	Срок сдачи	Подпись руководителя
Аналитический обзор научно-технической, нормативной и методической литературы	03.03.2025	
Проектирование архитектуры программно-аппаратного комплекса управления компонентами «умного дома»	22.03.2025	
Разработка серверного программного модуля и программного модуля для демонстрационного устройства «умного дома»	12.04.2025	
Проведение тестирования программных модулей	26.04.2025	
Компоновка текста работы и сдача на нормоконтроль	22.05.2025	
Подготовка презентации и доклада	30.05.2025	

Руководитель работы _____/ *Ю. Г. Плаксина* /

Студент _____/ *Е. Е. Загребнев* /

Аннотация

Е.Е. Загребнев. Разработка программно-аппаратного комплекса управления компонентами «умного дома». – Челябинск: ФГАОУ ВО «ЮУрГУ (НИУ)», ВШ ЭКН; 2025, 96 с., 14 ил., библиогр. список – 18 наим.

Настоящая работа посвящена разработке программно-аппаратного комплекса управления компонентами «умного дома». В ходе исследования проведён анализ современных решений в области автоматизации жилых помещений, рассмотрены существующие программные и аппаратные платформы, а также выявлены их основные недостатки, связанные с ограниченной гибкостью, сложностью интеграции и высокой стоимостью.

В работе предложена собственная архитектура комплекса, включающая серверную часть на базе FastAPI, клиентское веб-приложение на React, коммуникационный слой на протоколе MQTT и демонстрационное устройство. Разработанный комплекс отличается модульностью и простотой развертывания благодаря использованию контейнеризации Docker. Проведено тестирование работоспособности системы, подтверждена её надёжность и возможность дальнейшего расширения функциональности.

Результаты работы могут быть использованы для внедрения современных технологий автоматизации в жилых и коммерческих помещениях, а также послужить основой для дальнейших исследований и разработок в области «умного дома».

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Современное общество стремительно развивается в направлении цифровизации и автоматизации различных сфер жизни. Тренд на привлечение вычислительной техники для решения задач автоматизации затронул как производственные процессы, так и бытовые, породив при этом самостоятельную отрасль под названием «интернет вещей» (англ. Internet of Things, IoT), в домен которой входит такое понятие, как «умный дом» – жилое помещение, оснащённое системами, способными обеспечивать комфорт, безопасность и энергоэффективность за счёт автоматического управления такими элементами бытия, как, например, освещение, климат, электроприборы и системы безопасности. Системы управления компонентами «умного дома» стали ключевым элементом реализации этой концепции.

Актуальность темы настоящей работы обусловлена растущей потребностью в интеграции современных технологий в повседневную жизнь [1], что позволяет повысить качество жизни, снизить энергозатраты и обеспечить удобство управления домашними системами. Кроме того, развитие «интернета вещей» и рост доступности различных сенсоров и исполнительных устройств создают благоприятные условия для создания комплексных систем управления.

Целью данной выпускной квалификационной работы является разработка программно-аппаратного комплекса, обеспечивающего управление компонентами «умного дома». В рамках работы планируется исследовать существующие решения, определить требования к системе, разработать аппаратную часть и программное обеспечение, а также провести тестирование и оценку эффективности созданного комплекса.

Результаты работы могут быть использованы как основа для внедрения современных технологий автоматизации в жилых помещениях.

1. АНАЛИТИЧЕСКИЙ ОБЗОР НАУЧНО-ТЕХНИЧЕСКОЙ, НОРМАТИВНОЙ И МЕТОДИЧЕСКОЙ ЛИТЕРАТУРЫ

1.1. Понятие «умного дома»

Концепция «умного дома» представляет собой интегрированную систему автоматизации жилого пространства, которая обеспечивает удалённое и автоматическое управление инженерными системами и бытовыми устройствами с целью повышения комфорта, безопасности и энергоэффективности проживания. Термин «умный дом» впервые появился в конце XX века и с тех пор претерпел значительное развитие, отражая эволюцию технологий и потребностей пользователей.

В научной литературе умный дом определяется как комплекс аппаратных и программных средств, объединённых в единую сеть, которая позволяет контролировать и управлять освещением, климатом, системами безопасности, бытовой техникой и другими компонентами жилого помещения [2]. Основная задача таких систем - автоматизация рутинных процессов и обеспечение адаптивного реагирования на изменения внешних и внутренних условий.

Согласно ГОСТ Р 56508-2015 «Системы автоматизации зданий. Термины и определения», умный дом – это «здание с интегрированной системой управления инженерными системами и оборудованием, обеспечивающей автоматизацию функций и удалённое управление» [3].

Ключевыми характеристиками умного дома являются:

– интеграция различных подсистем в единую платформу [4];

– автоматизация процессов на основе данных с датчиков и заданных сценариев [5];

– удалённый доступ и управление через мобильные устройства или интернет [6];

– персонализация настроек под предпочтения пользователей [7];

– энергоэффективность за счёт оптимального использования ресурсов [8].

В последние годы развитие «интернета вещей» значительно расширило возможности умных домов, позволяя подключать к системе большое количество разнообразных устройств и обеспечивать их взаимодействие в реальном времени.

Понятие умного дома охватывает не только техническую инфраструктуру, но и концептуальные подходы к созданию комфортной, безопасной и энергоэффективной среды проживания, основанной на современных технологиях автоматизации и связи.

1.2. Обзор современных научно-технических исследований и разработок в области систем «умных домов»

Современные исследования в области умных домов сосредоточены на внедрении и развитии технологий искусственного интеллекта для повышения функциональности и адаптивности систем автоматизации жилых помещений. В отличие от базового понятия умного дома, которое определяет его как интегрированную систему управления инженерными и бытовыми устройствами, научно-технические разработки направлены на создание интеллектуальных платформ, способных к самостоятельному обучению и прогнозированию поведения пользователей.

Одним из основных направлений является использование искусственного интеллекта (ИИ) для автоматизации рутинных задач и оптимизации энергопотребления. Например, алгоритмы машинного обучения анализируют данные с датчиков и поведения жильцов, позволяя системе адаптировать работу освещения, отопления и кондиционирования в реальном времени, что снижает энергозатраты и повышает комфорт [9]. При этом ИИ

способен выявлять аномалии в работе оборудования и обеспечивать проактивное реагирование на потенциальные угрозы безопасности.

Развитие голосовых интерфейсов и интеллектуальных ассистентов (Amazon Alexa, Google Assistant, «Алиса») значительно расширяет возможности управления умным домом, делая его более доступным и удобным для пользователей. Голосовое управление становится центральным элементом взаимодействия, позволяя управлять устройствами без физического контакта [10].

Интеграция IoT-устройств обеспечивает обмен данными между многочисленными компонентами системы, создавая единую экосистему, где сенсоры, исполнительные механизмы и управляющие модули работают слаженно и в режиме реального времени. Современные разработки уделяют особое внимание вопросам кибербезопасности и защите персональных данных, применяя методы шифрования и аутентификации, чтобы предотвратить несанкционированный доступ и обеспечить конфиденциальность информации.

Кроме того, исследования направлены на создание интеллектуальных кухонь и бытовой техники с ИИ, которые способны самостоятельно управлять процессами приготовления пищи и контролировать запасы продуктов, что повышает удобство и экономит время пользователей.

1.3. Анализ существующих программно-аппаратных комплексов и систем автоматизации домашнего пространства

Современный рынок систем автоматизации домашнего пространства представлен широким спектром программно-аппаратных комплексов, которые отличаются по архитектуре, функциональности, способам управления и уровню интеграции с другими устройствами. В России и мире наибольшее распространение получили как закрытые экосистемы крупных

компаний, так и открытые платформы с возможностью гибкой настройки и расширения.

Одним из ярких представителей закрытых систем является «Умный дом Sber» – экосистема компании Сбербанк, в которой управление осуществляется через виртуальных ассистентов семейства «Салют». Система позволяет контролировать освещение, микроклимат и создавать сценарии для одного или группы устройств. Управление возможно как голосом, так и через мобильное приложение «Сбер Салют». Преимуществом решения от «Сбера» является автономная работа при отсутствии интернет-соединения, однако высокая стоимость оборудования и относительно молодой возраст продукта ограничивают его массовое распространение [11].

«Умный дом», построенный на решениях от компании «Яндекс», представляет собой более открытый вариант, построенный вокруг голосового помощника «Алиса». Отсутствие базовой станции упрощает установку, а низкая стоимость и возможность подключения устройств сторонних производителей делают систему доступной. Основным недостатком разработки компании «Яндекс» является полная зависимость от облачных сервисов: при отсутствии интернета управление становится невозможным.

В противоположность закрытым решениям, популярностью пользуются открытые платформы, такие как Home Assistant. Эта система с открытым исходным кодом может быть установлена на различных платформах (Raspberry Pi, Windows, Linux и др.) и обеспечивает локальное управление устройствами без необходимости постоянного подключения к интернету. Home Assistant поддерживает множество интеграций с устройствами различных производителей, что позволяет пользователям самостоятельно формировать конфигурацию умного дома. К недостаткам относятся сложность первоначальной настройки и ограниченный удалённый доступ.

Среди профессиональных решений выделяется система Crestron, ориентированная на комплексное управление аудио-видео оборудованием,

освещением, шторами, микроклиматом и системами безопасности. Одним из ключевых достоинств систем компании Crestron является широкий выбор пользовательских интерфейсов: сенсорные панели, радио- и ИК-пульты, что обеспечивает удобство эксплуатации в различных сценариях [12].

Современные системы автоматизации строятся на основе централизованных контроллеров, которые связывают датчики и исполнительные устройства, обеспечивая взаимодействие и управление. Контроллеры могут работать локально или через облачные сервисы, что влияет на доступность функций при отсутствии интернет-соединения. Беспроводные протоколы передачи данных (ZigBee, Z-Wave, Bluetooth) обеспечивают энергоэффективность и простоту установки, в то время как проводные системы требуют профессионального монтажа и подходят преимущественно для новых построек [13].

На российском рынке также представлены гибридные решения, такие как Rubetek, предлагающие собственные устройства и совместимость с техникой других производителей. Центр управления Rubetek позволяет создавать сценарии и объединять устройства в группы, обеспечивая комплексную автоматизацию.

Современный ландшафт систем умного дома характеризуется разнообразием архитектурных решений: от закрытых экосистем с упором на удобство и интеграцию до открытых платформ с высокой степенью кастомизации. Выбор конкретного комплекса зависит от требований к функциональности, бюджету, уровню технической подготовки пользователя и условий эксплуатации.

1.4. Обзор нормативных документов и стандартов

В последние годы в Российской Федерации разработана и утверждена серия национальных стандартов, направленных на регулирование

проектирования, создания и эксплуатации систем умного дома в рамках концепции киберфизических систем. Эти стандарты обеспечивают единые требования к архитектуре, функциональности, безопасности и совместимости компонентов умных домов, что способствует формированию единого рынка и повышению качества цифровых жилых комплексов.

Так, ГОСТ Р 71200-2023 «Системы киберфизические. Умный дом. Общие положения» устанавливает базовые принципы и типовую структуру систем умного дома, включая информационные системы, внутридомовые и внутриквартирные подсистемы [14]. Помимо этого, стандарт регламентирует совместимость оборудования различных производителей и определяет этапы создания систем автоматизации, начиная с разработки технического задания и заканчивая эксплуатацией и модернизацией.

ГОСТ Р 71199-2023 «Системы киберфизические. Умный дом. Термины и определения» обеспечивает единообразие терминологии и понятийной базы в области умных домов [15].

ГОСТ Р 71866–2024 «Системы киберфизические. Умный дом. Общие технические требования к автоматизированным системам управления зданием (АСУЗ)» детализирует требования к архитектуре, классам устройств, базовому набору оборудования, системам управления освещением, многоабонентским домофонам и другим компонентам [16].

В стандартах подробно прописаны требования к базовому набору устройств умного дома, который включает системы автоматизации освещения, отопления, водоснабжения, электроснабжения, а также инженерной и физической безопасности. Обязательным элементом являются IP-домофоны, видеокамеры, системы контроля и управления доступом, датчики протечек, температуры, газа, пожарные извещатели и другие сенсоры, обеспечивающие безопасность и комфорт жильцов.

Особое внимание уделяется цифровой архитектуре умного дома, предусматривающей использование мобильных приложений, веб-

интерфейсов, голосового управления и интерактивных цифровых поверхностей для взаимодействия пользователей с системой. Стандарты также регламентируют этапы создания умных домов, включая проектирование, монтаж, пуско-наладочные работы, испытания, обучение персонала и эксплуатацию.

Внедрение данных нормативных документов способствует не только техническому развитию рынка умных домов, но и формированию прозрачных требований для застройщиков и производителей оборудования. В частности, применение стандартов становится обязательным при выполнении государственных и муниципальных заказов, а также служит основой для присвоения классам умных многоквартирных домов.

Нормативно-правовая база в России по системам умного дома сегодня представляет собой комплекс взаимосвязанных стандартов, обеспечивающих единые технические, функциональные и организационные требования, что способствует развитию высокотехнологичного и безопасного жилого пространства.

1.5. Методические подходы к проектированию и разработке систем управления «умным домом»

Проектирование и разработка систем управления умным домом требуют комплексного и системного подхода, включающего анализ требований, выбор архитектуры, протоколов связи, обеспечение безопасности и планирование масштабируемости системы.

На первом этапе осуществляется сбор функциональных и нефункциональных требований [17]. Важно определить, какие функции должна выполнять система умного дома, такие как управление освещением, климат-контролем, системами безопасности и другими подсистемами. При этом необходимо учитывать ограничения, связанные с условиями

эксплуатации, включая особенности электроснабжения, требования к безопасности, производительности и совместимости с существующей инфраструктурой. Одновременно с этим формулируются пользовательские сценарии без привязки к конкретным технологиям, чтобы глубже понять реальные потребности жильцов и избежать ненужной автоматизации. При проектировании также следует учитывать системные требования, уделяя особое внимание надежности, отказоустойчивости и возможностям масштабирования системы.

Выбор архитектурного подхода является следующим важным этапом. Возможны различные варианты организации системы: централизованная архитектура, при которой все устройства подключены к центральному контроллеру; децентрализованная архитектура, в которой устройства взаимодействуют напрямую друг с другом; а также гибридные решения, сочетающие преимущества обоих подходов. Определение архитектуры зависит от требований к надежности, масштабируемости и удобству эксплуатации системы [18].

На этапе выбора протоколов связи и оборудования принимаются во внимание особенности различных технологий, таких как Zigbee, Z-Wave, Wi-Fi, Bluetooth и других. Каждый стандарт обладает своими преимуществами и ограничениями, в том числе по дальности передачи данных, энергопотреблению и уровню надежности. Для эффективной работы системы важно обеспечить совместимость оборудования от разных производителей и предусмотреть возможность интеграции всех компонентов в единую платформу управления.

Разработка алгоритмов управления и программного обеспечения включает создание центрального блока обработки данных, который получает команды от пользователя и сигналы от датчиков, а затем формирует управляющие воздействия на исполнительные устройства. Программное обеспечение должно поддерживать многозадачный режим работы,

обеспечивать удобный пользовательский интерфейс через веб-приложения, мобильные устройства или стационарные панели управления, а также иметь механизм регулярного обновления. Дополнительно рекомендуется включить функции аналитики и визуализации данных для повышения удобства эксплуатации и упрощения диагностики работы системы.

Особое внимание уделяется обеспечению безопасности системы. Применяются методы многоуровневой защиты, включая шифрование передаваемых данных, внедрение двухфакторной аутентификации и регулярное обновление программного обеспечения. Для повышения надежности предусматривается наличие резервных каналов управления на случай отказа связи или отключения интернета. Кроме того, необходимо изолировать сеть умных устройств от основной домашней сети с целью минимизации рисков несанкционированного доступа.

Планирование масштабируемости и удобства эксплуатации требует от проектировщика предусмотреть возможность лёгкого добавления новых устройств и расширения функциональности без необходимости полной реконструкции системы. Важной задачей является также обучение пользователей правильной работе с системой, создание подробных инструкций и проведение практических занятий для повышения эффективности использования умного дома. При проектировании следует заранее предусмотреть конфигурацию прокладки кабелей и размещение оборудования таким образом, чтобы обеспечить возможности для будущего расширения инфраструктуры.

Процесс проектирования начинается с этапа сбора и анализа данных об объекте и потребностях заказчика. На основании этой информации создается эскизный проект, который включает предварительную оценку бюджета и согласование технических решений с заказчиком. Далее разрабатывается рабочий проект с детализированной схемой размещения оборудования и календарным планом выполнения работ. После согласования начинается этап

выбора и закупки необходимого оборудования, его установки и настройки. Завершающим этапом является тестирование системы, обучение пользователей и ввод системы в эксплуатацию.

Методические подходы к проектированию и разработке систем управления умным домом основаны на системном анализе требований, выборе оптимальной архитектуры и технологий, обеспечении безопасности и масштабируемости, а также акценте на удобство эксплуатации и обучении пользователей. Применение такого комплексного подхода позволяет создавать эффективные, надежные и адаптивные решения, максимально соответствующие потребностям жильцов.

1.6. Обзор проблем при производстве «умных домов» и путей их решения

Системы умного дома, несмотря на их высокую привлекательность и удобство, сталкиваются с рядом значительных проблем и вызовов, которые влияют на эффективность функционирования, уровень безопасности и комфорт использования.

Одной из ключевых проблем является отсутствие единых стандартов и совместимости между устройствами и системами. Различные производители применяют собственные протоколы и технологии, что значительно усложняет процесс интеграции оборудования различных брендов в единую систему. Это приводит к затруднениям в масштабировании умного дома и ограничивает выбор доступных решений для пользователей.

Безопасность систем умного дома также остается одной из самых острых проблем. Существующие технологии подвержены угрозам взлома, что может привести к несанкционированному доступу к личной информации жильцов, системам видеонаблюдения и управлению различными устройствами внутри дома. Кроме того, беспроводные системы могут подвергаться риску

заглушения или перехвата радиосигналов, что создает дополнительные угрозы для безопасности проживания.

Серьезным вызовом является и сложность установки и настройки систем. Монтаж умного дома требует значительных временных и финансовых затрат, особенно в случаях, когда речь идет о внедрении систем в уже построенные и отремонтированные объекты. Необходимость прокладки кабельных сетей, установки датчиков и источников бесперебойного питания может вызывать неудобства для жильцов и влечь за собой дополнительные расходы.

Надежность оборудования также вызывает опасения. Часто наблюдаются сбои и поломки в работе умных устройств, что может быть связано как с использованием некачественных компонентов, так и с ошибками на этапе установки. Нестабильная работа системы снижает уровень доверия со стороны пользователей и требует дополнительных вложений в ремонт и техническое обслуживание.

Еще одной проблемой является несоответствие ожидаемой экономической эффективности реальным результатам. Хотя умные дома часто позиционируются как способ экономии электроэнергии, некоторые устройства потребляют значительное количество ресурсов, что нивелирует потенциальную экономию. Дополнительные неудобства вызывает также необходимость регулярной замены батареек в беспроводных датчиках.

Быстрое устаревание технологий усиливает сложность эксплуатации умных домов. Стремительное развитие отрасли приводит к тому, что устройства быстро теряют актуальность, и пользователи вынуждены регулярно обновлять оборудование для сохранения его функциональности и совместимости с новыми стандартами.

В многоквартирных домах нередко наблюдаются проблемы с качеством освещения и инженерных систем. Использование дешевых и несовместимых светильников и управляющих устройств приводит к нарушению эстетики

интерьера, появлению эффекта мерцания света и необходимости частой замены оборудования за счет жильцов.

Несмотря на очевидную перспективность и удобство применения, системы умного дома сталкиваются с целым комплексом проблем, связанных с безопасностью, совместимостью оборудования, надежностью функционирования, энергоэффективностью и сложностью внедрения. Для повышения качества и доступности умных домов необходимо продолжение научных исследований, развитие стандартов и совершенствование технологий.

Для преодоления указанных проблем в сфере разработки и эксплуатации систем умного дома представляется необходимым реализация целого ряда комплексных решений.

Во-первых, важно продолжать работу над унификацией протоколов связи и разработкой межплатформенных стандартов. Создание отраслевых соглашений и переход к открытым интерфейсам позволят обеспечить совместимость устройств различных производителей, тем самым облегчая интеграцию и расширение функциональности комплекса. Инициативы по стандартизации должны поддерживаться как государственными структурами, так и отраслевыми альянсами.

Во-вторых, следует уделить особое внимание вопросам кибербезопасности. Повышение уровня защищённости систем возможно за счёт внедрения современных методов аутентификации (двухфакторной, по сертификатам), шифрования передаваемых данных (TLS), а также регулярного обновления прошивок устройств. Разработка систем автоматического мониторинга подозрительной активности и ограничение доступа по IP-адресам способны снизить риск несанкционированного вмешательства.

Снижение затрат и повышение удобства внедрения возможно благодаря развитию беспроводных технологий нового поколения (например, Wi-Fi 6,

Thread, Bluetooth LE Mesh) и созданию модульных, легко масштабируемых систем, не требующих прокладки кабелей. Отказ от избыточной проводной инфраструктуры и ориентация на беспроводные датчики с длительным сроком службы аккумуляторов позволят упростить установку оборудования, особенно в уже эксплуатируемых помещениях.

Для повышения надёжности оборудования необходимо применение проверенных микросхем, сертифицированных источников питания, а также использование встроенных механизмов самотестирования и восстановления. Производители должны обеспечивать прозрачную техническую поддержку, а проектировщики – предусматривать резервные сценарии работы и возможность локального управления при сбоях.

Решение проблемы экономической неэффективности возможно через использование умных сценариев автоматизации, позволяющих учитывать поведение пользователя, наличие людей в помещении, динамику энерготарифов. Интеграция с учётными системами потребления и автоматизация отключения ненагруженных линий значительно повышают эффективность расходования ресурсов.

Борьба с устареванием оборудования предполагает поддержку модульности и прошивок с возможностью обновления по воздуху (OTA). Выбор оборудования с длительным жизненным циклом и поддержкой обновлений позволяет сократить расходы и обеспечить совместимость с новыми компонентами в будущем.

Особое внимание должно быть уделено вопросам эргономики и качества компонентов освещения, особенно в многоквартирных жилых зданиях. Использование светильников с контролируемой цветовой температурой и индексом цветопередачи, поддерживающих стандартные протоколы управления, позволяет избежать визуальных искажений, мерцания и нарушений эстетики интерьера.

Решение актуальных проблем, связанных с системами умного дома, требует не только технологических инноваций, но и системного подхода, охватывающего стандартизацию, безопасность, удобство эксплуатации и устойчивость к изменениям во времени. Лишь при комплексной проработке этих аспектов возможно формирование по-настоящему надёжной, доступной и масштабируемой инфраструктуры «умного» жилого пространства.

1.7. Выводы по итогам проведённого обзора литературы

В результате проведённого аналитического обзора научно-технической, нормативной и методической литературы были рассмотрены ключевые понятия и современные тенденции в области систем «умного дома». Проанализированы существующие программно-аппаратные комплексы, их архитектурные особенности, преимущества и недостатки, а также выявлены основные проблемы.

Особое внимание было уделено нормативной базе, регулирующей проектирование и эксплуатацию систем автоматизации жилых помещений, что позволило определить требования к архитектуре и функциональности разрабатываемого комплекса. Были рассмотрены методические подходы к проектированию систем управления «умным домом», включающие анализ требований, выбор архитектуры, протоколов связи и обеспечение масштабируемости.

Проведённый анализ показал, что несмотря на наличие на рынке как закрытых, так и открытых решений, ни одно из них не лишено недостатков, связанных с ограниченной гибкостью, сложностью интеграции и высокой стоимостью. Это подтверждает актуальность разработки собственного программно-аппаратного комплекса, ориентированного на открытость, модульность, простоту развертывания и возможность дальнейшего расширения.

2. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ПРОГРАММНО-АППАРАТНОГО КОМПЛЕКСА УПРАВЛЕНИЯ КОМПОНЕНТАМИ «УМНОГО ДОМА»

2.1. Общая архитектура программно-аппаратного комплекса

Программно-аппаратный комплекс управления компонентами «умного дома» представляет собой распределённую систему, архитектура которой строится по принципу клиент–серверного взаимодействия с использованием брокера сообщений. Взаимодействие между пользователем, управляющим программным обеспечением и физическими исполнительными устройствами осуществляется с применением унифицированных протоколов и сервисов, развёрнутых в контейнеризованной среде.

Функциональное ядро комплекса сосредоточено в серверной части, включающей в себя веб-сервер приложений, брокер сообщений и систему управления базами данных. Эти элементы взаимодействуют внутри изолированной виртуальной сети, формируемой с использованием средств оркестрации контейнеров. Серверная часть отвечает за обработку пользовательских запросов, маршрутизацию команд к периферийным устройствам, приём и хранение телеметрической информации, извлечение пользовательских и конфигурационных данных, а также за формирование представления информации для пользовательского интерфейса.

Клиентская часть реализована в виде одностраничного веб-приложения, обеспечивающего графическое представление состояния системы, визуализацию истории, а также возможность ручного управления подключёнными компонентами. Взаимодействие между клиентским интерфейсом и серверным приложением осуществляется через

стандартизированный программный интерфейс прикладного уровня, построенный на основе REST.

Ключевым элементом коммуникации между сервером и микроконтроллерными устройствами выступает брокер сообщений, реализующий протокол MQTT. Через него осуществляется передача управляющих команд и телеметрических данных, что обеспечивает устойчивость взаимодействия и возможность масштабирования за счёт логической изоляции компонент. Микроконтроллеры, интегрированные в состав конечных устройств, подписываются на соответствующие топики брокера и реагируют на поступающие команды, а также передают обратно информацию о своём состоянии.

Все элементы программно-аппаратного комплекса развернуты в виде контейнеризованных сервисов, что позволяет обеспечить воспроизводимость среды, упрощённое управление зависимостями и независимую разработку отдельных компонентов системы. Используемая инфраструктура обеспечивает сетевую связанность всех сервисов, а также доступность необходимых ресурсов для их функционирования.

Схематическое представление общей архитектуры комплекса приведено на рисунке 1, созданном при помощи пакета программного обеспечения PlantUML.

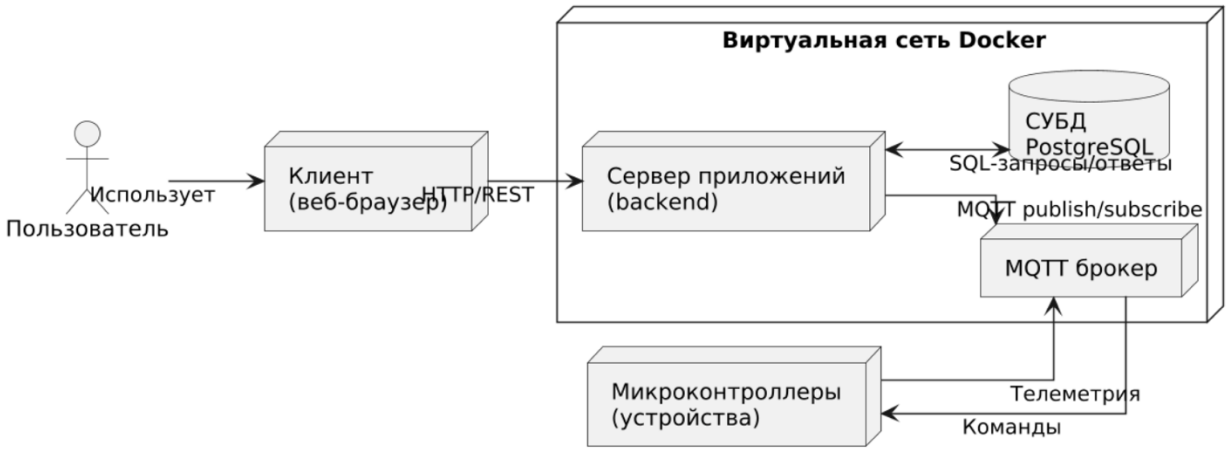

Рисунок 1 – Архитектура программно-аппаратного комплекса
(схема создана при помощи пакета программного обеспечения PlantUML)

2.2. Архитектура серверного (backend) приложения

Серверное приложение программно-аппаратного комплекса реализует ключевую логику управления системой и выступает в роли промежуточного звена между пользовательским интерфейсом, брокером сообщений и системой управления базами данных. В рамках реализованной архитектуры это приложение выполняет функции приёма и обработки входящих HTTP-запросов, маршрутизации управляющих команд, обработки телеметрических данных и предоставления клиентскому интерфейсу актуального состояния компонентов «умного дома».

В качестве технологической основы серверного приложения выбран асинхронный веб-фреймворк FastAPI, обеспечивающий высокую производительность при работе с большим числом параллельных запросов. Логика приложения структурирована на основе модульного подхода, при котором функциональные блоки – обработчики API-запросов, обработчики MQTT-сообщений, а также модули взаимодействия с базой данных – вынесены в отдельные независимые части кода, что повышает читаемость, масштабируемость и сопровождаемость системы.

Для организации хранения и извлечения информации серверне приложение использует объектно-реляционное отображение с помощью библиотеки SQLAlchemy. Это позволяет абстрагироваться от низкоуровневых SQL-запросов, соблюдая при этом строго типизированную схему взаимодействия с базой данных. Подключение к СУБД осуществляется через конфигурируемый пул соединений, параметры которого задаются посредством переменных окружения и адаптированы под условия контейнерного исполнения. Сервер обрабатывает как запросы на запись (например, при фиксации телеметрических событий), так и запросы на чтение (включая получение пользовательских данных, истории состояний и текущей конфигурации устройств).

Интеграция серверного приложения с брокером сообщений осуществляется через клиентскую библиотеку для протокола MQTT, реализующую как подписку на топики, так и публикацию сообщений. Таким образом, серверная часть способна не только передавать управляющие команды конечным устройствам, но и реагировать на поступающие от них сигналы, обновляя состояние системы в реальном времени. При получении MQTT-сообщений, содержащих информацию о текущем состоянии оборудования, сервер осуществляет их сбор, валидацию и последующую запись в базу данных, обеспечивая целостность и хронологическую достоверность собираемой информации.

Контейнеризация серверного приложения обеспечивает его изоляцию и независимость от окружения, в котором развёрнута система. Docker-образ сервиса включает в себя все необходимые зависимости, а запуск приложения осуществляется с учётом заданных переменных конфигурации, включающих параметры доступа к системе управления базами данных (СУБД) и MQTT-брокеру. Вся межсервисная коммуникация внутри системы осуществляется через виртуальную сетевую инфраструктуру Docker Compose, что позволяет

гарантировать безопасность и управляемость взаимодействий между компонентами.

Схематическое представление внутренней архитектуры серверного приложения, отражающее основные модули и их связи, приведено на рисунке 2, созданном при помощи пакета программного обеспечения PlantUML.

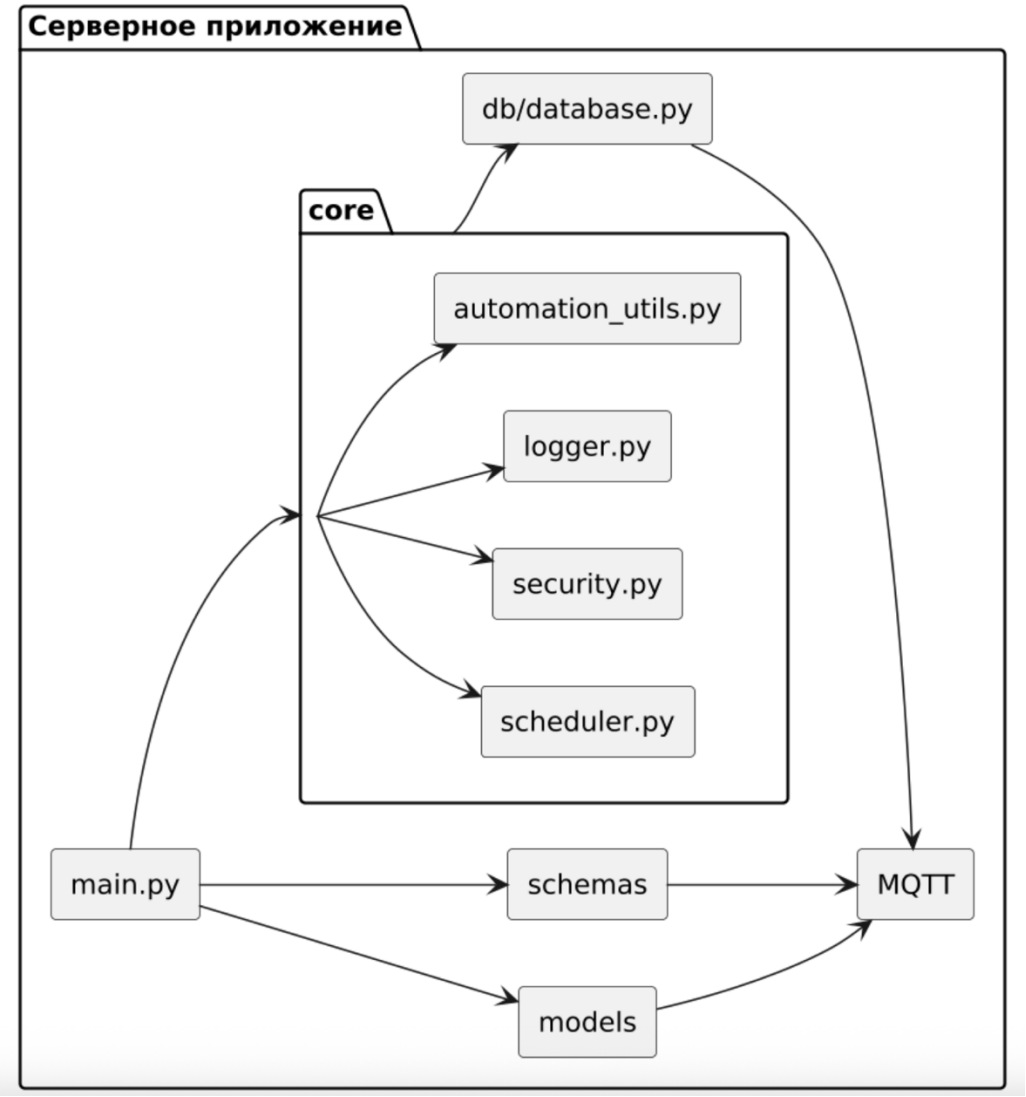

Рисунок 2 – Архитектура серверного приложения
(схема создана при помощи пакета программного обеспечения PlantUML)

2.3. Архитектура клиентского (frontend) приложения

Клиентская часть программно-аппаратного комплекса реализована в виде одностраничного веб-приложения, основанного на библиотеке React. Основное назначение клиентского приложения заключается в обеспечении визуального взаимодействия пользователя с системой, включая отображение текущего состояния компонентов, просмотр истории изменений, а также управление исполнительными устройствами.

В рамках архитектурной модели одностраничного приложения реализуется концепция динамического обновления интерфейса без необходимости перезагрузки страницы. Это достигается путём программной маршрутизации, позволяющей изменять содержимое представления в зависимости от действия пользователя, при сохранении единой точки входа в систему. Благодаря этому взаимодействие с системой приобретает интерактивный и отзывчивый характер, что особенно важно в контексте управления оборудованием в реальном времени.

Клиентское приложение организует обмен данными с серверной частью через интерфейс прикладного уровня, реализованный в формате REST. Все обращения к серверному приложению выполняются посредством асинхронных HTTP-запросов, что позволяет минимизировать время отклика и повысить отзывчивость пользовательского интерфейса. Данные, полученные от сервера, обрабатываются и отображаются в виде визуальных компонентов, отражающих актуальное состояние соответствующих элементов системы.

Архитектура клиентского приложения построена на компонентной модели, в которой каждый элемент интерфейса представлен как независимый модуль с собственным состоянием и логикой отображения. Компоненты объединяются в иерархическую структуру, обеспечивающую логическое разделение функциональных областей интерфейса: панель управления, список устройств, история событий, настройки и прочие элементы. Для управления

состоянием приложения используются встроенные механизмы React, включая хуки и контекст, что позволяет централизованно контролировать данные, передаваемые между компонентами.

Асинхронность взаимодействия и подписка на изменения состояния обеспечивают возможность динамического обновления информации без необходимости ручного вмешательства со стороны пользователя. Это особенно актуально при отображении телеметрических данных или обратной связи от устройств, работающих в режиме реального времени.

Схематическое представление внутренней архитектуры клиентского приложения, отражающее основные модули и их связи, приведено на рисунке 3, созданном при помощи пакета программного обеспечения PlantUML.

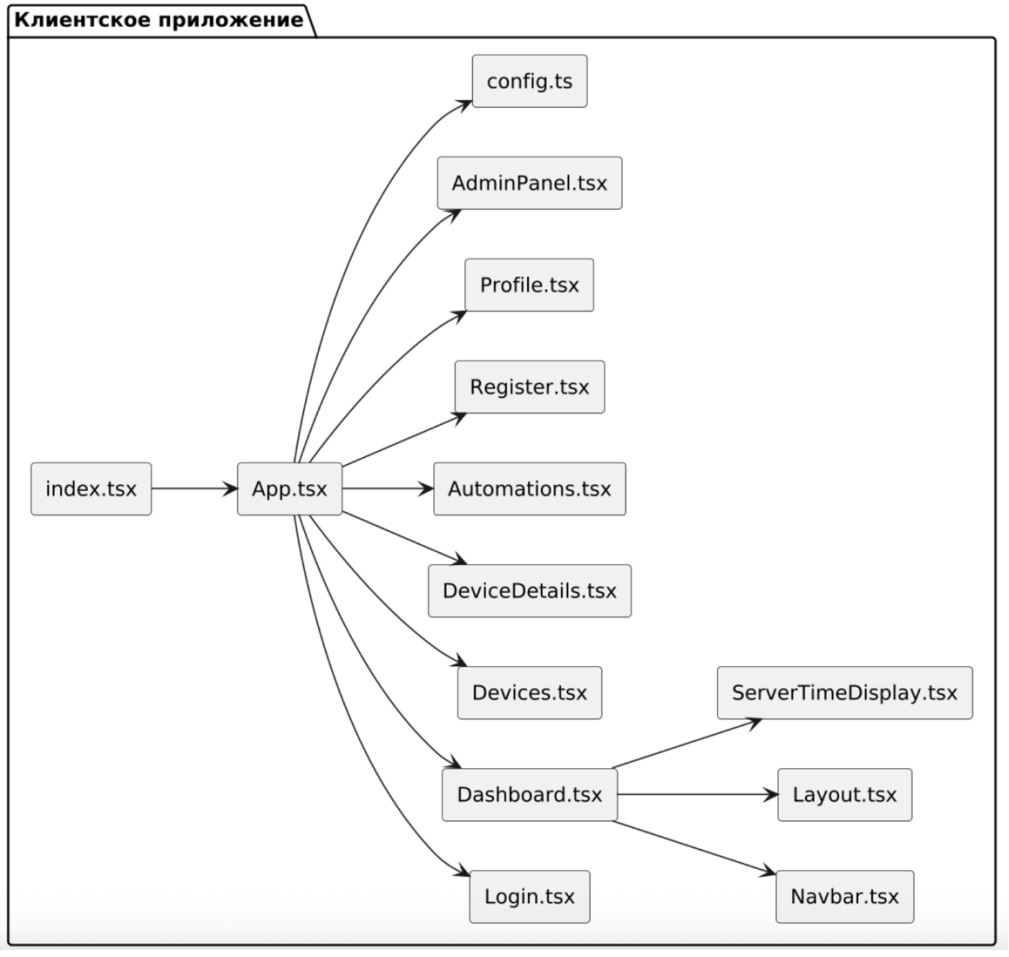

Рисунок 3 – Архитектура клиентского приложения
(схема создана при помощи пакета программного обеспечения PlantUML)

2.4. Организация передачи данных через MQTT

Обмен информацией между серверной частью комплекса и физическими исполнительными устройствами осуществляется посредством протокола MQTT, являющегося лёгким брокер-ориентированным решением для передачи сообщений в условиях ограниченных ресурсов и нестабильных сетевых соединений. Использование данного протокола обусловлено его низкой нагрузкой на канал передачи данных, поддержкой шаблонизации адресов и возможностью двустороннего взаимодействия между компонентами системы в режиме реального времени.

В центре организации передачи данных находится MQTT-брокер, развёрнутый в контейнеризированной среде и подключённый к общей виртуальной сети программно-аппаратного комплекса. Все микроконтроллерные устройства, участвующие в функционировании системы, подписаны на определённые топики брокера, соответствующие их логической принадлежности или физическому идентификатору. Управляющие команды публикуются серверным приложением в топик формата /home/device/<id>/control, где <id> – уникальный идентификатор устройства.

Каждое устройство, входящее в состав системы, реализует клиентскую часть протокола MQTT и подписывается на соответствующий топик, откуда получает команды, касающиеся его состояния, режима работы или параметров функционирования. Помимо приёма сообщений, устройства также публикуют телеметрию, содержащую информацию о текущем состоянии, времени последней активности, значениях сенсоров и других характеристиках. Эти сообщения поступают в топик /home/device/<id>/state, откуда серверное приложение получает их, обрабатывает и сохраняет в базе данных для последующего анализа или отображения в пользовательском интерфейсе.

Передаваемые через MQTT сообщения формируются в формате JSON и включают в себя как идентификаторы устройств и типы действий, так и значения параметров, требующих установки. Такое представление обеспечивает универсальность форматов и облегчает их последующую обработку. Иерархическая структура топиков, использующая сегменты home, device, <id>, отражает логическую адресацию устройства в пределах системы и позволяет избирательно подписывать компоненты на необходимые группы сообщений, исключая избыточный трафик.

Организация подписки и публикации реализована с учётом требований отказоустойчивости и безопасности: в случае временного разрыва соединения клиентские библиотеки обеспечивают автоматическое восстановление связи, а брокер может использовать механизм «last will» (рус. – «завещание», «предсмертное волеизъявление»), позволяющий информировать систему о внезапной потере контакта с устройством. Кроме того, соединение между микроконтроллером и брокером осуществляется с использованием шифрования по протоколу SSL/TLS, что позволяет обеспечить конфиденциальность и целостность передаваемой информации.

Схематическое представление механизма передачи данных по протоколу MQTT приведено на рисунке 4.

Рисунок 4 – Диаграмма алгоритма работы брокера MQTT

2.5. Архитектура системы хранения данных

Система хранения и обработки данных в программно-аппаратном комплексе основана на реляционной модели, реализуемой с использованием системы управления базами данных PostgreSQL. База данных служит централизованным хранилищем всех сущностей, участвующих в функционировании системы, включая пользователей, устройства, журналы событий, правила автоматизации, авторизационные сессии и параметры доступа. Вся логика обращения к данным инкапсулирована в серверном приложении, построенном на основе объектно-реляционного отображения.

Объекты базы данных моделируются в виде типизированных классов с использованием библиотеки SQLAlchemy, что позволяет явно задавать типы, связи и ограничения. Вся структура хранения проектировалась с соблюдением принципов нормализации, что обеспечило логическую целостность данных, возможность изоляции пользовательской информации и поддержку масштабируемости.

Центральной сущностью является таблица users, содержащая сведения об авторизованных пользователях: имя, адрес электронной почты, хэш пароля, роль в системе, статус активности и метка времени создания. С этой таблицей напрямую связаны devices, automations, logs и sessions. Каждому пользователю могут принадлежать несколько устройств, доступ к которым строго разграничивается. Эта связь реализуется через поле owner_id в таблице devices.

Текущая информация о состоянии устройств, а также сведения о последней активности хранятся в таблице devices. Для каждого устройства сохраняется уникальный идентификатор, тип, статус, внутреннее состояние (в виде сериализованного JSON-объекта), дата регистрации и дата последнего взаимодействия. Состояние обновляется при получении телеметрии через

MQTT и синхронизируется с визуальным представлением на стороне клиентского интерфейса.

Логи событий формируются при любом значимом действии пользователя или системы и записываются в таблицу logs. Каждая запись содержит тип события, сообщение, дополнительные параметры (если имеются), идентификатор пользователя, IP-адрес инициатора и временную метку. Эта информация используется для аудита, мониторинга, отладки и анализа безопасности. Дополнительно реализована связь журналов с устройствами, что позволяет установить контекст возникновения события.

Механизм управления доступом к системе реализован через таблицу ip_access, в которой для каждого IP-адреса задаются тип доступа (разрешённый или запрещённый), описание, статус активности, дата создания и последнее обновление. Это обеспечивает фильтрацию входящих соединений, как по безопасности, так и по административным политикам.

Авторизационные процессы обслуживаются через таблицу sessions, в которой хранятся токены активных сессий, их время создания и истечения, IP-адрес и строка user-agent. Такая модель позволяет отслеживать многосессионность, обнаруживать аномалии в поведении и обеспечивать выход из системы по тайм-ауту.

Дополнительную категорию представляют таблицы, связанные с системой автоматизации. Таблица automations содержит пользовательские сценарии, определяющие действия, выполняемые при выполнении заданных условий. Здесь хранятся параметры триггера (тип, время, географические координаты, смещение), тип действия, параметры передачи, а также связь с конкретным пользователем и устройством. Каждое правило сопровождается меткой последнего срабатывания и статусом активности.

Хранение данных сопровождается использованием пула соединений с СУБД, параметры которого конфигурируются через переменные окружения и адаптированы под условия контейнеризации. Такое решение позволяет

обеспечить высокую доступность и отказоустойчивость, а также стабильную производительность при многопользовательской нагрузке.

Структура базы данных, отражающая взаимосвязь между сущностями, представлена на рисунке 5.

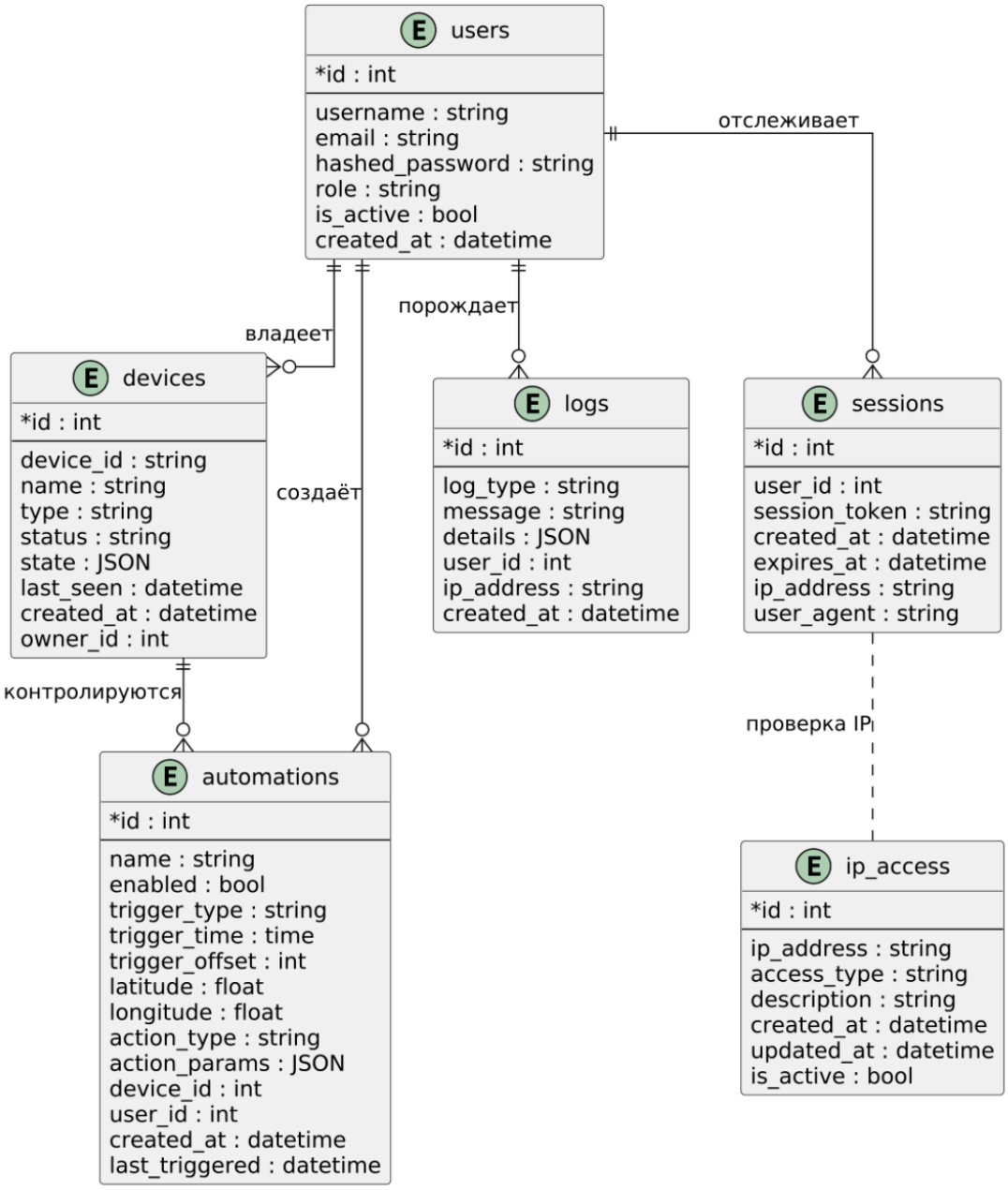

Рисунок 5 – Структура базы данных

(схема создана при помощи пакета программного обеспечения PlantUML)

2.6. Механизм взаимодействия с устройствами «умного дома»

Взаимодействие между серверной частью программно-аппаратного комплекса и исполнительными устройствами, построенными на базе микроконтроллеров ESP32, реализуется по схеме публикации и подписки с использованием брокера сообщений MQTT. Такой подход обеспечивает слабую связанность компонентов, асинхронность обработки команд и высокую масштабируемость архитектуры. Обмен сообщениями происходит в рамках зафиксированных топиков, структура которых отражает иерархию адресации в пределах домашней сети.

Микроконтроллеры подключаются к MQTT-брокеру с применением защищённого протокола передачи данных SSL/TLS, что позволяет предотвратить перехват и подмену управляющих сообщений. Каждое устройство подписывается на уникальный управляющий топик, соответствующий его идентификатору, и реагирует на входящие команды, отправляемые серверным приложением. Состояние устройства публикуется в отдельный телеметрический топик, что позволяет системе отслеживать текущее положение исполнительного механизма, уровень сигнала, внутренние параметры и другую технически значимую информацию.

Формат сообщений, передаваемых в обоих направлениях, представляет собой сериализованные JSON-объекты, содержащие набор ключевых параметров: тип действия, временная метка, идентификатор источника, а также, в случае управляющего сообщения, – требуемое значение. Стандартизация структуры обмена позволяет обрабатывать входящие пакеты без жёсткой привязки к конкретной модели устройства, ограничиваясь логическим сопоставлением параметров.

Прошивка, загружаемая в память ESP32, реализует все функции сетевого взаимодействия, подписки и публикации сообщений, а также

обработку входящих команд и передачу состояния периферийных компонентов. Она построена на базе асинхронной модели, использующей неблокирующую обработку событий, что обеспечивает минимальные задержки отклика и устойчивость к временным сбоям соединения. При этом устройство сохраняет внутреннее состояние и при восстановлении связи синхронизируется с серверной частью посредством публикации текущих данных.

Механизм работы взаимодействия между сервером, брокером сообщений и микроконтроллером отображён на рисунке 6.

Рисунок 6 – Схема взаимодействия между сервером, брокером сообщений и микроконтроллером
(схема создана при помощи пакета программного обеспечения PlantUML)

2.7. Конфигурация средств доставки и развёртывания серверного программного модуля

Развёртывание программно-аппаратного комплекса осуществляется с применением технологии контейнеризации, обеспечивающей воспроизводимую и изолированную среду исполнения. В качестве инструмента оркестрации используется Docker Compose, позволяющий описать структуру и взаимосвязи всех компонентов системы в декларативной форме.

Каждый элемент комплекса – серверное приложение, веб-интерфейс, брокер сообщений, база данных, а также вспомогательные сервисы – развёртывается в виде самостоятельного контейнера. Все сервисы подключены к общей виртуальной сети, благодаря чему обеспечивается взаимодействие между ними без необходимости проброса портов наружу. Уровень сетевой изоляции управляется средствами Docker, что позволяет разграничивать доступ и минимизировать внешние зависимости.

Серверная часть разворачивается как отдельный сервис, содержащий API-приложение на FastAPI. Она запускается в связке с брокером Mosquitto, обслуживающим публикацию и подписку на MQTT-сообщения. Для обеспечения устойчивости к перезапуску брокера применяются тома, в которых хранятся данные о сессиях и конфигурации. База данных PostgreSQL разворачивается в выделенном контейнере с подключённым постоянным томом, обеспечивающим сохранность данных между перезапусками.

Веб-интерфейс, реализованный на React, компилируется в статические ресурсы, обслуживаемые с помощью встроенного HTTP-сервера. Он подключается к серверному приложению через REST-интерфейс по локальной сети, формируемой средствами Compose. Для корректного старта всех компонентов серверная часть указывает зависимости на базу данных и брокер сообщений, используя директиву depends_on.

Docker Compose позволяет централизованно управлять переменными окружения, логированием, сетевыми параметрами и томами хранения. Это облегчает как локальную отладку, так и последующий перенос системы на удалённые хосты или в облачную среду. Дополнительно, контейнеризация способствует повышению надёжности, упрощает автоматическое масштабирование и изоляцию компонентов при необходимости обновления или замены отдельных модулей.

Общая структура развертывания комплекса представлена на рисунке 7.

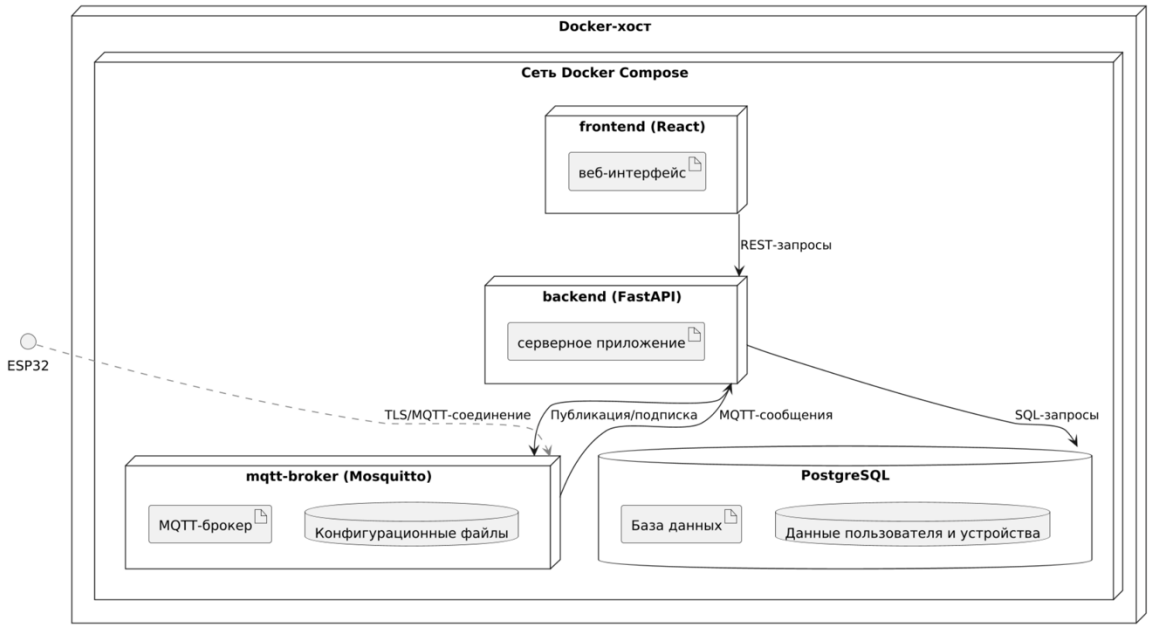

Рисунок 7 – Структура контейнеризованной серверной программной части (схема создана при помощи пакета программного обеспечения PlantUML)

2.8. Выводы по итогам проектирования

В ходе проектирования архитектуры программно-аппаратного комплекса был сформирован целостный образ системы, отражающий как логическую структуру взаимодействия между компонентами, так и физическую реализацию программных и аппаратных модулей. Разработанная архитектура базируется на принципах модульности, масштабируемости и

слабо связанного взаимодействия, что обеспечивает гибкость конфигурации, устойчивость к сбоям и возможность расширения функциональности без нарушения целостности системы.

Ключевыми решениями стали использование брокера сообщений MQTT для организации двустороннего обмена между сервером и микроконтроллерами, реализация REST-интерфейса для связи между клиентской и серверной частями, а также применение контейнеризации и оркестрации через Docker Compose. Каждое из этих решений позволило снизить зависимость компонентов друг от друга, упростить развёртывание и обеспечить воспроизводимость среды.

Сформированная модель хранения данных отражает как текущие состояния устройств и активность пользователей, так и параметры автоматизации и правила доступа. Структура базы данных продумана с учётом будущего масштабирования, что позволяет безболезненно добавлять новые категории сущностей, не нарушая логической целостности схемы.

Проектирование микропрограммного обеспечения позволило обеспечить устойчивое взаимодействие устройств с сервером и гарантированную доставку телеметрии. Благодаря асинхронной модели обработки событий, прошивка остаётся отзывчивой даже в условиях перегрузки или временной недоступности сервера.

Предложенная инфраструктура развёртывания упрощает настройку системы, обеспечивает сетевую связанность компонентов и допускает гибкую адаптацию конфигурации под различные условия эксплуатации. На основании спроектированной архитектуры возможно дальнейшее развитие комплекса, включая расширение набора поддерживаемых устройств, реализацию интеллектуального анализа данных и интеграцию с внешними системами.

3. РАЗРАБОТКА СЕРВЕРНОГО ПРОГРАММНОГО МОДУЛЯ И ПРОГРАММНОГО МОДУЛЯ ДЛЯ ДЕМОНСТРАЦИОННОГО УСТРОЙСТВА «УМНОГО ДОМА»

3.1. Общие принципы реализации практической части

Разработка программных модулей комплекса была организована на основе принципов модульности, слабой связанности компонентов и соблюдения границ ответственности между слоями системы. Основное внимание было уделено устойчивости исполнения, расширяемости архитектуры и упрощению тестирования. Комплекс разделён на две ключевые части: серверный программный модуль, обеспечивающий логическую обработку и координацию компонентов, и микропрограммное обеспечение для демонстрационного устройства, реализующее выполнение управляющих команд и передачу состояния через сеть.

Серверный модуль построен с использованием асинхронного веб-фреймворка FastAPI, что позволило реализовать неблокирующую модель обработки запросов и упростило интеграцию с брокером сообщений. Для хранения и выборки данных используется система управления базами данных PostgreSQL, связанная с сервером через ORM-абстракцию. В рамках единого приложения реализованы интерфейсы аутентификации пользователей, маршруты для управления состоянием устройств, а также обработчики обратной телеметрии, поступающей от микроконтроллеров.

Механизмы межкомпонентного взаимодействия опираются на брокер сообщений MQTT, который позволяет эффективно обрабатывать асинхронные события. Серверный модуль выполняет публикацию управляющих команд и подписку на телеметрию от устройств в соответствующих топиках, что обеспечивает двустороннюю связь без прямого подключения между участниками.

Разработка прошивки для микроконтроллера велась на языке программирования C++ с использованием фреймворка ESP-IDF и библиотеки для асинхронной работы с протоколом MQTT. Программный модуль обеспечивает автоматическое подключение к сети, установление защищённого соединения с брокером, подписку на управляющий канал и передачу состояния в серверную часть. В прошивку встроен минимальный механизм обработки сбоев связи, предусматривающий переподключение и повторную синхронизацию.

Оба программных модуля (ПМ) – серверный и ПМ устройства «умного дома» – были спроектированы с ориентацией на минимальные зависимости и возможность функционирования в изолированной среде. Это позволило организовать процесс тестирования и отладки отдельных компонентов независимо друг от друга и упростить развёртывание в контейнеризованной среде.

3.2. Разработка серверного программного модуля

Изначально предполагалось, что серверная логика комплекса будет размещена непосредственно на микроконтроллере ESP32. Такой подход должен был позволить отказаться от внешнего вычислительного узла, упростив архитектуру за счёт интеграции логики и управления на одном устройстве. Однако в процессе реализации были выявлены критические ограничения, связанные с дефицитом оперативной и постоянной памяти. По этой причине архитектурное решение было пересмотрено: серверный программный модуль был перенесён на вычислительную платформу с архитектурой x86 и операционной системой GNU/Linux (Debian 12 «Bookworm»)

Разработка велась на языке программирования Python версии 3.11. В качестве основной среды разработки применялась PyCharm, обеспечивающая

поддержку интерактивной отладки, типизации, анализа зависимостей и работы с системой контроля версий.

Серверное приложение реализовано с использованием фреймворка FastAPI, ориентированного на создание асинхронных API-приложений. Выбор этой технологии обусловлен необходимостью неблокирующей обработки запросов, простой интеграции с брокером сообщений и возможностью строгой типизации маршрутов. Для взаимодействия с базой данных применяется SQLAlchemy с использованием декларативной модели и асинхронного драйвера.

Внутренняя структура проекта организована по слоям, отражающим архитектурное разделение обязанностей:

– директория routers/ содержит определения маршрутов API и описания входных/выходных моделей,

– в services/ размещена бизнес-логика приложения,

– models/ содержит ORM-отображения таблиц базы данных,

– папка core/ отвечает за конфигурацию, инициализацию брокера и базы данных, а также вспомогательные утилиты.

Для обмена сообщениями между сервером и микроконтроллерами используется MQTT-брокер, на который сервер подписывается на топики состояния (/home/device/<id>/state) и публикует управляющие команды в топики вида /home/device/<id>/control. Подключение к брокеру реализовано с использованием асинхронного клиента MQTT, инициализируемого при запуске приложения. Исходные коды MQTT-клиента и MQTT-сервиса приведены в листингах А.1 и А.2 приложения А. В случае недоступности брокера предпринимается повторное соединение с экспоненциальной задержкой.

Механизм аутентификации построен на основе протокола Bearer Token с применением JWT. Исходный код модуля аутентификации приведён в листинге А.3 приложения А. Пользовательские токены проверяются при

каждом запросе к защищённым маршрутам, а данные аутентификации сохраняются в базе данных в хэшированном виде. Дополнительно реализовано логирование входов, действий пользователей и регистрация IP-адресов, с возможностью их фильтрации на уровне бизнес-логики.

Журналирование событий и ошибок организовано с использованием стандартного модуля logging, с возможностью ротации логов и сохранения отчётов в базу данных. При возникновении исключений система формирует информативные сообщения, содержащие данные о причине сбоя, пользователе и типе действия.

В процессе разработки особое внимание уделялось устойчивости к отказам и предсказуемому поведению при перегрузках. За счёт изоляции сервисов в контейнерах (см. листинг А.4 приложения А) обеспечивается надёжность и возможность перезапуска отдельных компонентов без остановки всей системы.

Таким образом, серверный программный модуль представляет собой полнофункциональное асинхронное приложение с поддержкой авторизации, управления состоянием устройств, двустороннего обмена сообщениями и журналирования активности. Его архитектура обеспечивает удобство сопровождения, расширяемость и совместимость с будущими версиями клиентского и микроконтроллерного ПО.

3.3. Разработка клиентского REST-интерфейса

Клиентский REST-интерфейс является одним из ключевых компонентов серверного модуля, обеспечивая программный доступ к основным функциям системы со стороны как пользовательского веб-интерфейса, так и микроконтроллерных устройств. Он был разработан с нуля на основе фреймворка FastAPI, что позволило реализовать строго типизированные, валидируемые и документированные маршруты в асинхронной модели.

Архитектурно был выбран подход, основанный на REST-парадигме, что обусловлено необходимостью совместимости с браузерами, стандартными HTTP-клиентами, а также упрощённой отладкой и возможностью тестирования. Рассматривались альтернативы в виде WebSocket и gRPC, однако они были отвергнуты из-за избыточности при текущем уровне функциональности. REST-подход оказался более предсказуемым в контексте микросервисной архитектуры, в частности при взаимодействии с фронтендом и простейшими HTTP-клиентами со стороны устройств.

Каждый REST-маршрут реализован в виде асинхронной функции в рамках модулей, размещённых в каталоге backend/routers/. Структуры входных и выходных данных определены в backend/schemas/ с использованием библиотеки Pydantic, что позволило обеспечить статическую проверку типов, а также валидацию данных до передачи в бизнес-логику. Это позволило минимизировать ошибки при обработке запросов и централизовать контроль над форматом данных.

Особое внимание было уделено вопросам безопасности. Доступ к маршрутам, связанным с управлением устройствами, возможен только при наличии действительного JWT-токена. Пользовательские токены генерируются при успешной аутентификации и передаются в заголовке Authorization при каждом последующем запросе. В каждом защищённом маршруте осуществляется проверка прав доступа и соответствие идентификаторов ресурса пользователю, от имени которого выполняется операция. Таким образом исключается возможность несанкционированного доступа к чужим данным, даже при прямом обращении к URL.

Реализация маршрутов управления устройствами (/devices/, /devices/{id}/state, /devices/{id}/control) была сопряжена с необходимостью унификации форматов данных между сервером, клиентским интерфейсом и микроконтроллером. С этой целью в схему был введён JSON-параметр state, обеспечивающий хранение гибкой структуры состояния без жёсткой привязки

к типу устройства. Это решение позволило поддерживать на одном уровне как простые светодиодные устройства, так и потенциально более сложные сенсоры и реле.

На этапе разработки активно применялись инструменты Swagger UI (автоматически предоставляемый FastAPI по маршруту /docs) и Postman, с помощью которых выполнялась ручная отладка всех маршрутов, включая проверку валидации, поведения при ошибках и работы системы авторизации. Это позволило на раннем этапе устранить несогласованности между схемами и бизнес-логикой.

В процессе реализации был выработан подход, при котором вся логика, связанная с REST-интерфейсом, максимально изолирована от взаимодействия с базой данных и брокером сообщений. Взаимодействие с инфраструктурными компонентами осуществляется через слой сервисов, размещённый в backend/services/, что позволило обеспечить переиспользуемость кода и облегчить модульное тестирование.

Таким образом, клиентский REST-интерфейс представляет собой разработанный с нуля, полнофункциональный API, обеспечивающий аутентификацию, управление состоянием устройств, регистрацию пользователей, настройку автоматизаций и журналирование действий. Он стал надёжной точкой входа во внутреннюю логику системы и может быть расширен без модификации основного ядра приложения.

3.4. Разработка веб-интерфейса управления системой

Клиентский интерфейс представляет собой одностраничное веб-приложение, разработанное с использованием библиотеки React. Его основная задача заключается в обеспечении интерактивного взаимодействия пользователя с системой управления компонентами «умного дома», включая выполнение операций входа, управления устройствами, настройки

автоматизаций и просмотра системной информации. Интерфейс реализует асинхронную модель взаимодействия с сервером через REST-API, описанный в предыдущей подглаве.

Проект был создан с использованием официального инструмента инициализации create-react-app, после чего подвергался модификациям и оптимизации под конкретные требования. Основной пользовательский интерфейс формируется на основе компонентной архитектуры, где каждый элемент страницы реализован как изолированный компонент с собственным состоянием и логикой обновления. Для реализации визуальной части интерфейса была выбрана библиотека Chakra UI, предоставляющая готовые компоненты и обеспечивающая адаптивную верстку без необходимости писать CSS вручную.

Идентификация и аутентификация пользователей реализованы через специальную экранную форму входа (см. рисунок 8).

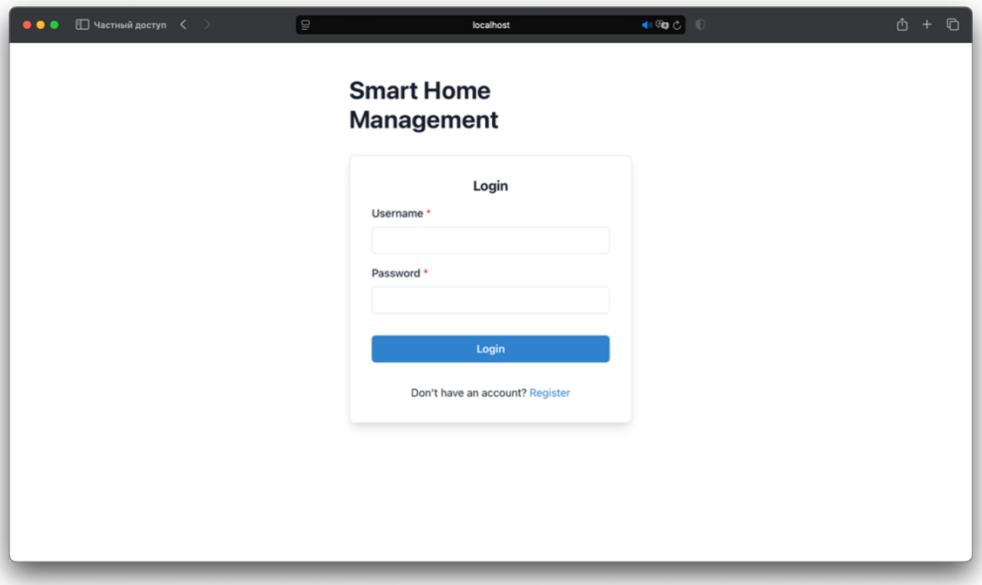

Рисунок 8 – Экранная форма входа в систему управления компонентами «умного дома»

Форма производит проверку учётных данных и получение токена доступа. В случае отсутствия учётной записи пользователь может пройти процедуру регистрации через соответствующую экранную форму (см. рисунок 9), после чего происходит автоматический вход в систему.

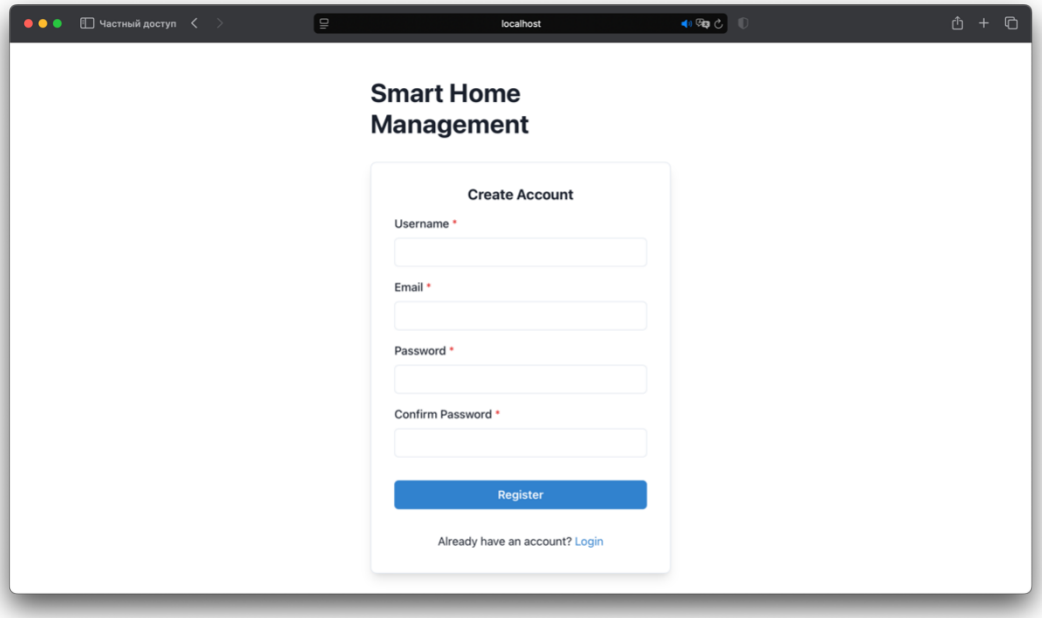

Рисунок 9 – Экранная форма регистрации в сервисе

Все действия, связанные с входом, регистрацией и проверкой токенов, выполняются посредством асинхронных HTTP-запросов, адресованных маршрутам REST-интерфейса.

После успешной аутентификации пользователь получает доступ к главному интерфейсу, отображающему список доступных устройств и их текущее состояние. Управление отдельным устройством осуществляется через специализированную экранную форму (см. рисунок 10), где отображается состояние устройства и доступны элементы управления. Все изменения отправляются в серверную часть и затем транслируются на уровень микроконтроллера через брокер сообщений.

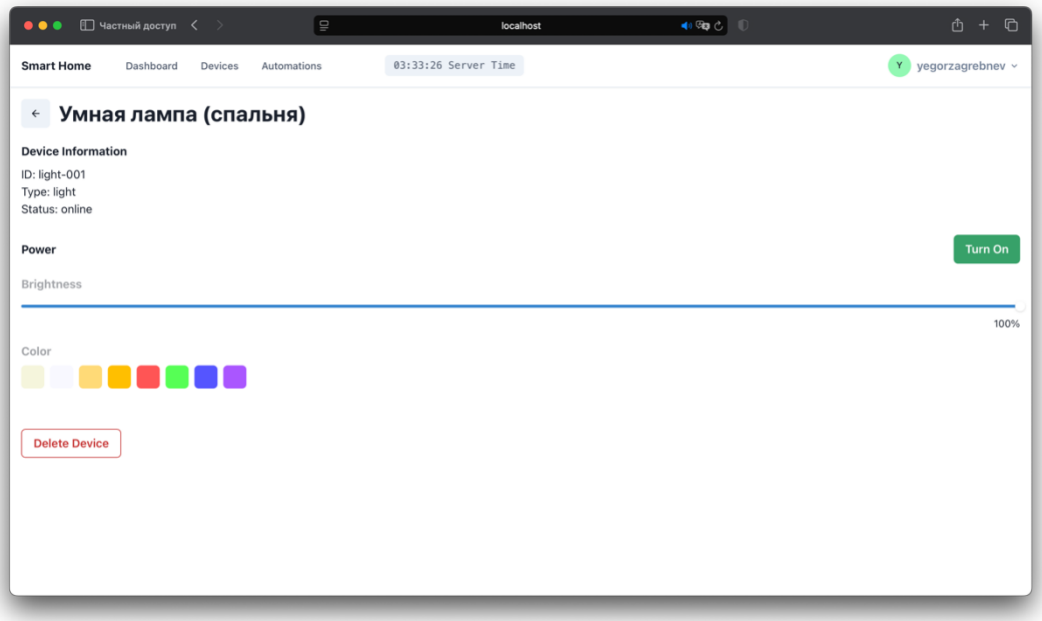

Рисунок 10 – Экранная форма управления устройством «умного дома»

Настройка пользовательских сценариев автоматизации реализована на отдельной странице (см. рисунок 11), где можно задать параметры триггера, действия и связать их с конкретным устройством. Реализация интерфейса построена на многоступенчатой форме, позволяющей пользователю поэтапно задать необходимые параметры, после чего сформированная конфигурация отправляется на сервер и сохраняется в базе данных.

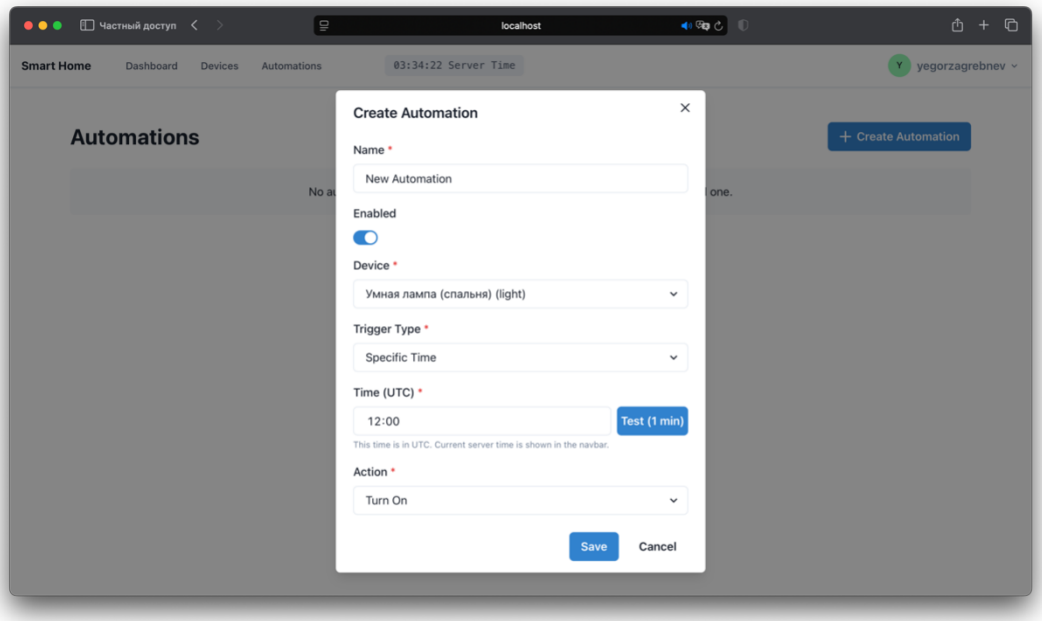

Рисунок 11 – Экранная форма настройки нового сценария автоматизации

Дополнительные функции включают страницу настроек учётной записи (см. рисунок 12), где реализованы смена пароля, редактирование личных данных и выход из системы. Реализация этой части интерфейса построена с учётом требований безопасности и защиты от несанкционированного доступа: все операции требуют подтверждённой сессии и валидного токена.

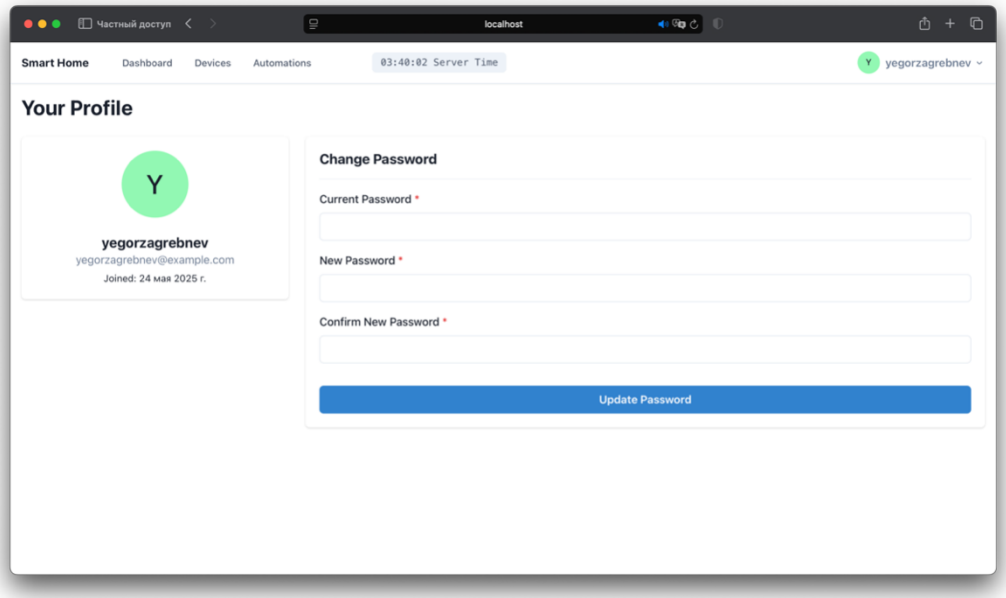

Рисунок 12 – Экранная форма настроек учётной записи

Особое внимание в ходе разработки было уделено адаптивности интерфейса. Интерфейс автоматически перестраивается при уменьшении ширины экрана и обеспечивает корректную работу на мобильных устройствах, планшетах и десктопах. Вёрстка протестирована на ряде устройств, включая смартфоны с диагональю от 5 дюймов. Общий вид мобильной версии интерфейса представлен на рисунке 13.

Рисунок 13 – Экранная форма управления устройством «умного дома», адаптированная для мобильных устройств

Фронтенд-приложение взаимодействует исключительно с REST-интерфейсом, не обращаясь к базе данных или брокеру напрямую. За счёт этого достигается высокая степень изоляции между клиентской и серверной частью, что упрощает сопровождение и масштабирование системы.

3.5. Разработка программного модуля для демонстрационного устройства «умного дома»

В качестве демонстрационного устройства в рамках данной работы был использован микроконтроллер ESP32, оснащённый адресной светодиодной лентой WS2812B. Выбор указанных компонентов обусловлен рядом факторов. Во-первых, ESP32 представляет собой высокоинтегрированное решение с поддержкой Wi-Fi и Bluetooth, что делает его удобным для реализации проектов в области умной электроники. Во-вторых, как микроконтроллер, так и светодиодная лента отличаются доступной стоимостью, что делает их привлекательными при создании прототипов и учебных макетов.

Управление лентой осуществляется посредством одного цифрового вывода, благодаря чему упрощается аппаратная реализация и снижается количество используемых выводов микроконтроллера.

Питание устройства осуществляется от стандартного источника постоянного тока напряжением 5 В, что также повышает универсальность применяемых компонентов. За счёт минимализма конструкции обеспечивается надёжная работа устройства и простота в его повторении другими разработчиками.

Указанная аппаратная часть служит важным связующим звеном между программным обеспечением и физическим исполнением проекта, позволяя в реальном времени наблюдать за откликами системы на поступающие команды, а также за передачей телеметрических данных обратно на сервер.

Принципиальная электрическая схема такого устройства представлена на рисунке 14.

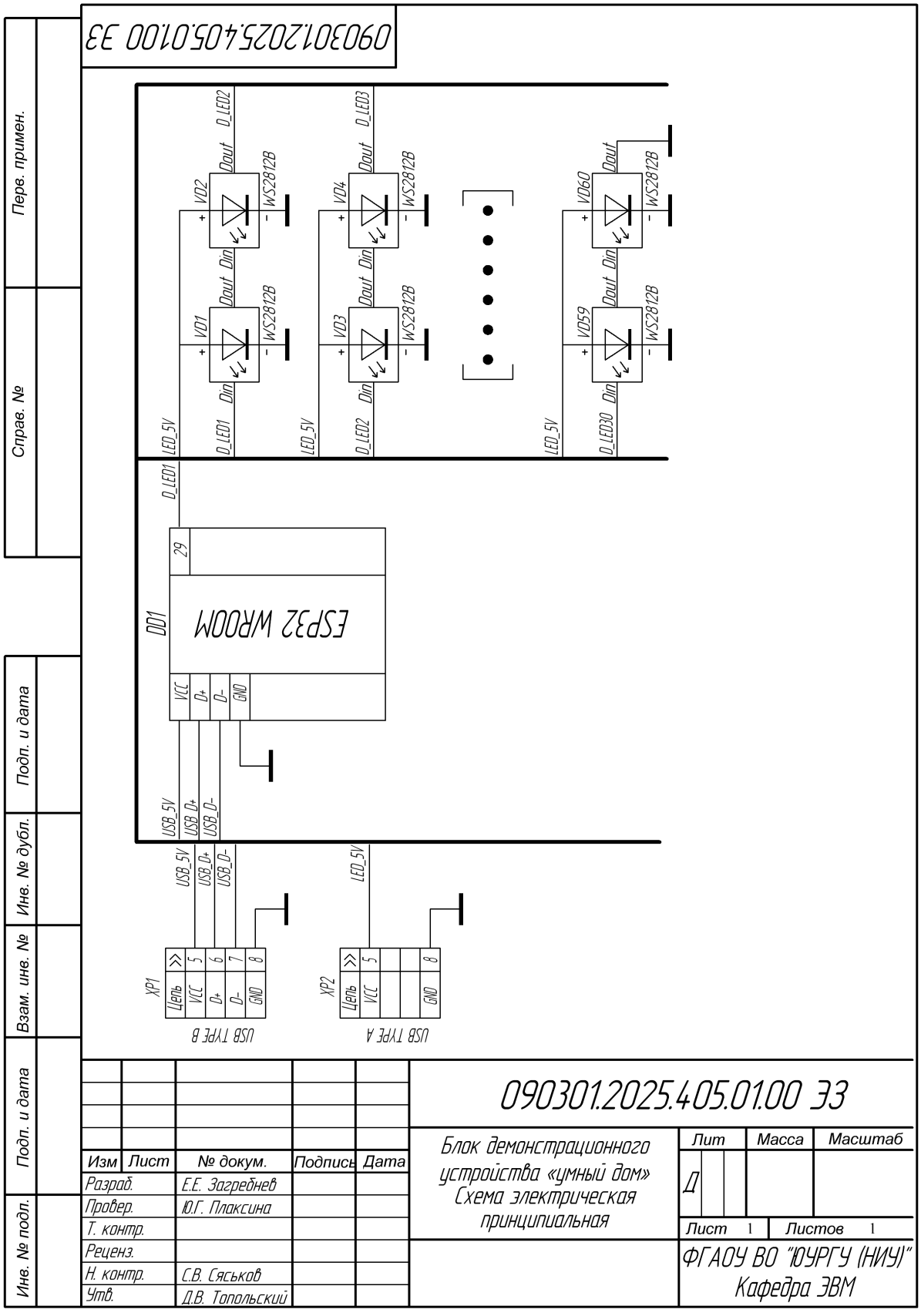

Рисунок 14 – Принципиальная электрическая схема демонстрационного устройства «умного дома»

Разработка прошивки велась на языке C++ с использованием фреймворка ESP-IDF. Исходный код прошивки приведён в листинге В.1 приложения В. Первоначально предпринималась попытка реализовать серверную логику непосредственно на ESP32, включая хранение состояний, маршрутизацию команд и аутентификацию. Однако в процессе работы были выявлены архитектурные и ресурсные ограничения: нестабильность работы TCP-стека, недостаток оперативной памяти и сложность расширения логики. В результате архитектура системы была переработана, и микроконтроллер получил строго определённую роль – подписчика и отправителя данных в рамках MQTT-протокола.

После подключения к Wi-Fi устройство инициирует защищённое соединение с брокером сообщений, используя TLS. Устройство подписывается на управляющий топик вида /home/device/<id>/control и публикует информацию о своём состоянии в топик /home/device/<id>/state. Структура управляющего сообщения представляет собой сериализованный JSON-объект, содержащий параметры, такие как цвет свечения, уровень яркости, режим анимации и длительность действия. Ответным сообщением в топик состояния устройство публикует аналогичный JSON-объект, содержащий подтверждённые параметры, что позволяет пользователю видеть актуальное состояние в интерфейсе.

На уровне прошивки реализована система буферизации и отложенного выполнения команд. Каждое входящее сообщение проверяется на корректность структуры и временную метку. В случае поступления новой команды до завершения текущей, активная команда может быть прервана только при определённых условиях (например, если текущая анимация не является критически важной). Это обеспечивает защиту от неконтролируемого поведения в случае многократных быстрых отправок с клиента. Внутри устройства также реализована блокировка повторной обработки идентичных команд, поступающих с минимальным временным

интервалом (так называемый debounce), что особенно актуально в условиях нестабильной сети.

Поведение устройства в случае потери связи с брокером строго определено. При разрыве соединения активируется таймер повторного подключения с экспоненциальной задержкой, а сам микроконтроллер переходит в режим ожидания. При длительном отсутствии связи (более 60 секунд) устройство переходит в визуальный диагностический режим: на светодиодной ленте воспроизводится предустановленная анимация, информирующая пользователя о потере связи. После успешного восстановления соединения и получения подтверждения от брокера устройство публикует актуальное состояние и возвращается к обычному режиму работы.

В состав прошивки включены модули инициализации сетевого подключения, клиентской MQTT-логики, парсинга JSON-сообщений, управления периферийным устройством и ведения отладочного протокола по UART. Все компоненты сгруппированы в рамках проекта firmware, структура которого представлена в архиве firmware.zip. Компиляция прошивки осуществляется с использованием утилиты idf.py, загрузка на устройство выполняется через стандартный загрузчик ESP32.

3.6. Инфраструктура развёртывания

Инфраструктура всего комплекса развертывается при помощи Docker Compose, что обеспечивает простую и воспроизводимую установку всех сервисов на любой хост с установленным Docker. Основной файл конфигурации – docker-compose.yml, расположенный в корне репозитория проекта, описывает четыре ключевых сервиса: базу данных PostgreSQL, MQTT-брокер Mosquitto, серверное приложение и клиентский интерфейс. В

результате при запуске все компоненты автоматически получают изоляцию, сеть и нужные тома для хранения данных.

В секции services конфигурации сначала описан сервис postgres, который использует образ postgres:15. В переменных окружения задаётся пользователь homeuser, пароль homepassword и название базы homemanagement. Директива volumes указывает на создание именованного тома postgres_data, монтируемого внутрь контейнера для постоянного хранения данных. Порт 5432 на хосте проброшен на такой же порт внутри контейнера, однако в рамках внутренней сети он не используется напрямую приложением, поскольку ORM обращается к postgres по имени сервиса. Сетевая конфигурация задаёт один единственный мостовой драйвер home_network, в котором будут находиться все сервисы комплекса.

Следующим описан mosquitto – контейнер с образом eclipse-mosquitto:2. Здесь настроена папка конфигурации ./docker/mosquitto/config, папка для данных и папка для логов, каждая из которых монтируется в соответствующий каталог внутри контейнера. Порты 1883 и 9001 проброшены для возможного подключения извне, но внутри сети сервисы обращаются к брокеру по имени mosquitto. Именно на него подписывается серверное приложение, публикуя и получая сообщения в пределах топиков /home/device/<id>/{state,control}.

Ключевой элемент развертывания – сервис backend. Он строится из локального каталога ./backend по Dockerfile, где прописаны инструкции по установке всех Python-зависимостей и окружения на основе Python 3.13. Внутри этого контейнера через переменные окружения передаются параметры подключения к базе (DATABASE_URL=postgresql://homeuser:homepassword@postgres:5432/home management) и к брокеру (MQTT_BROKER_HOST=mosquitto, MQTT_BROKER_PORT=1883), а также параметры пула соединений SQLAlchemy для оптимизации производительности. Порт 8000 контейнера проброшен наружу, что позволяет проверять REST-интерфейс вручную.

Директива depends_on гарантирует, что backend не начнёт работу до готовности postgres и mosquitto. Все четыре сервиса подключаются к одной сети home_network, что обеспечивает локальную связность без необходимости проброса лишних портов за пределы Docker.

Для клиентского интерфейса описан сервис frontend, который строится аналогичным образом из каталога ./frontend по своему Dockerfile. После сборки React-приложение разворачивается на встроенном сервере (чаще всего npm start или nginx), и порт 3000 делает его доступным для локальных тестов. Поскольку фронтенд напрямую обращается через REST к сервису по адресу http://backend:8000, директива depends_on: - backend гарантирует, что при старте фронтенда сервис бэкенда уже готов к обработке запросов. Внутри той же сети home_network ни фронтенд, ни бэкенд не пробрасывают свои внутренние порты, за исключением явных записей в ports, которые нужны только для локальной проверки.

Вся конфигурация работоспособна в связке с дополнительным скриптом launch.sh, который упрощает запуск комплекса. Этот скрипт делает попытку собрать образы, запустить контейнеры и следить за их статусом. Его содержание (файл launch.sh) выполняет команду docker-compose up --build -d, после чего проверяет результат и выводит ссылку на адреса фронтенда (http://localhost:3000) и бэкенда (http://localhost:8000/docs). При необходимости он может очищать предыдущие контейнеры командами docker-compose down и docker volume prune, чтобы гарантировать чистую среду.

В результате такой организации развертывания весь комплекс оказывается полностью изолированным в единой виртуальной сети Docker, где каждый сервис доступен по своему имени: postgres, mosquitto, backend и frontend. Тома обеспечивают сохранность данных пользователей и телеметрии между перезапусками. Переход на новую машину или перенос в облако

сводится к копированию репозитория и однократному запуску скрипта launch.sh с установленным Docker Engine.

3.7. Выводы по итогам разработки программных модулей

В рамках настоящей главы были реализованы ключевые программные компоненты комплекса управления элементами «умного дома», включая серверный модуль, прошивку исполнительного устройства и клиентский веб-интерфейс. Каждый из этих компонентов был разработан в соответствии с проектными решениями, изложенными в предыдущей главе, и прошёл практическую проверку в условиях, приближённых к реальной эксплуатации.

Серверный модуль, разработанный с использованием FastAPI и PostgreSQL, реализует полноценную логику аутентификации, маршрутизации запросов, взаимодействия с MQTT-брокером и исполнения сценариев автоматизации. Архитектура кода построена по модульному принципу, что обеспечивает читаемость, сопровождаемость и возможность масштабирования. Использование асинхронных обработчиков позволило добиться высокой производительности при обслуживании параллельных соединений.

Программный модуль для микроконтроллера ESP32 реализован на языке C++ в среде Arduino IDE. Он обеспечивает устойчивое подключение к сети, взаимодействие с брокером сообщений и управление адресной светодиодной лентой. Архитектура прошивки основана на разделении ответственности между модулями, что позволило изолировать задачи управления соединениями, логики обработки команд и отправки состояния. Предусмотрена защита от сбоев связи и автоматическое восстановление соединений.

Клиентский веб-интерфейс разработан с применением React, TypeScript и Chakra UI. Он предоставляет пользователю доступ ко всем функциям

комплекса, включая регистрацию, управление устройствами, настройку автоматизаций и мониторинг состояния системы. Поддержка WebSocket-соединений обеспечивает получение обновлений в реальном времени. Интерфейс адаптивен, безопасен и протестирован на различных типах устройств.

В ходе разработки были решены практические задачи, связанные с обеспечением устойчивости соединений, безопасностью хранения токенов аутентификации, а также получением и анализом метрик производительности. Полученные значения времени отклика подтверждают пригодность комплекса для использования в системах, работающих в режиме, близком к реальному времени.

Реализованные программные модули формируют технологическую основу для построения масштабируемой и надёжной системы управления компонентами «умного дома», а также закладывают базу для последующего развития проекта, включая расширение перечня поддерживаемых устройств и интеграцию с мобильными платформами.

4. ПРОВЕДЕНИЕ ТЕСТИРОВАНИЯ ПРОГРАММНЫХ МОДУЛЕЙ

4.1. Цель и методы тестирования

Целью тестирования являлась верификация корректности реализации основных компонентов программно-аппаратного комплекса, а также подтверждение их соответствия требованиям, сформулированным в техническом задании. Тестирование проводилось по методике функциональной валидации, включающей как модульное, так и интеграционное исследование поведения системы. Особое внимание уделялось устойчивости компонентов к некорректным данным, потере связи, а также оценке временных характеристик при выполнении операций управления.

4.2. Тестирование на предмет соответствия функциональным требованиям

В рамках проверки управления устройствами была протестирована возможность изменения параметров исполнительного устройства через веб-интерфейс. Пользователь, прошедший процедуру аутентификации, имел доступ к управлению устройствами, ассоциированными с его учётной записью. При изменении параметров, таких как цвет свечения или уровень яркости, соответствующие команды формировались и публиковались в MQTT-топик /home/device/<id>/control, а устройство реагировало на них почти мгновенно. Факт выполнения команды подтверждался публикацией состояния в топик /home/device/<id>/state, что соответствовало требованию отображения состояния устройства в реальном времени.

Для проверки взаимодействия с MQTT-брокером были смоделированы ситуации потери соединения, некорректных данных и нестабильной сети.

Устройство на базе ESP32 использовало защищённое соединение по протоколу SSL/TLS и при восстановлении связи автоматически повторно подключалось к брокеру, подтверждая тем самым выполнение требования о восстановлении связи в течение 10 секунд. В логах сервера отображались все случаи поступления сообщений, в том числе не соответствующих шаблону, что позволяло отслеживать устойчивость к ошибкам формата и контролировать повторные подключения.

Веб-интерфейс проверялся на соответствие требованиям к аутентификации, разграничению доступа и корректному отображению пользовательских данных. Пользователь не имел доступа к устройствам, не принадлежащим его учётной записи. Данные хранились в базе PostgreSQL, при этом выполнение операций чтения и записи происходило с минимальной задержкой. Интерфейс, построенный с использованием системы компонентов Chakra UI, корректно отрабатывал все элементы взаимодействия, включая формы, переключатели и динамические компоненты. Особое внимание уделялось сохранению состояния при переходах и обновлении страницы, корректной работе авторизации и адекватному отображению сообщений об ошибке.

4.3. Результаты проведённого тестирования программных модулей

Особое внимание в ходе тестирования уделялось временным характеристикам. Замеры времени отклика системы на управляющее действие показывали значения от 280 до 450 мс в условиях штатной нагрузки (не более 5 команд в секунду), что укладывается в заявленные в техническом задании пределы. Ни в одном из протестированных сценариев не было зафиксировано превышения порога в 600 мс.

Проверялась также безопасность взаимодействия. Серверная часть требовала аутентификации с использованием уникальной пары логин–пароль, а доступ по IP-адресу ограничивался в соответствии с политиками фильтрации. Попытки неавторизованного доступа к REST-методам приводили к корректной генерации ответа с кодом 401 Unauthorized и регистрацией события в журнале.

Дополнительно проводилось тестирование устойчивости комплекса к сбоям. Были проведены отключения брокера MQTT и базы данных на уровне контейнера. Во всех случаях система демонстрировала корректное восстановление после устранения неполадки: сервер автоматически переподключался к базе, микроконтроллер – к брокеру, и при этом сохранялась целостность данных и история состояний.

Подробные результаты тестирования сведены в таблицу 1.

Таблица 1 – Таблица тестовых сценариев и результатов тестирования

Сценарий	**Ожидаемый результат**	**Фактический результат**	**Статус**
Регистрация нового пользователя	Пользователь создаётся, токен возвращается	Успешно	Пройден
Вход с корректными учётными данными	Авторизация, перенаправление на главный экран пользовательского интерфейса	Успешно	Пройден
Управление устройством через веб-интерфейс	Устройство меняет состояние, приходит подтверждение	Успешно	Пройден
Отображение текущего состояния устройства	Интерфейс отражает актуальное состояние устройства	Успешно	Пройден

Продолжение таблицы 1

Сценарий	**Ожидаемый результат**	**Фактический результат**	**Статус**
Подключение устройства к брокеру с SSL/TLS	Устройство подключается, обмен сообщениями происходит	Успешно	Пройден
Потеря связи и автоматическое восстановление	Устройство повторно подключается за ≤10 секунд	Успешно (среднее время восстановления соединения – 7,8 секунд на 10 попыток)	Пройден
Разграничение доступа к устройствам	Пользователь видит только свои устройства	Успешно	Пройден
Попытка доступа без авторизации	Ответ 401, отказ в доступе, запись в журнал	Успешно	Пройден
Некорректный JSON в управляющем сообщении	Устройство не реагирует, система остаётся стабильной	Успешно	Пройден
Запуск комплекса в контейнеризированной среде	Все сервисы стартуют, работают в общей сети	Успешно	Пройден

4.4. Выводы по итогам проведённого тестирования

В ходе тестирования были проверены все основные функциональные блоки разработанного программно-аппаратного комплекса. Особое внимание уделялось корректности взаимодействия между серверной частью, клиентским мобильным приложением, встроенным программным

обеспечением конечного устройства, а также промежуточной инфраструктурой, обеспечивающей сетевое взаимодействие компонентов системы.

Проверка проводилась поэтапно, с акцентом на выявление возможных ошибок в логике обмена данными, устойчивость компонентов при различных сценариях использования, а также соответствие поведения системы заранее сформулированным требованиям, изложенным в техническом задании. Механизмы визуализации текущего состояния исполнительных устройств и приёма управляющих сигналов также показали свою состоятельность в рамках выполненных испытаний.

Результаты тестирования подтвердили надёжную реализацию двусторонней связи между сервером и конечными устройствами, включая корректную передачу телеметрии и управляющих команд. В процессе испытаний система демонстрировала стабильную работу при типовой нагрузке и корректную реакцию на внешние воздействия, такие как временное отсутствие сетевого соединения, повторные запросы или перезапуск отдельных компонентов.

Кроме того, в процессе тестирования не были выявлены критические сбои, способные нарушить функционирование комплекса в целом. Система корректно восстанавливалась после отказов, демонстрируя предусмотренную устойчивость к сбоям. Это позволяет говорить о том, что принятые проектные и архитектурные решения соответствуют заявленным критериям надёжности и обеспечивают необходимый уровень производительности для прототипа.

Таким образом, результаты тестирования подтверждают как соответствие функциональных характеристик системы предъявленным требованиям, так и возможность её дальнейшего масштабирования и развития. Полученные данные свидетельствуют о достаточной зрелости проекта для использования в рамках учебной или опытно-промышленной эксплуатации.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы была разработана и реализована программно-аппаратная система управления компонентами «умного дома», включающая серверное программное обеспечение, веб-интерфейс пользователя и прошивку демонстрационного устройства на базе микроконтроллера ESP32. Архитектура комплекса построена по принципу клиент–серверного взаимодействия с применением брокера сообщений, а все компоненты развернуты в контейнеризованной среде, обеспечивающей воспроизводимость, масштабируемость и изоляцию.

Реализация серверного модуля с использованием языка Python, фреймворка FastAPI, системы управления базами данных PostgreSQL и брокера сообщений MQTT позволила обеспечить надёжную маршрутизацию запросов, безопасную аутентификацию и хранение пользовательских данных. Клиентская часть предоставила пользователю интуитивный графический интерфейс с возможностью управления устройствами, настройки сценариев автоматизации и отслеживания истории состояний. В качестве демонстрационного устройства использовалась ESP32, управляющая адресной светодиодной лентой и обеспечивающая обратную связь о текущем состоянии.

Проект прошёл этап функционального тестирования, результаты которого подтвердили соответствие системы всем заявленным требованиям технического задания. Полученные результаты подтверждают работоспособность системы в условиях реального времени, её устойчивость к ошибкам и возможность масштабирования.

Выполненная работа продемонстрировала практическую реализацию распределённой системы управления компонентами «умного дома» и может быть использована в дальнейшем как основа для более сложных решений в области автоматизации и дистанционного контроля устройств.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Системы умного дома (рынок России) [Электронный ресурс]. – Режим доступа: https://www.tadviser.ru/index.php/Статья:Системы_умного_дома_(рынок_России) – Дата обращения: 15.12.2024.
2. Kumar, M. The Impact of IoT on Smart Home Energy Management / M. Kumar, K. M. Pandey // International Journal of Soft Computing and Engineering (IJSCE). – 2023. – Vol. 13, № 5. – P. 7–15. – Режим доступа: https://www.ijsce.org/wp-content/uploads/papers/v13i5/D364714040924.pdf – Дата обращения: 15.12.2024.
3. ГОСТ Р 56508–2015. Системы автоматизации зданий. Термины и определения [Электронный ресурс]. – Режим доступа: https://docs.cntd.ru/document/1200121688 – Дата обращения: 15.12.2024.
4. Умный дом: особенности, характеристики и преимущества [Электронный ресурс]. – Режим доступа: https://nazarov-gallery.ru/news/umnyy_dom_kharakteristiki_i_preimushchestva/ – Дата обращения: 28.04.2025.
5. Система «Умный дом»: что входит? [Электронный ресурс] // FreeHome ABB. – Режим доступа: https://freehomeabb.ru/info/sistema-umnyy-dom-chto-vkhodit/ – Дата обращения: 28.04.2025.
6. Умный дом: особенности и преимущества [Электронный ресурс]. – Режим доступа: https://freehomeabb.ru/info/umnyy-dom-osobennosti-i-preimushchestva/ – Дата обращения: 28.04.2025.
7. Что такое умный дом? [Электронный ресурс]. – Режим доступа: https://knx24.com/news/base/chto_takoe_umnyy_dom/ – Дата обращения: 28.04.2025.

8. Плюсы умного дома [Электронный ресурс]. – Режим доступа: https://knx24.com/news/base/plyusy_umnogo_doma/ – Дата обращения: 28.04.2025.
9. Харчикова, М. А. Использование технологий искусственного интеллекта при создании системы умного дома [Электронный ресурс] / М. А. Харчикова, С. С. Илюхина // Развитие бизнеса: стратегии, проекты, финансы и коммуникации. – 2023. – С. 814–818. – URL: https://elar.urfu.ru/bitstream/10995/125524/1/978-5-91256-595-3_193.pdf – Дата обращения: 28.04.2025.
10. Искусственный интеллект – следующий шаг для умных домов [Электронный ресурс]. – Режим доступа: https://www.unite.ai/ru/искусственный-интеллект---следующий-шаг-для-умных-домов/ – Дата обращения: 28.04.2025.
11. Тукмачева, Ю. А. Обзор и анализ автоматизированных систем «умный дом», представленных на российском сегменте рынка / Ю. А. Тукмачева // Вестник науки. – 2025. – № 4(12). – С. 45–52. – URL: https://www.xn----8sbempclcwd3bmt.xn--p1ai/article/6320 – Дата обращения: 28.04.2025.
12. Ву, Т. З. Анализ систем автоматизированного управления умным домом / Т. З. Ву // Молодой ученый. – 2011. – № 4(27). – Т. 1. – С. 28–31. – URL: https://moluch.ru/archive/27/2914/ – Дата обращения: 28.04.2025.
13. Система автоматизации дома как новое качество жизни [Электронный ресурс]. – Режим доступа: https://controleng.ru/avtomatizatsiya-zdanij/rubetek/ – Дата обращения: 28.04.2025.
14. ГОСТ Р 71200–2023. Системы киберфизические. Умный дом. Общие положения [Электронный ресурс]. – Режим доступа: https://docs.cntd.ru/document/1304634008 – Дата обращения: 28.04.2025.
15. ГОСТ Р 71199–2023. Системы киберфизические. Умный дом. Термины и определения [Электронный ресурс]. – Режим доступа: https://docs.cntd.ru/document/1304634007 – Дата обращения: 28.04.2025.

16. ГОСТ Р 71866–2024. Системы киберфизические. Умный дом. Общие технические требования к автоматизированным системам управления зданием [Электронный ресурс]. – Режим доступа: https://docs.cntd.ru/document/1310964223 – Дата обращения: 28.04.2025.
17. Simbirsoft. Проектируем умный дом с использованием методов бизнес-анализа [Электронный ресурс]. – Режим доступа: https://www.simbirsoft.com/blog/proektiruem-umnyy-dom-s-ispolzovaniem-metodov-biznes-analiza/ – Дата обращения: 28.04.2025.
18. Проектирование телекоммуникационной составляющей системы «Умный дом» [Электронный ресурс]. – Режим доступа: https://дцо.рф/proektirovanie-telekommunikatsionnoj-sostavlyayushhej-sistemy-umnyj-dom/ – Дата обращения: 28.04.2025.

ПРИЛОЖЕНИЕ А

Листинг А.1 – MQTT-сервис для управления устройствами «умного дома»

```
# ===== СЕРВИС MQTT ДЛЯ УПРАВЛЕНИЯ IOT УСТРОЙСТВАМИ =====
# Этот модуль предоставляет высокоуровневый интерфейс для работы с MQTT устройствами

# Импорт типов для аннотации
from typing import Dict, Any, Optional

# Импорт для работы с датой и временем
from datetime import datetime

# Импорт SQLAlchemy для работы с базой данных
from sqlalchemy.orm import Session

# Импорт моделей нашего приложения
from app.models import Device, DeviceStatus

# Импорт системы логирования
from app.core.logger import log_mqtt_event

# Импорт MQTT клиента
from .client import get_mqtt_client


class MQTTService:
    """
    Сервис для управления MQTT устройствами и обработки их состояний.

    Основная функциональность:
    - Обработка обновлений состояния устройств
    - Отправка команд устройствам
    - Мониторинг подключенности устройств
    - Интеграция с базой данных
    - Логирование MQTT событий
    """

    def __init__(self, db: Session, logger=None):
        """
        Инициализация MQTT сервиса.

        Args:
            db (Session): Сессия базы данных для работы с устройствами
            logger: Логгер для записи событий (опционально)
        """
        # Сохранение сессии БД и логгера
        self.db = db
        self.logger = logger

        # Получение MQTT клиента (singleton)
        self.mqtt_client = get_mqtt_client(logger)

        # Регистрация колбэка для обработки обновлений состояния устройств
        self.mqtt_client.register_device_status_callback(self.handle_device_status)

    def handle_device_status(self, device_id: str, state_data: Dict[str, Any]):
        """
        Обработка обновлений состояния устройства, полученных через MQTT.
```

Продолжение листинга А.1

```
        Args:
            device_id (str): Уникальный идентификатор устройства
            state_data (Dict[str, Any]): Новое состояние устройства в формате JSON

        Returns:
            bool: True при успешном обновлении, False при ошибке

        Процесс обработки:
        1. Поиск устройства в базе данных
        2. Обновление статуса на "онлайн"
        3. Сохранение времени последней активности
        4. Обновление состояния устройства
        5. Логирование события
        """
        try:
            # Поиск устройства в базе данных по device_id
            device = self.db.query(Device).filter(Device.device_id == device_id).first()

            if device:
                # === ОБНОВЛЕНИЕ ДАННЫХ УСТРОЙСТВА ===

                # Установка статуса "онлайн" при получении данных
                device.status = DeviceStatus.ONLINE.value

                # Обновление времени последней активности
                device.last_seen = datetime.utcnow()

                # Сохранение нового состояния устройства
                device.state = state_data

                # Сохранение изменений в базе данных
                self.db.commit()

                # === ЛОГИРОВАНИЕ СОБЫТИЯ ===

                # Запись события в систему логирования
                log_mqtt_event(
                    db=self.db,
                    message=f"Device {device_id} status updated",
                    details=state_data,
                    user_id=device.owner_id
                )

                return True
            else:
                # Устройство не найдено в базе данных
                if self.logger:
                    self.logger.warning(f"Received state update for unknown device:
{device_id}")
                return False
        except Exception as e:
            # Обработка ошибок при обновлении состояния
            if self.logger:
                self.logger.error(f"Error updating device status: {str(e)}")
            return False
```

Продолжение листинга А.1

```
    def send_device_command(self, device_id: str, command: Dict[str, Any]) -> bool:
        """
        Отправка команды устройству через MQTT.

        Args:
            device_id (str): Уникальный идентификатор устройства
            command (Dict[str, Any]): Команда для отправки в формате JSON

        Returns:
            bool: True при успешной отправке, False при ошибке

        Поддерживаемые команды:
        - Включение/выключение устройства
        - Изменение яркости
        - Установка цвета
        - Любые другие команды, поддерживаемые устройством

        Процесс отправки:
        1. Проверка существования устройства в БД
        2. Определение типа действия
        3. Отправка команды через MQTT клиент
        4. Логирование команды
        """
        try:
            # === ВАЛИДАЦИЯ УСТРОЙСТВА ===

            # Проверка существования устройства в базе данных
            device = self.db.query(Device).filter(Device.device_id == device_id).first()
            if not device:
                if self.logger:
                    self.logger.warning(f"Attempting to send command to unknown device:
{device_id}")
                return False

            # === ОПРЕДЕЛЕНИЕ ТИПА ДЕЙСТВИЯ ===

            # Определение действия из команды (по умолчанию "control")
            action = "control"
            if "action" in command:
                action = command["action"]

            # === ОТПРАВКА КОМАНДЫ ===

            # Отправка команды через MQTT клиент
            result = self.mqtt_client.publish(device_id, action, command)

            # === ЛОГИРОВАНИЕ КОМАНДЫ ===

            # Запись отправленной команды в лог
            log_mqtt_event(
                db=self.db,
                message=f"Command sent to device {device_id}",
                details=command,
                user_id=device.owner_id
            )

            return result
```

Продолжение листинга А.1

```
        except Exception as e:
            # Обработка ошибок при отправке команды
            if self.logger:
                self.logger.error(f"Error sending device command: {str(e)}")
            return False

    def update_device_status(self):
        """
        Периодическая проверка статуса устройств и маркировка оффлайн устройств.

        Логика проверки:
        - Получение всех онлайн устройств из БД
        - Проверка времени последней активности каждого устройства
        - Маркировка как "оффлайн" устройств, неактивных более 30 секунд
        - Логирование изменений статуса

        Вызывается планировщиком для поддержания актуального статуса устройств.
        """
        try:
            # === ПОЛУЧЕНИЕ АКТИВНЫХ УСТРОЙСТВ ===

            # Получение всех устройств со статусом "онлайн"
            devices = self.db.query(Device).filter(Device.status ==
DeviceStatus.ONLINE.value).all()

            # Получение текущего времени для сравнения
            now = datetime.utcnow()

            # === ПРОВЕРКА КАЖДОГО УСТРОЙСТВА ===

            # Проверка времени последней активности каждого устройства
            for device in devices:
                if device.last_seen:
                    # Расчет времени с последней активности в секундах
                    time_diff = (now - device.last_seen).total_seconds()

                    # Если устройство неактивно более 30 секунд
                    if time_diff > 30:
                        # === МАРКИРОВКА КАК ОФФЛАЙН ===

                        # Изменение статуса на "оффлайн"
                        device.status = DeviceStatus.OFFLINE.value

                        # Сохранение изменений в БД
                        self.db.commit()

                        # Логирование изменения статуса
                        log_mqtt_event(
                            db=self.db,
                            message=f"Device {device.device_id} marked as offline",
                            user_id=device.owner_id
                        )
        except Exception as e:
            # Обработка ошибок при обновлении статусов
            if self.logger:
                self.logger.error(f"Error updating device status: {str(e)}")
```

Окончание листинга А.1

```
def get_mqtt_service(db: Session, logger=None) -> MQTTService:
    """
    Фабричная функция для создания экземпляра MQTT сервиса.

    Args:
        db (Session): Сессия базы данных
        logger: Логгер для записи событий (опционально)

    Returns:
        MQTTService: Экземпляр MQTT сервиса

    Использование:
        mqtt_service = get_mqtt_service(db, logger)
        mqtt_service.send_device_command("device_1", {"action": "on"})
    """
    return MQTTService(db, logger)
```

Листинг А.2 – MQTT-клиент для связи с устройствами «умного дома»

```
# ===== MQTT КЛИЕНТ ДЛЯ СВЯЗИ С IOT УСТРОЙСТВАМИ =====
# Этот модуль обеспечивает низкоуровневую связь с MQTT брокером и устройствами

# Импорт MQTT клиента от Eclipse Paho
import paho.mqtt.client as mqtt

# Импорт стандартных библиотек Python
import json         # Для работы с JSON данными
import os           # Для работы с переменными окружения
import time         # Для задержек и временных операций
import threading    # Для многопоточности
import ssl          # Для SSL/TLS шифрования

# Импорт типов для аннотации
from typing import Dict, Any, Callable, Optional

# Импорт для работы с переменными окружения
from dotenv import load_dotenv

# Загрузка переменных окружения из файла .env
load_dotenv()

# === КОНФИГУРАЦИЯ MQTT ПОДКЛЮЧЕНИЯ ===

# Адрес MQTT брокера (по умолчанию localhost)
MQTT_BROKER_HOST = os.getenv("MQTT_BROKER_HOST", "localhost")

# Порт MQTT брокера (стандартный порт 1883)
MQTT_BROKER_PORT = int(os.getenv("MQTT_BROKER_PORT", "1883"))

# Уникальный ID клиента для подключения к брокеру
MQTT_CLIENT_ID = os.getenv("MQTT_CLIENT_ID", "home_backend")

# Префикс топиков для устройств умного дома
MQTT_TOPIC_PREFIX = os.getenv("MQTT_TOPIC_PREFIX", "/home/device")
```

Продолжение листинга А.2

```
# Использование TLS шифрования (по умолчанию отключено)
MQTT_USE_TLS = os.getenv("MQTT_USE_TLS", "false").lower() == "true"


class MQTTClient:
    """
    MQTT клиент для связи с устройствами умного дома.

    Основная функциональность:
    - Подключение к MQTT брокеру с автопереподключением
    - Подписка на топики состояния устройств
    - Публикация команд управления устройствами
    - Обработка входящих сообщений от устройств
    - Поддержка TLS шифрования
    - Многопоточная обработка сообщений

    Топики MQTT:
    - /home/device/{device_id}/state - состояние устройства
    - /home/device/{device_id}/control - команды управления
    """

    def __init__(self, client_id=None, logger=None):
        """
        Инициализация MQTT клиента.

        Args:
            client_id (str, optional): Уникальный ID клиента
            logger: Логгер для записи событий
        """
        # === ОСНОВНЫЕ ПАРАМЕТРЫ ===

        # ID клиента (используется для идентификации на брокере)
        self.client_id = client_id or MQTT_CLIENT_ID

        # Создание экземпляра MQTT клиента
        self.client = mqtt.Client(client_id=self.client_id)

        # Флаг состояния подключения
        self.connected = False

        # Логгер для записи событий
        self.logger = logger

        # Словарь обработчиков для различных топиков
        self.topic_handlers = {}

        # Колбэк для обработки обновлений состояния устройств
        self.device_status_callback = None

        # === НАСТРОЙКА КОЛБЭКОВ ===

        # Регистрация обработчиков событий MQTT
        self.client.on_connect = self._on_connect       # При подключении
        self.client.on_disconnect = self._on_disconnect # При отключении
        self.client.on_message = self._on_message       # При получении сообщения

        # === НАСТРОЙКА TLS ШИФРОВАНИЯ ===
```

Продолжение листинга А.2

```
        # Включение TLS если требуется
        if MQTT_USE_TLS:
            self.client.tls_set(ca_certs=None, certfile=None, keyfile=None,
                                cert_reqs=ssl.CERT_REQUIRED,
                                tls_version=ssl.PROTOCOL_TLS,
                                ciphers=None)

        # === ИНИЦИАЛИЗАЦИЯ ПОТОКА ПЕРЕПОДКЛЮЧЕНИЯ ===

        # Создание фонового потока для поддержания соединения
        self.reconnect_thread = threading.Thread(target=self._reconnect_loop)
        self.reconnect_thread.daemon = True  # Поток завершится при завершении программы

        # Флаг работы клиента
        self.running = True

    def start(self):
        """
        Запуск MQTT клиента и потока подключения.

        Returns:
            MQTTClient: Возвращает self для цепочки вызовов
        """
        # Запуск фонового потока переподключения
        self.reconnect_thread.start()
        return self

    def stop(self):
        """
        Остановка MQTT клиента и закрытие соединения.
        """
        # Остановка цикла переподключения
        self.running = False

        # Отключение от брокера
        self.client.disconnect()

    def _log(self, message, level="info"):
        """
        Запись сообщения в лог с указанным уровнем.

        Args:
            message (str): Сообщение для записи
            level (str): Уровень лога ("info", "error", "warning")
        """
        if self.logger:
            if level == "info":
                self.logger.info(message)
            elif level == "error":
                self.logger.error(message)
            elif level == "warning":
                self.logger.warning(message)

    def _reconnect_loop(self):
        """
        Бесконечный цикл переподключения к MQTT брокеру.
```

Продолжение листинга А.2

```
        Логика работы:
        - Проверка состояния подключения каждую секунду
        - При отсутствии подключения - попытка переподключения
        - При ошибке подключения - ожидание 5 секунд перед повтором
        - Запуск цикла обработки сообщений после подключения
        """
        while self.running:
            if not self.connected:
                try:
                    # Логирование попытки подключения
                    self._log(f"Connecting to MQTT broker at
{MQTT_BROKER_HOST}:{MQTT_BROKER_PORT}")

                    # Попытка подключения к брокеру
                    self.client.connect(MQTT_BROKER_HOST, MQTT_BROKER_PORT, 60)

                    # Запуск цикла обработки сообщений в отдельном потоке
                    self.client.loop_start()
                except Exception as e:
                    # Логирование ошибки подключения
                    self._log(f"Failed to connect to MQTT broker: {str(e)}", "error")

                    # Ожидание перед следующей попыткой
                    time.sleep(5)

            # Проверка состояния каждую секунду
            time.sleep(1)

    def _on_connect(self, client, userdata, flags, rc):
        """
        Колбэк вызывается при подключении к MQTT брокеру.

        Args:
            client: Экземпляр MQTT клиента
            userdata: Пользовательские данные
            flags: Флаги подключения
            rc (int): Код результата подключения (0 = успех)
        """
        if rc == 0:
            # === УСПЕШНОЕ ПОДКЛЮЧЕНИЕ ===

            # Установка флага подключения
            self.connected = True
            self._log("Connected to MQTT broker")

            # Подписка на все топики состояния устройств
            state_topic = f"{MQTT_TOPIC_PREFIX}/+/state"
            self.client.subscribe(state_topic)
            self._log(f"Subscribed to topic: {state_topic}")
        else:
            # === ОШИБКА ПОДКЛЮЧЕНИЯ ===

            self._log(f"Failed to connect to MQTT broker with result code {rc}", "error")

    def _on_disconnect(self, client, userdata, rc):
        """
```

Продолжение листинга А.2

```
        Колбэк вызывается при отключении от MQTT брокера.

        Args:
            client: Экземпляр MQTT клиента
            userdata: Пользовательские данные
            rc (int): Код причины отключения (0 = плановое отключение)
        """
        # Сброс флага подключения
        self.connected = False

        if rc != 0:
            # Неожиданное отключение
            self._log(f"Unexpected disconnection from MQTT broker: {rc}", "warning")
        else:
            # Плановое отключение
            self._log("Disconnected from MQTT broker")

    def _on_message(self, client, userdata, msg):
        """
        Колбэк для обработки входящих MQTT сообщений.

        Args:
            client: Экземпляр MQTT клиента
            userdata: Пользовательские данные
            msg: Объект сообщения с топиком и данными

        Обрабатываемые типы сообщений:
        - Обновления состояния устройств (топики */state)
        - Пользовательские обработчики для специфических топиков
        """
        try:
            # Извлечение топика и данных из сообщения
            topic = msg.topic
            payload = msg.payload.decode("utf-8")

            # Логирование полученного сообщения
            self._log(f"Received message on topic {topic}: {payload}")

            # === ОБРАБОТКА ОБНОВЛЕНИЙ СОСТОЯНИЯ УСТРОЙСТВ ===

            # Проверка, является ли это сообщением о состоянии устройства
            if topic.endswith("/state"):
                try:
                    # Извлечение ID устройства из топика
                    # Формат топика: /home/device/{device_id}/state
                    device_id = topic.split("/")[-2]

                    # Парсинг JSON данных состояния
                    state_data = json.loads(payload)

                    # Вызов колбэка для обработки состояния устройства
                    if self.device_status_callback:
                        self.device_status_callback(device_id, state_data)
                except json.JSONDecodeError:
                    # Ошибка парсинга JSON
                    self._log(f"Invalid JSON received on topic {topic}: {payload}",
"error")
```

Продолжение листинга А.2

```
                except Exception as e:
                    # Другие ошибки обработки состояния
                    self._log(f"Error processing state message: {str(e)}", "error")

            # === ВЫЗОВ ПОЛЬЗОВАТЕЛЬСКИХ ОБРАБОТЧИКОВ ===

            # Проверка наличия зарегистрированных обработчиков для топика
            if topic in self.topic_handlers:
                try:
                    # Вызов всех обработчиков для данного топика
                    for handler in self.topic_handlers[topic]:
                        handler(topic, payload)
                except Exception as e:
                    # Ошибка в пользовательском обработчике
                    self._log(f"Error in topic handler for {topic}: {str(e)}", "error")
        except Exception as e:
            # Общая ошибка обработки сообщения
            self._log(f"Error processing MQTT message: {str(e)}", "error")

    def publish(self, device_id: str, action: str, data: Dict[str, Any]):
        """
        Публикация сообщения (команды) для устройства.

        Args:
            device_id (str): Уникальный идентификатор устройства
            action (str): Тип действия ("control", "config", etc.)
            data (Dict[str, Any]): Данные команды в формате JSON

        Returns:
            bool: True при успешной публикации, False при ошибке

        Формат топика: /home/device/{device_id}/control
        """
        # Проверка наличия подключения к брокеру
        if not self.connected:
            self._log("Cannot publish message: not connected to MQTT broker", "warning")
            return False

        # === ФОРМИРОВАНИЕ СООБЩЕНИЯ ===

        # Формирование топика для команды устройству
        topic = f"{MQTT_TOPIC_PREFIX}/{device_id}/control"

        # Сериализация данных в JSON
        message = json.dumps(data)

        # === ПУБЛИКАЦИЯ СООБЩЕНИЯ ===

        # Отправка сообщения через MQTT
        result = self.client.publish(topic, message)

        # Проверка результата публикации
        if result.rc == mqtt.MQTT_ERR_SUCCESS:
            # Успешная публикация
            self._log(f"Published message to {topic}: {message}")
            return True
        else:
```

Продолжение листинга А.2

```
            # Ошибка публикации
            self._log(f"Failed to publish message to {topic}: {message}", "error")
            return False

    def register_device_status_callback(self, callback: Callable[[str, Dict[str, Any]], None]):
        """
        Регистрация колбэка для обработки обновлений состояния устройств.

        Args:
            callback: Функция для вызова при получении обновления состояния
                      Принимает параметры: (device_id: str, state_data: Dict[str, Any])
        """
        self.device_status_callback = callback

    def register_topic_handler(self, topic: str, handler: Callable[[str, str], None]):
        """
        Регистрация обработчика для конкретного MQTT топика.

        Args:
            topic (str): MQTT топик для обработки
            handler: Функция-обработчик
                     Принимает параметры: (topic: str, message: str)
        """
        # Добавление обработчика в словарь
        if topic not in self.topic_handlers:
            self.topic_handlers[topic] = []
        self.topic_handlers[topic].append(handler)

        # Подписка на топик если уже подключены к брокеру
        if self.connected:
            self.client.subscribe(topic)
            self._log(f"Subscribed to topic: {topic}")


# === SINGLETON ЭКЗЕМПЛЯР MQTT КЛИЕНТА ===

# Глобальная переменная для хранения единственного экземпляра клиента
mqtt_client = None

def get_mqtt_client(logger=None):
    """
    Получение или создание singleton экземпляра MQTT клиента.

    Args:
        logger: Логгер для записи событий

    Returns:
        MQTTClient: Единственный экземпляр MQTT клиента

    Паттерн Singleton обеспечивает:
    - Единое подключение к MQTT брокеру во всем приложении
    - Экономию ресурсов и соединений
    - Централизованное управление MQTT коммуникацией
    """
    global mqtt_client
    if mqtt_client is None:
```

Окончание листинга А.2

```
        # Создание и запуск нового экземпляра клиента
        mqtt_client = MQTTClient(logger=logger).start()
    return mqtt_client
```

Листинг А.3 – модуль аутентификации и управления пользователями

```
# ===== СЕРВИС MQTT ДЛЯ УПРАВЛЕНИЯ IOT УСТРОЙСТВАМИ # ===== API МАРШРУТЫ АУТЕНТИФИКАЦИИ И
УПРАВЛЕНИЯ ПОЛЬЗОВАТЕЛЯМИ =====
# Этот модуль содержит все эндпоинты для регистрации, входа, управления сессиями и профилем

# Импорт основных компонентов FastAPI
from fastapi import APIRouter, Depends, HTTPException, status, Request
from fastapi.security import OAuth2PasswordRequestForm  # Форма для получения данных входа

# Импорт SQLAlchemy для работы с базой данных
from sqlalchemy.orm import Session

# Импорт модулей для работы с датой и временем
from datetime import datetime, timedelta

# Импорт UUID для генерации уникальных токенов сессий
import uuid

# Импорт типов для аннотации
from typing import List, Optional

# Импорт компонентов нашего приложения
from app.db.database import get_db                                          # Функция получения
сессии БД
from app.models import User, Session as UserSession, UserRole        # Модели пользователя
и сессии
from app.core.security import verify_password, get_password_hash, create_access_token  #
Функции безопасности
from app.core.deps import get_current_active_user, check_ip_access     # Зависимости для
проверок
from app.core.logger import log_auth_event                           # Функция логирования
событий аутентификации
from app.schemas.user import UserCreate, UserResponse, SessionResponse, PasswordChange,
UserDelete  # Схемы данных

# Создание маршрутизатора для группировки эндпоинтов аутентификации
router = APIRouter()


@router.post("/token", response_model=dict)
async def login_for_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
    db: Session = Depends(get_db),
    request: Request = None,
    ip_access: bool = Depends(check_ip_access)
):
    """
    Эндпоинт для входа в систему и получения access токена.

    Args:
        form_data: Данные формы с username и password
```

Продолжение листинга А.3

```
        db: Сессия базы данных
        request: HTTP запрос для получения IP и User-Agent
        ip_access: Результат проверки IP доступа

    Returns:
        dict: Словарь с access_token, типом токена и ID сессии

    Raises:
        HTTPException: 403 при блокировке по IP, 401 при неверных данных, 400 для
неактивного пользователя

    Процесс входа:
    1. Проверка IP доступа
    2. Поиск пользователя по username
    3. Проверка пароля
    4. Проверка активности пользователя
    5. Создание access токена
    6. Создание записи сессии
    7. Логирование события
    """
    # Проверка разрешения доступа с данного IP адреса
    if not ip_access:
        log_auth_event(
            db=db,
            message=f"Login attempt blocked due to IP restrictions: {request.client.host}",
            ip_address=request.client.host
        )
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Access from this IP address is not allowed"
        )

    # Аутентификация пользователя - поиск по имени пользователя
    user = db.query(User).filter(User.username == form_data.username).first()

    # Проверка существования пользователя и корректности пароля
    if not user or not verify_password(form_data.password, user.hashed_password):
        log_auth_event(
            db=db,
            message=f"Failed login attempt for user: {form_data.username}",
            ip_address=request.client.host if request else None
        )
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect username or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    # Проверка активности пользователя (не заблокирован ли аккаунт)
    if not user.is_active:
        log_auth_event(
            db=db,
            message=f"Login attempt for inactive user: {form_data.username}",
            user_id=user.id,
            ip_address=request.client.host if request else None
        )
        raise HTTPException(
```

Продолжение листинга А.3

```
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Inactive user"
        )

    # Создание access токена с информацией о пользователе
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={
            "sub": user.username,    # Subject - имя пользователя
            "role": user.role,       # Роль пользователя для авторизации
            "user_id": user.id       # ID пользователя для быстрого доступа
        },
        expires_delta=access_token_expires
    )

    # Создание новой сессии пользователя для отслеживания активности
    session_token = str(uuid.uuid4())  # Уникальный токен сессии
    expires_at = datetime.utcnow() + access_token_expires

    new_session = UserSession(
        user_id=user.id,
        session_token=session_token,
        expires_at=expires_at,
        ip_address=request.client.host  if  request  else  None,          #  IP  для  аудита
безопасности
        user_agent=request.headers.get("user-agent") if request else None   # User-Agent
для идентификации устройства
    )

    # Сохранение сессии в базе данных
    db.add(new_session)
    db.commit()

    # Логирование успешного входа
    log_auth_event(
        db=db,
        message=f"User logged in: {form_data.username}",
        user_id=user.id,
        ip_address=request.client.host if request else None
    )

    return {
        "access_token": access_token,
        "token_type": "bearer",
        "session_id": new_session.id
    }


@router.post("/register", response_model=UserResponse)
async def register_user(
    user_data: UserCreate,
    db: Session = Depends(get_db),
    request: Request = None,
    ip_access: bool = Depends(check_ip_access)
):
    """
    Эндпоинт для регистрации нового пользователя.
```

Продолжение листинга А.3

```
    Args:
        user_data: Данные нового пользователя (username, email, password)
        db: Сессия базы данных
        request: HTTP запрос для получения IP
        ip_access: Результат проверки IP доступа

    Returns:
        UserResponse: Информация о созданном пользователе

    Raises:
        HTTPException: 403 при блокировке по IP, 400 если пользователь уже существует

    Особенности:
    - Первый зарегистрированный пользователь автоматически получает роль администратора
    - Пароли хешируются перед сохранением в БД
    - Проверяется уникальность username и email
    """
    # Проверка разрешения доступа с данного IP адреса
    if not ip_access:
        log_auth_event(
            db=db,
            message=f"Registration    attempt    blocked    due    to    IP    restrictions:
{request.client.host}",
            ip_address=request.client.host if request else None
        )
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Access from this IP address is not allowed"
        )

    # Проверка существования пользователя с таким же username или email
    existing_user = db.query(User).filter(
        (User.username == user_data.username) | (User.email == user_data.email)
    ).first()

    if existing_user:
        # Определение, какое поле дублируется
        field = "username" if existing_user.username == user_data.username else "email"
        log_auth_event(
            db=db,
            message=f"Registration failed: {field} already exists",
            ip_address=request.client.host if request else None
        )
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail=f"User with this {field} already exists"
        )

    # Хеширование пароля для безопасного хранения
    hashed_password = get_password_hash(user_data.password)

    # Определение роли: первый пользователь становится администратором
    user_count = db.query(User).count()
    role = UserRole.ADMIN.value if user_count == 0 else UserRole.USER.value

    # Создание нового пользователя
```

Продолжение листинга А.3

```
    db_user = User(
        username=user_data.username,
        email=user_data.email,
        hashed_password=hashed_password,
        role=role
    )

    # Сохранение пользователя в базе данных
    db.add(db_user)
    db.commit()
    db.refresh(db_user)  # Получение актуальных данных с ID

    # Логирование события регистрации
    log_auth_event(
        db=db,
        message=f"User registered: {user_data.username} (role: {role})",
        user_id=db_user.id,
        ip_address=request.client.host if request else None
    )

    return db_user


@router.get("/sessions", response_model=List[SessionResponse])
async def get_user_sessions(
    current_user: User = Depends(get_current_active_user),
    db: Session = Depends(get_db)
):
    """
    Получение списка всех активных сессий текущего пользователя.

    Args:
        current_user: Текущий аутентифицированный пользователь
        db: Сессия базы данных

    Returns:
        List[SessionResponse]: Список сессий с информацией о времени создания, IP и
устройстве

    Полезно для:
    - Мониторинга активных входов
    - Выявления подозрительной активности
    - Управления множественными сессиями
    """
    # Получение всех сессий текущего пользователя
    sessions = db.query(UserSession).filter(
        UserSession.user_id == current_user.id
    ).all()

    return sessions


@router.delete("/sessions/{session_id}")
async def end_session(
    session_id: int,
    password: str,
    current_user: User = Depends(get_current_active_user),
```

Продолжение листинга А.3

```
    db: Session = Depends(get_db),
    request: Request = None
):
    """
    Завершение конкретной сессии пользователя.

    Args:
        session_id: ID сессии для завершения
        password: Пароль пользователя для подтверждения
        current_user: Текущий аутентифицированный пользователь
        db: Сессия базы данных
        request: HTTP запрос для логирования

    Returns:
        dict: Сообщение об успешном завершении сессии

    Raises:
        HTTPException: 401 при неверном пароле, 404 если сессия не найдена

    Безопасность:
    - Требует подтверждение паролем
    - Логирует все попытки завершения сессий
    - Позволяет завершать только собственные сессии
    """
    # Проверка пароля для подтверждения действия
    if not verify_password(password, current_user.hashed_password):
        log_auth_event(
            db=db,
            message=f"Failed session termination attempt for user {current_user.username}:
invalid password",
            user_id=current_user.id,
            ip_address=request.client.host if request else None
        )
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect password"
        )

    # Find the session
    session = db.query(UserSession).filter(
        UserSession.id == session_id,
        UserSession.user_id == current_user.id
    ).first()

    if not session:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Session not found"
        )

    # Delete the session
    db.delete(session)
    db.commit()

    log_auth_event(
        db=db,
        message=f"User {current_user.username} terminated session {session_id}",
```

Продолжение листинга А.3

```
        user_id=current_user.id,
        ip_address=request.client.host if request else None
    )

    return {"message": "Session terminated successfully"}


@router.post("/change-password")
async def change_password(
    password_data: PasswordChange,
    current_user: User = Depends(get_current_active_user),
    db: Session = Depends(get_db),
    request: Request = None
):
    # Verify current password
    if not verify_password(password_data.current_password, current_user.hashed_password):
        log_auth_event(
            db=db,
            message=f"Failed password change attempt for user {current_user.username}:
invalid current password",
            user_id=current_user.id,
            ip_address=request.client.host if request else None
        )
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect current password"
        )

    # Update password
    current_user.hashed_password = get_password_hash(password_data.new_password)
    db.commit()

    log_auth_event(
        db=db,
        message=f"User {current_user.username} changed password",
        user_id=current_user.id,
        ip_address=request.client.host if request else None
    )

    return {"message": "Password changed successfully"}


@router.post("/delete-account")
async def delete_account(
    delete_data: UserDelete,
    current_user: User = Depends(get_current_active_user),
    db: Session = Depends(get_db),
    request: Request = None
):
    # Verify password
    if not verify_password(delete_data.password, current_user.hashed_password):
        log_auth_event(
            db=db,
            message=f"Failed account deletion attempt for user {current_user.username}:
invalid password",
            user_id=current_user.id,
            ip_address=request.client.host if request else None
```

Продолжение листинга А.3

```
        )
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect password"
        )

    # Delete the user (cascade will delete sessions and devices)
    db.delete(current_user)
    db.commit()

    log_auth_event(
        db=db,
        message=f"User account deleted: {current_user.username}",
        ip_address=request.client.host if request else None
    )

    return {"message": "Account deleted successfully"}


@router.post("/refresh-token", response_model=dict)
async def refresh_access_token(
    current_user: User = Depends(get_current_active_user),
    db: Session = Depends(get_db),
    request: Request = None
):
    """Refresh the access token before it expires"""
    # Create new access token
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={
            "sub": current_user.username,
            "role": current_user.role,
            "user_id": current_user.id
        },
        expires_delta=access_token_expires
    )

    # Create a new session
    session_token = str(uuid.uuid4())
    expires_at = datetime.utcnow() + access_token_expires

    new_session = UserSession(
        user_id=current_user.id,
        session_token=session_token,
        expires_at=expires_at,
        ip_address=request.client.host if request else None,
        user_agent=request.headers.get("user-agent") if request else None
    )

    db.add(new_session)

    # Clean up old sessions for this user
    old_sessions = db.query(UserSession).filter(
        UserSession.user_id == current_user.id,
        UserSession.expires_at < datetime.utcnow()
    ).all()
    for session in old_sessions:
```

Окончание листинга А.3

```
        db.delete(session)

    db.commit()

    log_auth_event(
        db=db,
        message=f"Token refreshed for user: {current_user.username}",
        user_id=current_user.id,
        ip_address=request.client.host if request else None
    )

    return {
        "access_token": access_token,
        "token_type": "bearer",
        "session_id": new_session.id
    }
```

Листинг А.4 – Файл конфигурации docker-compose.yml

```
services:
  postgres:
    image: postgres:15
    environment:
      POSTGRES_USER: homeuser
      POSTGRES_PASSWORD: homepassword
      POSTGRES_DB: homemanagement
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    networks:
      - home_network

  mosquitto:
    image: eclipse-mosquitto:2
    ports:
      - "1883:1883"
      - "9001:9001"
    volumes:
      - ./docker/mosquitto/config:/mosquitto/config
      - ./docker/mosquitto/data:/mosquitto/data
      - ./docker/mosquitto/log:/mosquitto/log
    networks:
      - home_network

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    environment:
      - DATABASE_URL=postgresql://homeuser:homepassword@postgres:5432/homemanagement
      - MQTT_BROKER_HOST=mosquitto
      - MQTT_BROKER_PORT=1883
      - SQLALCHEMY_POOL_SIZE=20
      - SQLALCHEMY_MAX_OVERFLOW=30
      - SQLALCHEMY_POOL_TIMEOUT=60
```

Окончание листинга А.4

```
      - SQLALCHEMY_POOL_RECYCLE=1800
    ports:
      - "8000:8000"
    depends_on:
      - postgres
      - mosquitto
    networks:
      - home_network

  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    depends_on:
      - backend
    networks:
      - home_network

networks:
  home_network:
    driver: bridge

volumes:
  postgres_data:
```

ПРИЛОЖЕНИЕ В

Листинг В.1 – исходный код программного модуля демонстрационного устройства «умного дома»

```
/*
 * Умная светодиодная лампа на ESP32
 * Поддерживает управление через MQTT:
 * - Включение/выключение
 * - Регулировка яркости (0-100%)
 * - Изменение цвета (HEX формат)
 * - Отправка текущего состояния
 */

// Подключение необходимых библиотек
#include <WiFi.h>          // Библиотека для подключения к WiFi
#include <PubSubClient.h>  // MQTT клиент для обмена сообщениями
#include <ArduinoJson.h>   // Парсинг и создание JSON сообщений
#include <FastLED.h>       // Управление светодиодной лентой

// Конфигурация подключения - обновите эти значения под свою сеть
const char* WIFI_SSID = "YourWiFiSSID";        // Имя WiFi сети
const char* WIFI_PASSWORD = "YourWiFiPassword"; // Пароль WiFi сети
const char* MQTT_SERVER = "MQTTBrokerIPAddress";     // IP адрес MQTT брокера
const int MQTT_PORT = 1883;                     // Порт MQTT брокера (стандартный 1883)
const char* DEVICE_ID = "UniqueDeviceID";           // Уникальный идентификатор устройства
const char* MQTT_USER = "";                        // Логин MQTT (оставить пустым если не
используется аутентификация)
const char* MQTT_PASSWORD = "";                   // Пароль MQTT (оставить пустым если не
используется аутентификация)

// MQTT топики для обмена сообщениями
String MQTT_TOPIC_PREFIX = "/home/device/";                                      // Префикс
всех топиков
String MQTT_STATE_TOPIC = MQTT_TOPIC_PREFIX + DEVICE_ID + "/state";          // Топик для
отправки состояния устройства
String MQTT_CONTROL_TOPIC = MQTT_TOPIC_PREFIX + DEVICE_ID + "/control";      // Топик для
получения команд управления

// Конфигурация светодиодной ленты
#define LED_PIN 12                    // Пин подключения ленты к ESP32
#define LED_COUNT 60                  // Количество светодиодов в ленте
#define LED_TYPE WS2812B              // Тип светодиодной ленты
#define COLOR_ORDER GRB               // Порядок цветов (зеленый-красный-синий)
#define STATE_REPORT_INTERVAL 5000    // Интервал отправки состояния (5 секунд)

// Инициализация массива для управления светодиодами
CRGB leds[LED_COUNT];

// Создание объектов для WiFi и MQTT подключения
WiFiClient espClient;              // WiFi клиент
PubSubClient mqttClient(espClient); // MQTT клиент

// Переменные состояния устройства
bool isOn = false;                          // Включена ли лампа
int brightness = 100;                       // Яркость (0-100%)
CRGB currentColor = CRGB(255, 255, 255);    // Текущий цвет (по умолчанию белый)
unsigned long lastStateReport = 0;          // Время последней отправки состояния
unsigned long lastReconnectAttempt = 0;     // Время последней попытки переподключения к
MQTT
```

Продолжение листинга В.1

```
void setup() {
  // Инициализация последовательного порта для отладки
  Serial.begin(115200);
  Serial.println("Smart Light starting up with FastLED");

  // Инициализация светодиодной ленты
  FastLED.addLeds<LED_TYPE, LED_PIN, COLOR_ORDER>(leds, LED_COUNT);
  FastLED.setBrightness(255); // Устанавливаем максимальную яркость (управляем яркостью через собственную логику)
  FastLED.clear();            // Выключаем все светодиоды при запуске
  FastLED.show();             // Применяем изменения

  // Подключение к WiFi сети
  setupWiFi();

  // Настройка MQTT клиента
  mqttClient.setServer(MQTT_SERVER, MQTT_PORT);  // Устанавливаем адрес и порт MQTT брокера
  mqttClient.setCallback(mqttCallback);        // Устанавливаем функцию обработки входящих сообщений

  // Применяем начальное состояние к светодиодам
  updateLEDs();
}

void loop() {
  // Проверяем подключение к WiFi и переподключаемся при необходимости
  if (WiFi.status() != WL_CONNECTED) {
    setupWiFi();
  }

  // Проверяем подключение к MQTT брокеру
  if (!mqttClient.connected()) {
    unsigned long now = millis();
    // Пытаемся переподключиться каждые 5 секунд
    if (now - lastReconnectAttempt > 5000) {
      lastReconnectAttempt = now;
      if (reconnectMQTT()) {
        lastReconnectAttempt = 0;  // Сбрасываем счетчик при успешном подключении
      }
    }
  } else {
    // Обработка входящих MQTT сообщений
    mqttClient.loop();

    // Периодическая отправка состояния устройства
    unsigned long now = millis();
    if (now - lastStateReport > STATE_REPORT_INTERVAL) {
      lastStateReport = now;
      reportState();  // Отправляем текущее состояние
    }
  }
}

/*
 * Функция подключения к WiFi сети
 */
void setupWiFi() {
  Serial.println("Connecting to WiFi...");
```

Продолжение листинга В.1

```
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

  int attempt = 0;
  // Пытаемся подключиться в течение 10 секунд (20 попыток по 0.5 сек)
  while (WiFi.status() != WL_CONNECTED && attempt < 20) {
    delay(500);
    Serial.print(".");
    attempt++;
  }

  if (WiFi.status() == WL_CONNECTED) {
    Serial.println("\nWiFi connected");
    Serial.println("IP address: " + WiFi.localIP().toString());
  } else {
    Serial.println("\nFailed to connect to WiFi");
  }
}

/*
 * Функция переподключения к MQTT брокеру
 * Возвращает true при успешном подключении
 */
boolean reconnectMQTT() {
  Serial.println("Connecting to MQTT broker...");

  // Создаем уникальный идентификатор клиента
  String clientId = "ESP32Client-" + String(DEVICE_ID);

  // Подключаемся к MQTT брокеру
  if (MQTT_USER && strlen(MQTT_USER) > 0) {
    // Подключение с аутентификацией
    if (mqttClient.connect(clientId.c_str(), MQTT_USER, MQTT_PASSWORD)) {
      onMQTTConnect();
      return true;
    }
  } else {
    // Подключение без аутентификации
    if (mqttClient.connect(clientId.c_str())) {
      onMQTTConnect();
      return true;
    }
  }

  Serial.println("MQTT connection failed, rc=" + String(mqttClient.state()));
  return false;
}

/*
 * Функция, вызываемая при успешном подключении к MQTT
 */
void onMQTTConnect() {
  Serial.println("Connected to MQTT broker");

  // Подписываемся на топик для получения команд управления
  mqttClient.subscribe(MQTT_CONTROL_TOPIC.c_str());
  Serial.println("Subscribed to " + MQTT_CONTROL_TOPIC);

  // Сразу отправляем текущее состояние после подключения
  reportState();
```

Окончание листинга В.1

```
}

/*
 * Функция обработки входящих MQTT сообщений
 * Вызывается автоматически при получении сообщения
 */
void mqttCallback(char* topic, byte* payload, unsigned int length) {
  Serial.println("Message received [" + String(topic) + "]");

  // Добавляем нулевой терминатор для корректного преобразования в строку
  char message[length + 1];
  memcpy(message, payload, length);
  message[length] = '\0';

  Serial.print("Payload: ");
  Serial.println(message);

  // Обрабатываем сообщения только из топика управления
  if (String(topic) == MQTT_CONTROL_TOPIC) {
    // Парсим JSON сообщение
    StaticJsonDocument<256> doc;
    DeserializationError error = deserializeJson(doc, payload, length);

    if (error) {
      Serial.println("Failed to parse JSON: " + String(error.c_str()));
      return;
    }

    // Флаг для отслеживания изменений состояния
    bool stateChanged = false;

    // Обработка команд включения/выключения
    if (doc.containsKey("action")) {
      String action = doc["action"].as<String>();
      if (action == "on") {
        isOn = true;
        stateChanged = true;
        Serial.println("Turning light ON");
      } else if (action == "off") {
        isOn = false;
        stateChanged = true;
        Serial.println("Turning light OFF");
      }
    }

    // Обработка изменения яркости
    if (doc.containsKey("brightness")) {
      int newBrightness = doc["brightness"].as<int>();
      if (newBrightness >= 0 && newBrightness <= 100) {
        brightness = newBrightness;
        stateChanged = true;
        Serial.println("Setting brightness to " + String(brightness));
      }
    }

    // Обработка изменения цвета
    if (doc.containsKey("color")) {
      String colorHex = doc["color"].as<String>();
```

Продолжение листинга В.1

```
      if (colorHex.startsWith("#") && colorHex.length() == 7) {
        // Конвертируем HEX цвет в RGB
        currentColor = hexToColor(colorHex);
        stateChanged = true;
        Serial.println("Setting color to " + colorHex);
      }
    }

    // Обновляем светодиоды если состояние изменилось
    if (stateChanged) {
      updateLEDs();   // Применяем новые настройки к светодиодам
      reportState();  // Отправляем обновленное состояние
    }
  }
}

/*
 * Функция обновления состояния светодиодной ленты
 * Применяет текущие настройки цвета, яркости и включения/выключения
 */
void updateLEDs() {
  if (isOn) {
    // Рассчитываем масштабирование яркости (0-100% -> 0.0-1.0)
    float scale = brightness / 100.0;
    CRGB scaledColor = CRGB(
      scale * currentColor.r,
      scale * currentColor.g,
      scale * currentColor.b
    );

    // Устанавливаем всем светодиодам одинаковый цвет
    fill_solid(leds, LED_COUNT, scaledColor);
  } else {
    // Выключаем все светодиоды
    FastLED.clear();
  }

  // Применяем изменения к физическим светодиодам
  FastLED.show();
}

/*
 * Функция конвертации HEX цвета в RGB формат
 * Принимает строку вида "#FF0000" и возвращает CRGB объект
 */
CRGB hexToColor(String hexColor) {
  // Убираем символ '#' если он присутствует
  if (hexColor.startsWith("#")) {
    hexColor = hexColor.substring(1);
  }

  // Конвертируем HEX строку в число
  uint32_t number = strtoul(hexColor.c_str(), NULL, 16);

  // Извлекаем компоненты RGB из числа
  uint8_t r = (number >> 16) & 0xFF;  // Красный канал
  uint8_t g = (number >> 8) & 0xFF;   // Зеленый канал
  uint8_t b = number & 0xFF;          // Синий канал
```

Окончание листинга В.1

```
  return CRGB(r, g, b);
}

/*
 * Функция отправки текущего состояния устройства
 * Создает JSON сообщение с текущими параметрами и отправляет в MQTT
 */
void reportState() {
  // Создаем JSON документ
  StaticJsonDocument<256> doc;

  // Добавляем параметры состояния
  doc["status"] = isOn ? "on" : "off";  // Статус включения
  doc["brightness"] = brightness;        // Текущая яркость

  // Конвертируем текущий цвет обратно в HEX формат
  char hexColor[8];
  sprintf(hexColor, "#%02X%02X%02X", currentColor.r, currentColor.g, currentColor.b);
  doc["color"] = hexColor;

  // Преобразуем JSON в строку
  String jsonString;
  serializeJson(doc, jsonString);

  // Отправляем состояние в MQTT (с флагом retain=true для сохранения последнего состояния)
  mqttClient.publish(MQTT_STATE_TOPIC.c_str(), jsonString.c_str(), true);
  Serial.println("State reported: " + jsonString);
}
```