**PEФEPAТ**

Пояснительная записка 70 стр., 28 рис, 11 источников.

Перечень ключевых слов: умный дом, *Java*, *Flutter*, разработка архитектуры, лампа с *Wi-Fi*.

Тема: «Разработка архитектуры и реализация системы умного дома».

Целью работы является разработка системы умного дома с возможностью удаленного управления и настройки через мобильное приложение.

В ходе работы планируется изучение теоретической и практической части по разработке мобильного приложения на базе фреймворка Flutter и серверной части на языке программирования *Java*.

В первой главе будет рассмотрена концепция умного дома. В данной главе будут приведены примеры умных устройств и методы их работы.

Во второй главе планируется создание архитектуры умного дома. В данной главе планируется выбрать технологии и фреймворки, которые обеспечат работоспособность планируемой системы, языки программирования вместе с аргументацией подобранных решений. Будет рассмотрен вопрос безопасности передачи данных между спроектированными системами.

В третьей главе реализуется система умного дома на базе планируемой архитектуры. Будут описаны возможности системы, как происходит взаимодействие с устройствами, сервером, мобильным приложением и базой данных.

В четвертой главе планируется рассмотрение дополнительных возможностей для расширения системы умного дома.

В пятой главе проведется технико-экономический расчет, в котором посчитаются все затраты при создании данной системы и ее внедрения в дом.

**ОГЛАВЛЕНИЕ**

[ВВЕДЕНИЕ 2](#_Toc136394925)

[1 КОНЦЕПЦИЯ УМНОГО ДОМА 4](#_Toc136394926)

[1.1 Производители устройств умного дома 5](#_Toc136394927)

[1.2 Связь устройств умного дома 11](#_Toc136394928)

[1.3 Управление умным домом 14](#_Toc136394929)

[2 РАЗРАБОТКА АРХИТЕКТУРЫ СИСТЕМЫ УМНОГО ДОМА 17](#_Toc136394930)

[2.1 Разработка требований для умного дома 18](#_Toc136394931)

[2.2 Задачи, решаемые подсистемами умного дома 19](#_Toc136394932)

[2.3 Выбор инструментов для разработки 20](#_Toc136394933)

[2.4 Выбор устройств и протокола для работы системы умного дома 28](#_Toc136394934)

[3 РЕАЛИЗАЦИЯ СИСТЕМЫ УМНОГО ДОМА НА БАЗЕ ВЫБРАННОЙ АРХИТЕКТУРЫ 30](#_Toc136394935)

[3.1 Разработка мобильного приложения 30](#_Toc136394936)

[3.2 Разработка серверного приложения 47](#_Toc136394937)

[3.3 Разработка хаба умного дома 59](#_Toc136394938)

[3.4. Добавление шифрования и деплой системы 61](#_Toc136394939)

[4 ВОЗМОЖНОСТИ РАСШИРЕНИЯ УМНОГО ДОМА 65](#_Toc136394940)

[5 ТЕХНИКО-ЭКОНОМИЧЕСКИЕ РАСЧЕТЫ 68](#_Toc136394941)

[ЗАКЛЮЧЕНИЕ 70](#_Toc136394942)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 71](#_Toc136394943)

ВВЕДЕНИЕ

**Актуальность.** С каждым годом технологии становятся более доступными и все больше встраиваются в повседневную жизнь людей, делая ее более комфортной и практичной. Благодаря быстрому развитию компьютерных систем появилась возможность удаленно контролировать и автоматизировать любые процессы. Например, при помощи автоматизированных систем контроля отпадает нужда в личной сверке данных или личном присутствии человека, следящим за некоторым объектом, так как данные системы способны самостоятельно собирать данные, при необходимости - проверять их, и отображать в любом удобном формате в виде текста, графиков или картинок. В данном случае системы автоматизации и контроля удобнее и проще в использовании, так как при помощи одного управляющего приложения можно следить за состоянием всех критичных для работы объекта систем в одном месте в легком для понимания виде. Также это помогает дополнительно обезопасить объект в случае несанкционированного проникновения. Среди других плюсов, такая система способна самостоятельно отрабатывать ситуации внештатные ситуации, например отключать электрический щит при обнаруженной протечке воды.

Данные системы существенно упрощают процесс обслуживания любого объекта, так как позволяют решить такие рутинные задачи, как:

* получение состояния отдельных датчиков, например, датчик протечки воды, состояние световых ламп;
* сбор информации о работе устройств;
* отправка уведомления пользователю в случае нештатной ситуации, например при протечке воды или открытии двери при включенной сигнализации;
* удаленное управление отдельными устройствами.

Данные системы могут обеспечивать безопасность и работоспособность не только коммерческих объектов, но и домов и квартир частных лиц. Примером такой системы может являться умный дом.

**Объект исследования:** технология беспроводного управления беспроводными устройствами в системе умного дома.

**Предмет исследования:** внутреннее устройство, безопасность и конфиденциальность систем умного дома.

**Цель работы и задачи исследований.** Целью данной работы является разработка системы умного дома, позволяющая удаленно управлять подключенными устройствами через мобильное приложение. Для достижения поставленной цели необходимо решить следующие задачи:

1. Выбор необходимой архитектуры для умного дома
2. Разработка серверной части системы
3. Написание мобильного приложения – клиента
4. Создание системы домашнего хаба для объединения устройств умного дома
5. Рассмотрение вопроса безопасности передаваемых данных через сеть Интернет
6. Развертывание приложения на рабочем окружении

В рамках проведённых теоретических и экспериментальных исследований все результаты были получены автором самостоятельно.

# **1 КОНЦЕПЦИЯ УМНОГО ДОМА**

Умным домом можно назвать систему, которая собирает данные с различных датчиков, установленных в доме пользователя, и удаленно управляет ими при помощи команд из пользовательского интерфейса. Умный дом – это гибко настраиваемая система, с ее помощью можно автоматизировать практически любые домашние процессы.

Первые попытки автоматизировать дом были предприняты в середине двадцатого века. Журнал “*Popular Mechanincs*” в издании за декабрь 1950-года напечатал историю об изобретателе Эмиле Мэтиасе [1]. Он встроил в свой дом в Джексоне, штат Мичиган, США, множество кнопок, которые отвечали за выполнение определенных действий. Например, Эмиль с помощью кнопки мог открыть дверь гаража или включить и выключить радио. На рисунке 1 приведен план его дома с обозначениями установленных устройств. Этот дом получил прозвище «*Push-Button Manor*», что в переводе означает «кнопочное поместье». Из статьи также можно узнать, что изобретатель спрятал в стенах своего дома больше семи тысяч футов проводки, а управляющая часть этого умного дома находилась в его кладовке. Журналист Артур Рейлтон оценил качество работы изобретателя: в доме нет ни одного висящего провода, который бы выдавал наличие любых вспомогательных систем.

Через двадцать пять лет, в 1975-ом году, компания *Pico Electronics* создала собственный стандарт для связи устройств: *X10*. Данный стандарт управлял электрическими приборами при помощи силовых линий, позволяя включать и выключать их при помощи пульта дистанционного управления. Это решение было популярно в виду дешевизны и простоты в установке.

Можно увидеть, что интерес к автоматизации жилых помещений с каждым годом только растет. Например, согласно исследованию о российском рынке умного дома [2], количество хабов умного дома в России в 2022 году составило 176 тысяч штук, “рост продаж в 2 раза год к году”. Это говорит об интересе людей к новшествам в сфере автоматизации умных домов и их открытости к новым технологиям.

## **1.1 Производители устройств умного дома**

На данный момент существует множество реализаций умного дома от разных производителей. Данные решения продаются как отдельными устройствами, так и набором, позволяя пользователю самостоятельно решить, какие системы дома он хочет автоматизировать и иметь возможность управлять удаленно.

Лидирующими производителями устройств для умного дома в странах СНГ являются компании *AJAX, Aqara* и Яндекс. Рассмотрим каждого из производителей подробнее.

*AJAX*

*AJAX* – это украинская компания, специализирующаяся на повышении безопасности объектов путем установки датчиков по всему дому. Их устройства созданы для защиты объектов при нештатных ситуациях, таких как:

* взлом дверей;
* прорыв труб;
* пожар;
* попытка ограбления.

Управление происходит через собственное мобильное приложение, позволяющее следить за безопасностью объектов. Для работы системы необходим хаб, который следит за состоянием всех установленных датчиков и передает данные на сервер.

Важной функцией их хабов является дублирование систем энергообеспечения. В случае, если электрическая сеть дома будет отключена, например, из общего щитка в подъезде, то хаб имеет возможность через *SIM*-карту, вставленную в нее, отправлять уведомления о нарушении электропитания, так как хаб имеет встроенный аккумулятор и *GSM*-модуль. Так же и с *WI-FI*: если хаб потеряет домашнюю сеть *WI-FI*, то он может «общаться» с серверами систем через мобильную сеть или *SMS*-сообщения [3].

Отличительной чертой данных устройств является возможность подключения к пульту охранной системы. Системы *AJAX* могут отправлять сигнал тревоги, на который отреагируют сотрудники правоохранительных органов.

Список возможных к установке датчиков достаточно большой, среди них:

* датчики протечки;
* датчики движения;
* пожарные датчики;
* датчики открытия;
* датчики разбития стекла;
* сирены;
* ретрансляторы;
* тревожные кнопки;
* устройства автоматизации.

*Aqara*

Другой компанией, поставляющей устройства умного дома, является *Aqara*. *Aqara* – это суб-бренд китайской компании *Xiaomi*, который специализируется на электронике и датчиках умного дома. Выбор устройств у компании достаточно обширен, это:

* веб-камеры;
* умные замки;
* умные колонки с голосовыми ассистентами;
* шторы с электроприводом;
* лампы;
* реле;
* датчики протечки;
* мониторы качества воздуха;
* датчики открытия дверей;
* датчики движения;
* датчик вибраций.

В системе умного дома от *Aqara* реализована система сценариев. Сценарии – это автоматический старт последовательности действий при совершении указанного пользователем действия. Например, если пользователь зайдет в жилье, то автоматически включатся лампы в прихожей и зале. Также может начать работать сирена, оповещающая о тревоге в случае нарушения протокола безопасности [4]. Другой сценарий: при срабатывании датчика вибрации на телефон придет уведомление, а затем включится сирена и начнется запись с видеокамеры. Данные решения помогают обезопасить помещение и быстрее отреагировать в случае нештатной ситуации.

Важной функцией умного дома от *Aqara* является возможность добавления пользователей в уже существующий умный дом. Например, администратор такого умного дома может открыть доступ к устройствам другим людям (например, членам семьи). Эта функция позволяет делегировать полномочия по использованию устройств и следить за состоянием умного дома и статистикой использования датчиков. Тажке это повышает безопасность, так как владельцу умного дома не требуется передавать собственные данные для входа в систему третьим лицам.

Управлением умным домом также происходит через мобильное приложение, которое соединяется с серверами и передает данные к хабу, который в свою очередь также следит за состоянием установленных в доме датчиков и передает информацию на сервера компании.

Яндекс

Другим большим производителей умных устройств является компания Яндекс. Среди их устройств можно отметить «Яндекс.Станция», умную колонку с голосовым управлением [5].

У Яндекса набор предметов домашней автоматизации меньше, чем у Aqara:

* датчики движения и освещения;
* беспроводные колонки;
* датчики температуры и влажности;
* датчики протечки воды;
* *Wi-Fi* лампы;
* хабы;
* музыкальные колонки.

Управление умным домом осуществляется при помощи голосового ассистента «Алиса» и мобильного приложения.

Отличительной чертой устройств Яндекса является необязательность установки хаба при наличии Яндекс.Станции. Станция используется одновременно и как хаб для связи с устройствами, и как музыкальная колонка с голосовым ассистентом. Это решение можно назвать практичным и удобным, так как не нужно покупать отдельное устройство для установки связи с умными устройствами.

В умном доме от Яндекса поддерживаются и другие устройства, работающие по протоколу *ZigBee*, в том числе и от компании *Aqara*.

1.2 Связь устройств умного дома

Устройства умного дома постоянно подключены к сети и обмениваются данными с сервером для быстрой и надежной обработки команд. Разные производители используют разные решения и методы передачи данных в своих решениях. Например, в системах умного дома устройства могут обмениваться командами по протоколу *ZigBee, Wi-Fi, Bluetooth* и другими. Рассмотрю протоколы для устройств умного дома.

*ZigBee*

*ZigBee* — это протокол беспроводной связи, который был разработан для создания низкопотребляющих и недорогих сетей устройств, используемых в умном доме, промышленности и других областях. Протокол *ZigBee* работает на частоте 2,4 ГГц и имеет дальность передачи данных до 20 внутри помещения [6].

Одной из главных особенностей протокола *ZigBee* является его низкое энергопотребление. Устройства, работающие на протоколе *ZigBee*, могут работать на батарейках до нескольких лет, что делает его хорошим выбором для использования в умном доме: пользователю не нужно постоянно следить за уровнем батареи в устройстве.

Протокол *ZigBee* поддерживает множество топологий сетей, включая «звезда», «сетка» и «дерево». При типе «сетка» количество переходов при отправке от устройства к устройству не ограничено, то есть если одно устройство потеряло контакт с хабом, то другие устройства имею возможность ретранслировать команды между собой. Также при данной топологии необязательно постоянное подключение к хабу. Максимальное расстояние между устройствами для передачи данных может достигать 20 метров. Максимальное число поддерживаемых устройств – 65000. Протокол также имеет множество функций безопасности, таких как шифрование по стандарту *AES-128* и аутентификация, что делает его более надежным и безопасным для использования.

Протокол *ZigBee* предусматривает работу на частоте 2.4 ГГц, таком же, как и *Wi-Fi*, таким образом возможны помехи в их работе. В Европе данный стандарт работает на частоте 868 МГц.

*Wi-Fi*

*Wi-Fi* — это популярный протокол беспроводной связи, который позволяет передавать данные по радиоволнам. В контексте умного дома, *Wi-Fi* используется для подключения устройств к сети Интернет и обеспечивает их удаленное управление через приложение.

Одним из главных преимуществ *Wi-Fi* является его широкое распространение и доступность. Большинство домов уже имеют установленную *Wi-Fi*-сеть, что делает его очень удобным для использования в умном доме. Также *Wi-Fi* имеет высокую скорость передачи данных и обеспечивает высокую производительность устройств в умном доме. Это позволяет управлять устройствами в режиме реального времени и обеспечивает быстрый доступ к данным и контролю устройств.

Среди недостатков рассматриваемого протокола можно отметить высокое энергопотребление и не самую высокую надежность для использования в критических системах умного дома. Кроме того, *Wi-Fi* может столкнуться с проблемами перегруженности сети, особенно если в доме много *WI-FI* устройств. Такой стандарт подойдет для работы устройств, имеющих постоянное подключение к электросети.

*Bluetooth*

*Bluetooth* — это протокол беспроводной связи, который используется для передачи данных между устройствами в непосредственной близости друг от друга. В контексте умного дома, *Bluetooth* редко используется для непосредственной работы системы умного дома. Некоторые производители используют его для первоначальной настройки устройства.

Одной из главных преимуществ протокола *Bluetooth* является его низкое энергопотребление. *Bluetooth* устройства могут работать на батарейках до нескольких лет, это делает его возможным вариантом для использования в системах, рассчитанных на долгое время работы. *Bluetooth* имеет невысокую скорость передачи данных, около 1 Мбит/с, но этого достаточно для передачи запросов.

Однако, *Bluetooth* имеет свои недостатки. У данного протокола небольшой радиус действия, что ограничивает его использование в больших домах. Кроме того, *Bluetooth* не поддерживает топологию «сетка», не позволяя через себя передавать данные на другие устройства, что является критичным фактором не в пользу данного протокола.

*Z-Wave*

*Z-Wave* — это протокол беспроводной связи, который используется для создания сетей устройств в умном доме.

Протокол похож по своим характеристикам на *ZigBee*: он также использует топологию сети «сетка», устройства также могут передавать через себя команды на другие устройства, имеет такой же стандарт шифрования передаваемых данных *AES-128*. Отличия заключаются в ограниченности количества переходов между устройствами: «переслать» через себя команду могут максимум 4 устройства. Также ограничено максимальное количество устройств – всего 232.

В Европе данный протокол функционирует на волне 868-869 МГц. Для функционирования *Z-Wave* необходим центральный хаб.

*Matter*

*Matter* — это протокол для умного дома, который разрабатывается в рамках сотрудничества между крупными компаниями, такими как *Apple, Amazon, Google* и др. Он был создан для упрощения и унификации процесса взаимодействия между устройствами в умном доме.

*Matter* выделяется своей универсальностью. Он позволяет различным устройствам в умном доме взаимодействовать между собой, независимо от их производителя, типа или используемого протокола. Это значит, что устройства, работающие на протоколе *Matter*, могут легко интегрироваться и работать вместе, что делает управление умным домом более удобным и простым.

Matter позволяет, например, использовать *Bluetooth* для первоначальной настройки устройства, а *Wi-Fi* – для передачи данных.

Протокол *Matter* также обеспечивает высокую безопасность и защиту данных. Он использует шифрование и аутентификацию для защиты данных и устройств в умном доме от взлома и хакерских атак.

Таким образом, *Matter* – это перспективный стандарт для решения проблемы использования разными устройствами разных протоколов. Он постепенно развивается, и производители еще не в полной мере начали его использование.

1.3 Управление умным домом

В прошлом параграфе было выяснено, что множество производителей оборудования для умного дома используют самые разнообразные протоколы связи, и вместе с этим разрабатывают свое мобильное приложение для оркестрирования устройствами. Это ведет к разрозненному и неудобному управлению умным домом через множество приложений. Для избежания путаницы существует множество фреймворков для объединения устройств в одном приложении. Такими приложениями являются *Apple HomeKit, Google Home и Samsung SmartThings*.

Данные системы позволяют подключать к себе как уже установленные устройства от других производителей, так и устройства, работающие напрямую с экосистемой *Google, Apple и Samsung*. Такие решения позволяют бесшовно объединить устройства в одном приложении, добавляя удобства при использовании умного дома.

Можно сделать вывод, что концепция умного дома существует достаточно долгое время и все еще интересна потребителю. В своих решениях производители устройств умного дома используют связку хаба и мобильного приложения для управления умным домом как самый надежный и удобный вариант. Хаб в данных решениях отвечает за передачу команд с и на устройства, что существенно облегчает управление устройствами в плане разработки такого решения.

Система умного дома существенно облегчает жизнь людей, позволяя удаленно следить за состоянием жилья через Интернет. Встает вопрос о приватности такого решения: как происходит обработка данных, имеет ли производитель полный доступ к устройствам и насколько эта интеграция устройств в жизнь человека расходится с правами на неприкосновенность частной жизни.

Существуют открытые программные решения для оркестрирования устройствами умного дома, которые пользователь может самостоятельно установить дома, но это предполагает более глубокое понимание устройства сети, а также настройка такого решения может занять больше времени. Такие решения существуют скорее для энтузиастов, а не для обычного потребителя, так как ручная настройка может запутать людей.

Также существуют опасения, что компании-производители могут использовать свои устройства для продажи данных пользователей в маркетинговых целях. Не маловероятно, что устройства могут быть скомпрометированы для слежки за пользователями. Производители не выкладывают ПО в открытый доступ для общественного анализа систем, из-за чего возникает вопрос о приватности и методах обработки данных компаниями. Пользователям остается вслепую доверять производителям, что они не будут использовать или обрабатывать информацию с устройств умного дома в корыстных целях.

Данные опасения побудили автора на создание собственной реализации системы умного дома.

# **2 РАЗРАБОТКА АРХИТЕКТУРЫ СИСТЕМЫ УМНОГО ДОМА**

Архитектура программного обеспечения – это способ связи компонентов системы между собой. Не существует четкого определения, что такое архитектура ПО, это можно назвать «рекомендациями» как написать поддерживаемый и чистый код.

Архитектура приложения определяет, какие функции системы за что отвечают и как они взаимодействуют между собой. Архитектура приложения разрабатывается после анализа требований к системе.

Можно провести параллели между планом здания и программой: то, по каким технологиям сооружено здание, определит его свойства, прочность, долговечность и надежность. Можно создать систему и без продумывания архитектуры и методов взаимосвязи компонентов, но это может усложнить процесс дальнейшего развития ПО в виду хаотичного расположения компонентов и сложной структуры, которая решала только сиюминутные задачи.

Разработанная и продуманная архитектура программного обеспечения позволять решить следующие задачи при создании необходимой системы:

* выделяет внутреннюю структуру программы и описывает, на каких уровнях выполняются необходимые задачи;
* упорядочивает взаимодействие внутренних элементов при вызове необходимых методов;
* определяет первостепенные и второстепенные элементы для реализации в планируемой системе;
* помогает определить необходимый стек технологий для имплементации в проекте;
* дает представление о масштабируемости ПО и его слабых местах;
* упрощает дальнейшее сопровождение ПО.

Можно представить архитектуру ПО как многоуровневую абстракцию, где на верхнем уровне будет изображена концептуальная схема разрабатываемого приложения, а на нижнем – используемые фреймворки и технологии, с помощью которых компоненты системы будут взаимодействовать друг с другом.

2.1 Разработка требований для умного дома

Выделю требования, которым должна отвечать система умного дома.

* легкость настройки, добавления и удаления умных устройств;
* безопасность передаваемых данных между компонентами системы;
* быстрота выполнения команд (меньше 2 секунд между отправкой команды и ее выполнения);
* возможность управления устройствами вне дома;
* простота использования;
* сохранение метрик отдельных умных устройств;
* корректная обработка возможных ошибок;
* поддержка постоянного соединения с устройствами через сеть Интернет.

Исходя из требований, система умного дома будет состоять из 3 компонентов: сервер, хаб и мобильное приложение. Данная архитектура используется в решениях от компаний-производителей, анализ которых был проведен в первой главе работы. Подобная архитектура позволяет удобно и удаленно управлять умными устройствами через мобильное приложение. Серверная часть должна принимать все запросы и обрабатывать их, а хаб – поддерживать соединение с сервером и устройствами.

2.2 Задачи, решаемые подсистемами умного дома

Каждая подсистема должна выполнять свои задачи в соответствии с требованиями к системе. Обозначим задачи, которые должны выполнять компоненты системы умного дома.

Задачи серверной части:

* аутентификация и авторизация пользователей;
* поддержка протоколов шифрования передаваемых данных;
* поддержка постоянного подключения к базе данных;
* передача пользователю информацию о подключенных к его умному дому устройствах;
* отправка и обработка команд для управления устройствами через собственный API;
* сохранение метрик устройств в базе данных;
* обработка возможных ошибок.

Задачи мобильного приложения:

* корректное и понятное отображение устройств умного дома;
* поддержка шифрования подключений;
* возможность настройки каждого устройства;
* удаление и добавление устройств;
* удаление и добавление хабов.

Задачи хаба:

* установка и поддержание постоянного защищенного соединения с сервером;
* обработка команд от сервера;
* передача на сервер ответов от умных устройств.

Исходя из требований и задач каждого компонента системы, рассмотрю существующие фреймворки и языки программирования для реализации данных требований в системе умного дома.

2.3 Выбор инструментов для разработки

Выбор решения для компонента сервера и хаба.

Сервер в системе умного дома является головным элементом: он получает запросы от приложения и поддерживает подключение с хабом. Его задача – корректно получать любую информацию, обрабатывать и сохранять.

Серверная часть также должна шифровать данные и проверять пользователя на возможность совершения действий внутри системы.

Рассмотрим существующие фреймворки для разработки серверных решений.

*Spring Boot*

*Spring Boot* — это фреймворк для разработки приложений на языке Java, который позволяет быстро и легко создавать приложения с минимальной конфигурацией. *Spring Boot* использует принципы автоматической конфигурации и управлением зависимостями. Данный фреймворк обеспечивает высокую производительность и масштабируемость приложений, что делает его хорошим выбором для разработки как небольших, так и больших проектов.

Помимо *Spring Boot*, существуют другие фреймворки, входящие в проект Spring для разработки на *Java*. Рассмотрим их.

*Spring Security*

Данный фреймворк отвечает за безопасность приложения. *Spring Security* предоставляет множество функций, таких как защита от атак типа *CSRF* и *XSS*, поддержка различных механизмов аутентификации (например, по логину и паролю или *OAuth* 2), возможность настройки прав доступа на основе ролей и правил, поддержка шифрования паролей и другое. *Spring Security* является одним из самых популярных фреймворков для обеспечения безопасности приложений на языке *Java*.

*Spring Data*

Рассматриваемый компонент отвечает за работу с базами данных. Он поддерживает как реляционные базы данных, например *PostgreSQL*, так и нереляционные базы данных. *Spring Data* позволяет разработчикам создавать и использовать простые интерфейсы для выполнения операций с базами данных, таких как создание, чтение, обновление и удаление данных. Он также предоставляет возможность автоматически создавать запросы на основе методов интерфейса, что упрощает написание кода. Также фреймворк обеспечивает поддержку транзакций, таким образом обеспечивая целостность данных.

*Spring MVC*

Данный модуль предназначен для разработки веб-приложений. Он предоставляет возможности для работы с веб-компонентами, такими как HTTP-запросы и ответы, сессии, параметры запросов и другие. *Spring MVC* использует паттерн проектирования *Model-View-Controller* для разделения логики приложения на три компонента: модель, представление и контроллер. Модель представляет данные приложения и обеспечивает доступ к ним. Представление отвечает за отображение данных пользователю. Контроллер обрабатывает запросы пользователя и управляет взаимодействием модели и представления.

*Spring Boot Starter WebSocket*

Данный модуль обеспечивает поддержку постоянной двусторонней связи между сервером и клиентом. Он использует протокол *WebSocket* для обмена данными. *Spring Boot Starter WebSocket* дает возможность создавать собственные *WebSocket*-эндпоинты на сервере и поддерживает подключение к эндпоинтам на стороне клиента. Он также добавляет поддержку аннотаций для обработки сообщений, отправленных через *WebSocket*-соединение.

*Node.JS*

Данный фреймворк создан для запуска *JavaScript* кода в серверных приложениях. Одним из основных преимуществ *Node.js* является его асинхронная модель выполнения кода. Вместо того, чтобы блокировать выполнение кода до завершения операции ввода-вывода, *Node.js* использует неблокирующие операции ввода-вывода, что позволяет приложению продолжать работу без ожидания завершения операции ввода-вывода. Минимальная конфигурация *Node.JS* не позволяет написать приложение, однако большое количество пакетов доступно для скачивания, что добавляет гибкости и расширяемости в проект.

*Rails on Ruby*

*Rails on Ruby* - это фреймворк для веб-разработки, написанный на языке *Ruby*. Он был создан в 2004 году Дэвидом Хейнемеер Ханссоном и с тех пор стал одним из самых популярных инструментов для создания веб-приложений.

*Rails on Ruby* предоставляет множество готовых решений для различных задач, таких как работа с базами данных, авторизация и аутентификация пользователей, обработка форм и многое другое. Это позволяет разработчикам сосредоточиться на создании бизнес-логики приложения, не тратя много времени на написание базового функционала.

Важной особенностью данного фреймворка является следование принципам конвенции над конфигурацией. Это означает, что изменять конфигурацию нужно только тогда, когда она не удовлетворяет требованиям. Это позволяет сосредоточиться на разработке, а не настройке программы.

При выборе фреймворка для написания системы стоит в первую очередь смотреть на язык программирования, который используется для создания приложения, скорость работы, уровень сложности и стоимость разработки. Все вышеперечисленные фреймворки способны выполнить серверные задачи, однако в данной работе будет использоваться фреймворк *Spring* и язык программирования *Java*. Данный выбор продиктован опытом разработки автором серверных приложений с использованием данного языка программирования, а также удобной конфигурацией, позволяющей быстро создавать необходимые компоненты для нужд системы. Также важной составляющей при выборе является безопасность, так как *Spring Security*, входящий в проект Spring*,* поддерживает необходимые функции по авторизации и аутентификации.

Для обеспечения безопасности передаваемых данных будет использоваться бесплатный сертификат *Let’s Encrypt*, дающий возможность шифрования данных при передаче. Данный сертификат будет применен в веб-сервере *nginx*, выступающим в роле прокси-сервера, то есть перенаправляющем запросы из внешней сети в локальную. Данный выбор был сделан в виду простоты настройки и дополнительной безопасности. Также возможно использование прокси-сервера для кеширования контента, что увеличивает скорость возвращения данных.

Список фреймворков и языков программирования для создания хаба умного дома совпадает со списком для серверной части системы. Для нужд хаба в этой работе выбран фреймворк *Spring* с поддержкой протокола *WebSocket*, который позволит инициализировать подключение к серверу умного дома. Выбор данного фреймворка также продиктован удобством разработки и масштабируемостью.

Выбор базы данных

База данных – важнейшая часть любой системы. В ней будут хранится все данные обо всех устройствах, подключенных к умному дому пользователей. Помимо этого, она будет хранить данные о самих пользователях, статистику о каждом устройстве и другую важную информацию.

Существует множество систем управления базами данных, например *MySQL, PostgreSQL* и *MongoDB*. Рассмотрим их подробнее.

*MySQL* — это система управления реляционными базами данных, которая используется для хранения, управления и доступа к данных. *MySQL* является одной из самых популярных СУБД в мире и используется в различных приложениях, включая веб-сайты, блоги, форумы, электронную коммерцию и многие другие.

*MySQL* был разработан в 1995 году и с тех пор был приобретен компанией *Oracle*. Данная СУБД поддерживает множество функций, включая транзакции, хранимые процедуры, триггеры, индексы и многие другие. *MySQL* также поддерживает множество языков программирования, включая *PHP, Java, Python* и другие.

СУБД *MySQL* ориентирована на быстроту выполнения операций: в ней поддерживается стандартные типы данных, но, например, отсутствует тип данных jsonb, который есть в *PostgreSQL*.

Вторая для рассмотрения СУБД – *PostgreSQL*. Это система управления реляционными базами данных с открытым исходным кодом, которая используется для хранения, управления и доступа к большим объемам данных. *PostgreSQL* был разработан в 1986 году и с тех пор получил широкое распространение в различных приложениях, включая веб-сайты, приложения для анализа данных, финансовые системы и многие другие.

Одной из основных причин популярности *PostgreSQL* является его способность к обработке сложных запросов и поддержке расширенных функций, таких как триггеры, хранимые процедуры и транзакции.

*PostgreSQL* имеет множество функций, которые делают его более гибким и мощным, чем другие СУБД. Например, *PostgreSQL* поддерживает полнотекстовый поиск, географические данные и *JSON*-данные. Он также имеет множество инструментов и ресурсов для управления базами данных, таких как *pgAdmin*, *phpPgAdmin* и другие.

*PostgreSQL* имеет открытый исходный код и доступен для загрузки и использования бесплатно.

*PostgreSQL* является надежной и мощной СУБД, которая может обрабатывать большие объемы данных и поддерживать вложенные функции. Данная СУБД также может быть легко масштабирована для управления большими объемами данных и высокой нагрузки.

Третья рассматриваемая база данных - *MongoDB* - документ-ориентированная база данных, которая хранит данные в виде документов, а не таблиц. Она позволяет гибко изменять структуру данных в процессе работы, не придерживаясь фиксированной схемы данных. *MongoDB* поддерживает горизонтальное масштабирование, что позволяет распределить данные на несколько серверов и обеспечить высокую доступность.

Данная СУБД имеет мощный язык запросов, который позволяет выполнять сложные запросы и агрегации данных. *MongoDB* также поддерживает индексы, что повышает производительность при выполнении запросов на больших объемах данных.

*MongoDB* имеет встроенную поддержку репликации и шардинга, что обеспечивает высокую доступность и масштабируемость. Она поддерживает различные форматы хранения данных, включая *JSON*, *BSON* и *CSV*.

Данная СУБД хорошо подходит для кеширования данных или для проектов, в которых структура данных может со временем измениться.

Исходя из перечисленных СУБД, в данной работе будет использоваться *PostgreSQL*. Данный выбор продиктован опытом автора в разработке приложений с использованием СУБД *PostgreSQL* и общей надежностью БД. В ней поддерживаются все принципы *ACID*, то есть атомарность, согласованность, изолированность и долговечность, что важно при хранении данных. Также важным параметром в базе данных является поддержка типа данных *jsonb,* необходимый в данной работе для хранения записей.

Выбор фреймворка для клиентского приложения.

Мобильное приложение является управляющим элементом системы умного дома. Через него пользователь будет взаимодействовать с умными устройствами.

Мобильное приложение должно вне зависимости от местоположения пользователя отправлять запросы на сервер и получать от него ответ.

Существует множество фреймворков и языков программирования для создания мобильных приложений, среди самых популярных можно отметить следующие:

* Фреймворк *Flutter* от *Google*
* Язык программирования *Swift* для *Apple* устройств
* Язык программирования *Kotlin* для *Android* устройств

Рассмотрю данные фреймворки.

*Flutter* — это фреймворк для разработки мобильных приложений, который был разработан компанией *Google*. *Flutter* использует язык программирования *Dart* и позволяет разработчикам создавать мобильные приложения. Отличительная черта данного фреймворка – возможность создания приложения с использованием одинакового кода для нескольких платформ, например *Android*, *iOS* и *Web*. Такие приложения называются кроссплатформенные.

*Flutter* использует в себе концепцию виджетов – это отдельные элементы интерфейса, с которыми взаимодействует пользователь. Во всем приложении выстраивается иерархия виджетов, которые можно настроить согласно требованиям. Этими виджетами являются как иконки и кнопки, так и тулбары и невидимые пользователю элементы.

*Flutter* имеет множество преимуществ, которые делают его привлекательным для выбора. Одним из главных преимуществ является его быстродействие и производительность. *Flutter* использует свой собственный движок рендеринга, который позволяет создавать красивые и плавные пользовательские интерфейсы. Скорость рисовки интерфейса достигает 60 кадров в секунду согласно заверениям *Google* [8].

*Flutter* также имеет мощную систему горячей перезагрузки, которая позволяет разработчикам мгновенно видеть изменения в приложении при внесении изменений в код. Это значительно ускоряет процесс разработки и упрощает тестирование приложений.

Следующий рассматриваемый объект – это язык программирования *Swift*, разработанный компанией *Apple* в 2014 году для создания приложений на *iOS*, *macOS*, *watchOS* и *tvOS*. Swift является мощным и гибким языком программирования, который может быть использован для создания различных типов приложений, включая мобильные приложения, настольные приложения и серверные приложения. Можно назвать *Swift* единственным выбором при написании нативного приложения для платформы *Apple*.

*Swift* также имеет мощную систему типов данных, которая позволяет разработчикам создавать безопасный и надежный код. В *Swift* имеется множество функций, которые делают его более гибким и мощным, таких как опциональные типы данных, функции высшего порядка, замыкания и другие.

*Swift* содержит множество инструментов и ресурсов для управления памятью, таких как автоматическое управление памятью и *ARC* (*Automatic Reference Counting*). Это позволяет разработчикам создавать приложения с высокой производительностью и эффективностью.

В *Swift* имеется множество инструментов и библиотек для создания приложений, таких как *UIKit*, *SwiftUI*, *Core* *Data* и другие. Он также имеет активное сообщество разработчиков, которые создают и поддерживают множество пакетов и библиотек для Swift.

Третий рассматриваемый элемент – *Kotlin*. Это статически типизированный язык программирования, который работает на виртуальной машине *Java*. Разработан компанией *JetBrains*. Этот язык программирования сочетает в себе функциональные и объектно-ориентированные возможности, что делает его очень гибким и удобным для разработки широкого спектра приложений.

*Kotlin* также предоставляет множество возможностей для безопасности, сокращения количества ошибок и повышения производительности приложений.

*Kotlin* стал официальным языком разработки для *Android*, что привело к резкому росту его популярности в последние годы.

Для разработки мобильного приложения будет использоваться фреймворк *Flutter*. Данный выбор обоснован кроссплатформенностью, а значит и экономичностью разработки, так как приложение может быть выпущено под несколько платформ. Также для мобильного приложения не требуются специфичные для платформы функции, соответственно *Flutter* является хорошим выбором для написания приложения. Дополнительным плюсом в сторону выбора *Flutter* является отсутствие сложной логики в мобильном приложении: ему достаточно передавать и отправлять данные, а также корректно отрисовывать интерфейс.

2.4 Выбор устройств и протокола для работы системы умного дома

В первой главе были рассмотрены умные устройства, доступные потребителю для установки в своем жилье. Для данной работы будут использованы две *Wi*-*Fi* лампы *Sonoff*, которые будут установлены в цоколь и подключены в режиме *DIY*-*mode*. Данный режим позволять использовать *API* устройства для управления им в домашней сети без подключения к серверам компании [9].

Транспортом для передачи команд будет выступать *Wi-Fi* в виду простоты настройки и наличия постоянного источника питания.

Таким образом, в данной главе было выяснено, что для системы умного дома будут использоваться следующие фреймворки, программы и устройства:

* *Spring Framework* для серверной части и хаба умного дома;
* Фреймворк *Flutter* для разработки мобильного приложения;
* СУБД *PostgreSQL* для хранения данных;
* *nginx* для проксирования запросов к серверной части системы и реализации шифрования передаваемых данных;
* Умные лампы *Sonoff* и протокол *Wi-Fi* для передачи данных;

# **3 РЕАЛИЗАЦИЯ СИСТЕМЫ УМНОГО ДОМА НА БАЗЕ ВЫБРАННОЙ АРХИТЕКТУРЫ**

В предыдущей главе было решено, что система умного дома будет написана на языках программирования *Dart* (для мобильного приложения) и Java (для сервера и домашнего хаба).

Базовый принцип работы системы умного дома следующий: пользователь из мобильного приложения отправляет запрос на сервер. Сервер авторизовывает пользователя, выполняет команду и отправляет ответ обратно пользователю. Таким образом будет происходить коммуникация систем при отправке команд из приложения.

В следующих параграфах опишем процесс разработки приложения.

3.1 Разработка мобильного приложения

Мобильное приложение в системе умного дома – главный метод управления системой. Оно должно отображать состояние подключенных устройств, управлять ими и уметь настраивать другие устройства.

Пользовательский интерфейс состоит из трех основных компонентов: авторизация, управление подключенными устройствами и настройки умного дома. Подробно опишем, как создавались эти компоненты.

Для компонента авторизации пользователя в мобильном приложении используется библиотека “*flutter-bloc*” для реализации авторизации пользователя вместе с методами для загрузки и обработки данных.

Данная библиотека реализует шаблон *BLoC* – это акроним от словосочетания «*Business Logic Component*» – архитектурный паттерн для разработки мобильных и веб-приложений, который разделяет приложение на три основных компонента: слои представления, бизнес-логики и данных.

Слой представления отвечает за отображение данных пользователю и взаимодействие с ним. Он содержит пользовательский интерфейс и обрабатывает пользовательские действия, такие как клики на кнопки или ввод текста.

Слой бизнес-логики содержит код приложения, который обрабатывает данные, полученные от уровня представления, и взаимодействует с уровнем данных для получения и сохранения данных. Слой бизнес-логики должен быть независим от слоёв представления и данных, чтобы обеспечить гибкость и возможность переиспользования кода.

Слой данных отвечает за хранение и получение данных в приложении. Он может использовать различные источники данных, такие как базы данных, файлы или внешние *API*.

*BLOC* архитектура позволяет разделить приложение на независимые компоненты, что упрощает разработку, тестирование и поддержку приложения. Она также обеспечивает гибкость и возможность переиспользования кода, что позволяет быстро адаптировать приложение к изменяющимся требованиям.

Последовательность работы *BLoC* компонентов имеет следующие шаги:

* Пользователь нажимает на элемент интерфейса, который запускает событие, необходимое для обработки.
* *BLoC* компонент определяет событие и выполняет необходимое для данного события заданную бизнес-логику.
* После завершения выполнения бизнес-логики *BLoC* генерирует состояние, которое реализуется в пользовательском интерфейсе.

Данный код получает экземпляр класса *AuthAPI*, который содержит в себе метод для отправки данных на сервер с логином и паролем пользователя.

*BLOC*-компонент имеет 4 состояния: *LoginInitial*, *LoginInProgress*, *LoginSuccess* и *LoginFailure*. Они используются для представления состояния входа пользователя в систему.

*LoginInitial* – это начальное состояние, которое включается при старте приложения.

*LoginInProgress* – состояние, когда пользователь вводит или ввел свои данные и нажал кнопку «войти»;

*LoginSuccess* – состояние, когда пользователь успешно вошел в систему;

*LoginFailure* – пользователю не удалось войти в систему;

*LoginBloc* – это класс, в котором реализуется бизнес-логика входа в приложение. При нажатии на кнопку «*Login*» вызывается метод \_*handleLoginWithEmailAndPasswordEvent*, который в свою очередь вызывает метод login в *AuthAPI* и принимает токен, идентифицирующий пользователя в данной сессии при успешном входе. При неудачном входе в нижней части экрана появляется предупреждение об ошибке.

После успешного входа пользователю открывается экран со всеми подключенными устройствами. Данный экран использует виджет *FutureBuillder*.

*FutureBuilder* – это виджет, который зависит от асинхронных методов загрузки данных и возвращает виджеты в зависимости от статуса и наличия этих данных. В листинге указано, что *FutureBuilder* принимает метод \_*getListOfDevices* в качестве асинхронного метода, который в свою очередь возвращает данные от *getListOfDevices* в классе *AuthAPI*, отправляющий запрос на сервер для получения списка всех доступных пользователю устройств. При наличии данных *FutureBuilder* через поле *projectSnap* передает их в виджет *ListView*. Виджет *ListView* позволяет построить листаемый на экране пользователя массив виджетов.

Для управления каждым отдельным устройством был создан класс *DeviceListItem*, который принимает в себя объект устройства умного дома и отображает его виджет на главном экране. Это позволяет идентифицировать каждое отдельное устройство, так как виджет *ListView* лишь занимается созданием списка отображения.

Класс *DeviceListItem* содержит в себе методы для корректного отображения имени устройства, его типа и состояния. Для этого используется несколько виджетов: *Card* и *ListTile*.

Виджет *ListTile* содержит в себе несколько важных методов: *onTap*() и *onLongPress*(). Первый позволяет при нажатии на виджет запустить определенный метод. В мобильном приложении при нажатии на виджет устройства реализовано включение или выключение умного устройства. Метод *onTap*() запускает выполнение функции *switchStateOfDevice*() в классе *AuthAPI* и передает ему уникальный id устройства вместе с параметром *isOn*, отвечающий за состояние новое устройства.

При долгом нажатии на виджет *ListTile* запускается метод *onLongPress*(). Данный метод в зависимости от типа устройства переводит пользователя на страницу с детальной информацией об устройстве. Реализовано данное переключение между экранами через статический метод *Navigator*.*push*(). Например, для устройства Lightning Bulb 1 доступна настройка выбора цвета, как и для любой лампы.

На данном экране пользователь может выбирать цвета лампы и просмотреть информацию об устройстве. Выбор цвета реализован при помощи класса *BlockPicker,* который принимает доступные для выбора цвета и устанавливает методы для управления цветами. При нажатии на определённый цвет, на сервер отправляется команда с *RGB*-значениями цвета и *id* устройства, которому необходимо назначить выбранный цвет.

Помимо страницы с доступными устройствами пользователю доступен экран настроек, который содержит в себе несколько настроек:

* добавить устройство;
* добавить хаб;
* удалить устройство;
* удалить хаб.

Эти виджеты отображаются при помощи метода *returnCardSettings*(), созданный как шаблон для удобного создания параметров. В нем указывается, какая иконка будет отображаться слева от параметра, само название параметра и по нажатию на виджет будет совершен переход в желаемую настройку.

Добавление нового устройства в систему умного дома происходит путем нажатия пользователем в настройках пункта «*Add new device*». На данном экране представлено несколько полей, которые пользователь должен заполнить, чтобы добавить устройство в свой умный дом.

На данном экране предлагается ввести имя устройства, его местоположение, серийный номер, тип устройства, хаб, к которому устройство будет привязано и *IP*-адрес устройства в сети. Данные настройки вводятся при помощи виджетов *TextField*.

Для работы виджетов *TextField* необходим контроллер ввода текста *TextEditingController*. Он позволяет отслеживать набираемый пользователем текст и, когда пользователь хочет сохранить данные, данный контролер текста передает пользовательский текст в методы сохранения нового устройства.

Помимо текста, на данном экране присутствуют два выпадающих меню: для выбора типа устройства и хаба, к которому привязывается устройство. Для реализации использовались два класса: *DropdownButton* и *DropdownMenuItem*. Первый класс позволяет создать само выпадающее меню, настроить функции при нажатии на него и выбрать стили. Второй класс, *DropdownMenuItem*, можно назвать оберткой для данных, которые пользователь видит на экране. Класс *DropdownMenuItem* необходим для корректного отображения списка опций и содержит в себе два поля: *value* и *child*. В поле *value* передается один из выбираемых объектов, а в *child* – класс, который будет отвечать за корректное отображение на экране этого объекта. В данной работе в *value* передается тип возможного девайса как объект типа *String*, а в *child* устанавливается этот же объект, но в обертке *Text*() для корректного отображения.

Такой же подход используется и для выпадающего меню с выбором хаба, к которому будет привязано новое устройство. Класс *DropdownButton* отвечает за создание самого виджета выпадающего меню, а класс *DropdownMenuItem* – за каждый объект, который можно выбрать. В случае с выбором хаба, у класса *DropdownMenuItem* в поле *value* устанавливается объект типа *HubDTO*, который инкапсулирует в себе один из хабов, привязанных к пользователю, а в поле *child* – виджет *Text*, который отображает в меню выбора хаба название привязываемого хаба.

Чтобы иметь возможность выбрать тип устройства и хаб, к которому привязывается устройство, необходимо заранее получить список возможных для выбора вариантов. Чтобы реализовать данное требование, в классе *NewDevice*, который полностью отвечает за экран создания нового устройства, используется виджет *FutureBuilder*, который отвечает за построение виджетов в зависимости от асинхронного метода загрузки данных, то есть при статусе загрузки данных будут отображаться различные виджеты. Данный виджет уже был упомянут в данной работе, он используется в данном классе схожим образом.

Асинхронный метод \_*getListOfHubs*(), передаваемый в поле future класса *FutureBuilder*, при помощи статических методов *getListOfHubs*() и *getDeviceTypes*() класса *AuthAPI* запрашивает у сервера данные о хабах пользователя и типах девайсах, которые возможно добавить в системе умного дома. После получения данных, они сохраняются в полях *listOfHubs* и *listOfDeviceTypes*, представляющие из себя массивы. После получения данных, поле *hasData* объекта *snapshot* становится true, что говорит о полном получении данных с сервера. Виджет перестраивается, и массивы с выбираемыми типами устройств и хабов преобразовываются в необходимые для работы с *DropdownButton* классы *DropdownMenuItem*. Таким образом реализован выбор хаба и типа устройства.

Сохранение данных происходит при нажатии пользователем на кнопку сохранения в правом нижнем углу экрана. Этот виджет реализован при помощи класса *FloatingActionButton*. Данный класс передается в поле *floatingActionButton* виджета *Scaffold* в классе *NewDevice*. В нем возможно настроить отображаемую иконку путем передачи класса *Icon* в поле *child*, а также настроить метод *onPressed*(), который будет запускаться при нажатии на кнопку сохранения данных.

Сохранение данных происходит в методе *submitData*(), который запускается после нажатия на кнопку сохранения. В данном методе вся информация забирается из полей, отвечающих за каждую характеристику умного устройства, и отправляются на сервер через метод *addNewDevice*() класса *AuthAPI*. Таким образом реализован экран добавления нового умного устройства.

Следующим экраном настроек является экран добавления хаба. В мобильном приложении он именуется «*Add new Hub*».

За создание виджетов при добавлении хаба на экране пользователя отвечает класс *NewHub*.

На данном экране используется несколько важных виджетов: *FloatingActionButton* и *TextField*. Первый виджет, аналогично предыдущему экрану добавления устройства, создает кнопку сохранения данных в правом нижнем углу и добавляет метод *onTap*(), который запускает необходимые действия по нажатии на кнопку. Второй виджет, *TextField*, отвечает за набранный пользователем текст. Таких виджетов представлено три: для ввода имени хаба, уникального *UUID* хаба и секретного ключа. Их использование будет освещено далее в этой работе.

При нажатии на кнопку сохранения запускается метод *submitData*(), который собирает введённый пользователем текст из контроллеров набранного текста и передает функции *addHub*() класса *AuthAPI* для отправки на сервер.

Третьим экраном настроек является экран удаления устройства.

На данном экране пользователь может удалить устройство умного дома, выбрав его и нажав на кнопку удаления в правом нижнем углу. Данное действие реализовано при помощи виджетов *FutureBuilder*, *DropdownMenu* и *FloatingActionButton*.

В классе *DeleteDevice*, в котором используются вышеперечисленные виджеты, *FutureBuilder*, аналогично другим экранам, получает в поле future асинхронный метод \_*getDevicesList*(), который в свою очередь вызывает метод *getListOfDevices* класса *AuthAPI*. При получении данных отображается виджет *DropdownButton*, в котором массив устройств, инкапсулированных в классе *Device*, преобразуется в массив *DropdownMenuItem*, необходимым для корректной работы виджета *DropdownButton*. При выборе устройства объект класса *Device* записывается в переменную *selectedDevice*.

При нажатии на кнопку удаления в правом нижнем углу метод *onTap*  
(), расположенный в виджете *FloatingActionButton*, вызывает метод *submitData*(), который передает методу *deleteDevice* в классе *AuthAPI* уникальный *id* выбранного для удаления устройства умного дома.

После выполнения данных действий устройство удаляется и перестает отображаться на главном экране приложения и в списке устройств, доступных для удаления.

Четвертым по счету экраном настроек умного дома является удаление хаба, в настройках он отображается как «*Delete Hub*». На рисунке 19 приведен пример данного экрана.

На данном экране пользователь может удалить хаб, привязанный к его аккаунту. Данное действие реализовано виджетами *FutureBuilder*, *FloatingActionButton* и *DropdownButton*.

В классе *DeleteHub*, в котором используются вышеперечисленные виджеты, *FutureBuilder*, как и на других экранах, получает в поле *future* асинхронный метод \_*getListOfHubs*(), который в свою очередь вызывает метод *getListOfHubs* класса *AuthAPI*. При получении данных отображается виджет *DropdownButton*, в котором массив хабов, инкапсулированных в классе *HubDTO*, преобразуется в массив *DropdownMenuItem*, необходимым для корректной работы виджета *DropdownButton*. При выборе хаба объект класса *HubDTO* записывается в переменную *selectedHub*.

При нажатии на кнопку удаления в правом нижнем углу метод *onTap*  
(), расположенный в виджете *FloatingActionButton*, вызывает метод *submitData*(), который передает методу *deleteHub* в классе *AuthAPI* уникальный *UUID* выбранного для удаления хаба умного дома.

После выполнения данных действий хаб удаляется и перестает отображаться на главном экране приложения и в списке устройств, доступных для удаления.

Важным пунктом является поддержка защищенного соединения между сервером и клиентом. В мобильном приложении все данные передаются по защищенному протоколу *HTTPS*. Для его имплементации не требуются дополнительные настройки, фреймворк *Flutter* самостоятельно занимается проверкой сертификатов и иными транспортными вопросами, позволяя сосредоточиться только на работе с данными. Об имплементации протоколов шифрования будет рассказано позднее в данной работе.

Внешний вид пользовательских страниц достаточно понятен и прост. Приложение не перегружено функциями и исправно работает.

Таким образом, было реализовано мобильное приложение для управления умным домом и описано, как каждый компонент системы взаимодействует с пользователем. Все поставленные задачи были выполнены.

3.2 Разработка серверного приложения

В предыдущей главе было установлено, что серверная часть системы умного дома будет работать на языке программирования *Java* с использованием фреймворка *Spring*. Все данные будут сохраняться в СУБД *PostgreSQL*, а для связи между хабами будет использоваться *WebSocket*.

В данной части работы будет описана разработка системы, ее безопасность, принципы хранения данных.

Сервер должен принимать запросы и отправлять на них ответы. Важно идентифицировать отправителя запроса и знать, имеет ли он разрешение на совершение запрашиваемых действий. Для этого был настроен фреймворк *Spring Security*, который отвечает за безопасность системы.

Его настройка происходит путем создания бинов (это объекты и методы, которые управляются фреймворком *Spring*) в классах, помеченных аннотацией @*Configuration*.

Для эндпоинтов авторизации не требуется токены или другие подтверждения авторизации, что логично, так как именно через них пользователь может получить токены доступа. Все остальные эндпоинты требуют авторизации, и без ключа или токена ими не получится воспользоваться.

Помимо настроек безопасности эндпоинтов, в листинге указан метод *addFilterAfter*, в которй передается объект *authTokenFilter*. Данный объект класса *AuthTokenFilter* позволяет определить пользователя по токену, который был передан в заголовке запроса. Данный класс занимается расшифровкой токенов, проверкой логина и пароля, который был зашифрован, а также проверяет, является ли токен просроченным или невалидным. Если все условия проверки были пройдены, то *Spring Security* допускает запрос к дальнейшей работе. Если, например, у токена истек срок годности, то *Spring* запретит продолжение работы и вернет 403 код ответа, что означает «Доступ запрещен».

Создание токена происходит в классе *JWTUtils* в методе *createJWT*. Данный метод является перегруженным, что означает, что методы с одним названием принимают разные параметры. Перегрузка метода нужна для ссоздания токена для хаба и пользоваетелей.

Оба метода работают по одному принципу: для создания токена требуется время срока действия токена, ключ шифрования и два поля “*id*” и “*sub*”, отвечающих за логин и пароль соответственно. В данной системе токен действует один час, после чего его нужно получить заново, отправив новый запрос на эндпоинт авторизации.

Сам токен является строчкой, разделенной на три части через точки. Первая часть токена является заголовком, вторая – зашифрованными данными, а третья – подписью.

В заголовке содержится информация о типе токена и алгоритме шифрования, в системе умного дома используется тип *JSON* *Web* *Token* и шифрование *HMAC*-*SHA256*.

Зашифрованными данными являются поля *id* и sub, которые являются логином и паролем пользователя соответственно. Список полей может быть производным.

В третьей части, подписи, используются 2 предыдущих части токена, которые кодируются при помощи преобразования в строку типа *base64url* и «складываются» с секретным ключом, определенным в файле *application*.*yaml* проекта умного дома при помощи алгоритма *HMAC*-*SHA256*. Таким образом, подпись позволяет проверить, были ли модифицированы данные заголовка и блока данных.

После успешной аутентификации можно вызывать заранее определенные эндпоинты. В случае с системой умного дома, эндпоинты разбиты по группам обязанностей: эндпоинт авторизации, эндпоинт для управления умными устройствами, эндпоинт регистрации и эндпоинт управления хабами. Рассмотрим их подробнее.

Эндпоинт авторизации является классом *AuthController*, помеченный аннтоациями @*RestController*, @*RequestMapping*(“/*api*/*auth*”) и @*RequiredArgsConstructor*. Эти аннотации необходимы для определения *Spring* фреймворком, какой метод запустить при вызове эндпоинта. В классе *AuthController* написано два метода, *login*() и *loginHub*(), отвечающие за авторизацию пользователя и хаба, который хочет соединиться с сервером. Процесс работы у двух методов похожий: сервер принимает *JSON*-объекты, которые преобразовываются *Spring* фреймворком при помощи десериализации в *Java*-объекты: поля из *JSON* преобразовываются в поля *Java*-объектов. Эти объекты передаются в класс *AuthServiceImpl*, который в свою очередь передает данные на проверку в методах класса *AuthProvider*. Если проверка прошла успешно, то есть логин и пароль совпали с данными из базы данных, то на основе данных пользователя формируется токен. Данный токен передается в класс *AuthenticatedUserDTO* или *AuthenticatedHubDTO*, в зависимости от клиента, вызвавшего метод авторизации, и отправляется как ответ с кодом 200.

Класс для управления умными девайсами *DeviceController* обширней: в нем находятся методы, позволяющие:

* получить список всех устройств, которые привязаны к аккаунту пользователя;
* включать или выключать устройство;
* получить статус устройства;
* получить информацию об устройстве;
* сменить цвет устройства (в случае с *Wi-FI* лампами);
* добавить новое устройство;
* удалить устройство;
* получить список всех доступных типов устройств.

Все методы, определенные в этом классе, имеют аннотацию @*RequestMapping* вместе с уникальным *URL*, позволяющую привязать вызываемую «ссылку» к определенному методу. Также в этой аннотации обозначены типы запросов, которые требуются для данного метода. Например, чтобы получить список всех устройств, необходимо отправить *GET*-запрос на адрес сервера вместе с *URL* «/*api/devices/list*».

Говоря о запросах, нужно отметить, что *POST*, *GET*, *PUT* и *DELETE* – запросы могут возвращать ответы в виде объектов, однако использование отдельных методов позволяет лучше понять *API* системы. Например, тип запроса означает «получить» или «обновить», а *URL* – та информация, которую мы передаем или желаем получить. Для каждого действия должен быть свой эндпоинт со своим типом запроса – это принцип *REST*-архитектуры.

В классе эндпоинтов для управления умными устройствами инкапсулирован объект *service* класса *DeviceServiceImpl*. Он отвечает за бизнес-логику при вызове эндпоинтов: все данные, принятые в эндпоинтах, переходят в сервис.

Данный сервис позволяет выполнять логику, которую запрашивает пользователь. В нем описаны методы взаимодействия с базой данных и веб-сокетами, подключенными в данный момент к серверу.

Например, если пользователь отправит *PUT*-запрос по *URL* «/*api/devices/{deviceId}/state*» вместе с уникальным идентификатором умного устройства и булевой переменной *isON*, отвечающей за включение или выключение устройства, то в методе *switchDeviceState*() класса *DeviceServiceImpl* будет произведены следующие операции:

1. Из базы данных будет получена сущность умного устройства в виде объекта *DeviceEntity*. В нем содержится вся информация об устройстве, например его тип, *IP*-адрес, название и другое.
2. В этой сущности будет изменен параметр статистики *json\_properties*, в котором сохраняется новое значение *isOn* в виде *true* или *false*.
3. Составляется информационный объект, который будет отправлен по веб-сокету на хаб.
4. Этот объект отправляется на хаб.

В данном классе описана бизнес-логика для следующий действий:

* получить список всех устройств пользователя;
* удалить устройство;
* включить или выключит устройство;
* добавить устройство;
* получить информацию об одном устройстве;
* поменять цвет лампы.

Таким же образом устроены и другие эндпоинты и сервисы для работы системы умного дома: в первых декларативно указываются, какие характеристики и объекты принимает эндпоинт, а сервис занимается выполнением вызываемых функций. Таким образом, помимо вышеприведенного сервиса *DeviceServiceImpl*, реализованы эндпоинты и сервисы для взаимодействия с хабами и авторизацией пользователей.

Помимо взаимодействия объектов в системе, важно упомянуть, какие данные и с какими характеристиками сохраняются в базе данных. В базе данных хранится несколько стержневых сущностей: пользователь, хаб и устройство. Рассмотрим их подробней.

В таблице пользователь, названной «users», хранятся имя и фамилия пользователя, логин и пароль от его аккаунта и уникальный *id* пользователя.

Таблица устройств, именованная в базе данных «*devices*», хранит в себе много важной информации:

* *device*\_*id* – уникальный *id* отдельно взятого умного устройства;
* *device\_name* – имя устройства;
* *device\_location* – местоположение устройства;
* *device\_serial* – серийный номер устройства;
* *local\_ip\_address* – локальный *IP*-адрес устройства;
* *device\_owner* – ссылка на владельца устройства;
* *hub\_id* – ссылка на привязанный хаб;
* *device\_type* – тип устройства;
* *json\_properties* – тип данных *jsonb*, который содержит в себе информацию о состоянии устройства.

Поле device\_type заполняется текстовым значением из enum-класса *DeviceType*, который отвечает за поддержку выбора типа устройства. В данной системе пока реализован тип устройства «лампа», отмеченная как «*BULB*».

В поле *json\_properties* содержится тип данных *json,* который сохраняет последнее состояние устройства, например цвет лампы или ее яркость.

Третьей таблицей является таблица хабов, названная «*hubs*». В ней содержатся следующие поля:

* *hub\_id* – уникальный *id* отдельно взятого хаба;
* *hub\_owner* – ссылка на таблицу «*users*» для определения владельца хаба;
* *hub\_name* – название хаба;
* *hub\_uuid* – уникальный *uuid* для верификации хаба;
* *hub\_secret* – секретный ключ хаба;

Для связи базы данных с серверным приложением в данном проекте используется фреймворк *Hibernate* 6. Он поддерживает постоянное соединение с базой данных и позволяет при помощи аннотаций и *Java*-объектов выполнять *CRUD* запросы к базе данных и получать от нее ответы. Для этого в коде реализованы *Java*-объекты сущностей базы данных.

Например, для таблицы пользователей был реализован класс *UserEntity*.

Для работы Hibernate необходимо определить классы, которые будет содержать все те же поля, что и в таблице, при помощи аннотации @*Entity*. Также требуется создать специальные классы-репозитории, помеченные аннотациями @*Repository* для доступа к таблицам базы данных. Данные аннотации позволяют фреймворку оперировать полями объектов для полноценной работы с БД и осуществления запросов к ней, упрощая взаимодействие с ней.

При помощи аннотаций @*JoinColumn* и @*OneToOne* возможно использовать *join*-команды и получать другие сущности в виде объектов. Так было сделано для сущности пользователя и хаба. Для этого необходимо указать, какой столбец в желаемой сущности «маппится» на другой стоблец изначальной сущности.

Таким образом, при помощи технологии объектно-реляционного отображения удалось представить сущности в виде объектов и работать с ними как с обычными классами и полями.

Помимо функционала сохранения данных и методов их обработки, в системе умного дома необходимо реализовать способ постоянного подключения хабов к серверу. Для этого использовался набор зависимостей *Spring Boot Starter WebSocket.*

Данный компонент позволяет использовать веб-сокет для поддержки постоянного двунаправленного соединения между системами. Веб-сокет – это технология, позволяющая создать постоянное соединение между двумя системами и асинхронно передавать по нему данные. В отличие от *HTTP*, где при передаче данных каждый раз создается новое подключение, веб-сокеты передают данные по уже созданному соединению.

В данной работе веб-сокет соединение будет инициироваться хабом, то есть клиентом. Веб-сокет также позволит обойти *NAT*, то есть сервер не сможет самостоятельно возобновить подключение, пока клиент сам не создаст его. Также это соединение будет передавать данные по протоколу *STOMP*, созданный для передачи сообщений внутри протокола *WebSocket*.

Важно знать, как устанавливается сокетное соединение. Для этого клиент отправляет *HTTP*-запрос на сервер. Внутри заголовка (*header*) располагается поле «*Connection*» со значением «*upgrade*», а после поля «*Upgrade*» указывается протокол «*websocket*». После получения данного запроса сервер отправляет 101 ответ, что означает готовность сервера сменить протокол взаимодействия. Так устанавливается рукопожатие между клиентом и сервером и совершается переход от HTTP к Web Socket протоколу.

Программный компонент настройки веб-сокета расположен в серверной части в классе *WebSocketController*. Он помечен аннотацией @*Controller*, чтобы обозначить данный класс как принимающий данные. В нем расположено два метода: greeting() и *responseFromHub*(). Метод *greeting*() позволяет определить пользователя и отправить ему свой ответ об успешном подключении. Данный метод помечен аннотациями @*MessageMapping* и @*SendToUser*, определяющие, какой *URL* присвоен данному методу и позволяющий вставить уникальный *id* хаба для ответа только ему соответственно.

Второй метод, *responseFromHub*(), занимается получением данных от хаба и их обработкой. Он помечен аннотацией @*MessageMapping*, определяющий *URL* данного эндпоинта, и передает объекту service класса *SocketConnectionService* пришедшие данные.

Класс *SocketConnectionService* позволяет отправлять через сокетное соединение на хаб информацию и обрабатывать данные, полученные от хаба. Для этого используются два вспомогательных класса: *SimpUserRegistry* и *SimpMessagingTemplate*.

*SimpUserRegistry* – это класс, который следит за установленными соединениями. Через него можно узнать, подключен ли хаб к серверу и можно ли отправлять данные на заданный хаб. Проверка происходит через метод *getUser*(*String* *userName*), и если данный метод вернет *null*, то значит хаб не подключен и будет выброшено исключение, что хаб недоступен.

Класс *SimpMessagingRemplate* позволяет отправить сообщение через веб-сокет на хаб, указав имя хаба, эндпоинт доставки и класс, который будет нести в себе информационное сообщение. Для удобной передачи сообщений был создан класс *ActionDTO*, в котором установлено два поля: строка *action* и отображение *map*. В строку *action* записывается действие, которое хаб должен выполнить, а в *map* – параметры для данного действия. Например, если пользователь укажет в приложении, какой цвет лампы должен быть установлен, то в классе *ActionDTO* в поле *action* будет записано значение “*CHANGE*\_*COLOR*”, а в *map* – *IP*-адрес устройства и *RGB* – значения желаемого цвета. Эти данные затем будут отправлены пользователю через метод *convertAndSendToUser()* в классе *SocketConnectionService*.

Важно установить правильные настройки для работоспособности веб-сокета. Для этого в классе *WebSocketConfig* были прописаны настройки для регистрации веб-сокетов, указаны конкретные эндпоинты веб-сокетов (в данной работе ими являются «/*hello*» и «/*resp*»), добавлены методы для конвертации классов в *JSON*, настроен брокер сообщений. Также в качестве параметра был передан перехватчик рукопожатий класс *UserHandshakeHandler*. В этом классе определен метод *determineUser*, при помощи которого из *URL* извлекается уникальный *UUID* устройства и устанавливается в класс *UserPrincipal* для обеспечения работоспособности методов определения хаба в сети.

При работе системы возможны ошибки, и для их обработки был написан класс *CustomResponseEntityExceptionHandler*. В нем находятся специальные методы, которые позволяют обрабатывать исключения во время работы программы и отправлять соответствующие сообщения клиентам. Например, если пользователь отправил неправильный логин или пароль, то активируется метод *handleUnsuccessfulLogin*, в котором указывается сообщение ошибки, код статуса и возникшее исключение. Это позволяет корректно обрабатывать исключения и сообщать о них пользователям.

Помимо вышеприведенного исключения, реализованы обработчики исключений для ошибок, связанных с отсутствием подключения к хабу или ошибкам, связанным с базой данных.

Таким образом, была написана серверная часть умного дома, в которой были выполнены все поставленные задачи.

3.3 Разработка хаба умного дома

Хаб – это устройство, который располагается в домашней сети пользователя и связывается с сервером для маршрутизации запросов от сервера к устройствам. В данной работе хабом может выступать любое устройство, имеющее доступ в сеть. Достаточно загрузить исполняемый код на устройство и запустить его.

Хаб должен выполнять следующие задачи:

* инициировать подключение к серверу;
* самостоятельно аутентифицироваться;
* выполнять команды от сервера.

Для реализации требований используется пакет *Spring Boo Starter WebSockets*. Он позволяет подключиться к серверу и обновить соединение до веб-сокета.

Для подключения к серверу хабу требуется авторизироваться на сервере. Для этого в методе *authHub* класса *AuthRequest* отправляется запрос с уникальными *UUID* хаба и секретным ключом хаба. Эти данные для каждого хаба уникальные и не могут повторяться. Также хаб получит токен только в том случае, если он был добавлен пользователем в систему через мобильное приложение.

Стоит отметить, что передача данных происходит по защищенному каналу. В данной работе связь между системами защищена посредством *TLS*-протокола.

После получения токена хаб отправляет запрос на создание сокетного подключения. Для этого на эндпоинт «/*hello*» отправляется запрос вместе с токеном и *uuid* хаба в *URL* запроса. Сервер проверяет токен и меняет протокол в случае успешной проверки данных.

Эндпоинтов у хаба нет. Вместо них создан класс *SocketStompHandler*, унаследованный от класса *StompSessionHandlerAdapter*. В данном классе есть пять методов: *afterConnected*(), *handleFrame*(), *getPayloadType*(), *handleException*() и *handleTransportError*().

Данные методы запускаются при определенных событиях веб-сокета. Первый метод, *afterConnected*(), позволяет выполнить логику после подключения к сокету. Второй, *handleFrame*(), позволяет получить данные, переданные от сервера, и обработать их. Третий по счету метод, *getPayloadType*(), позволяет указать, какой объект будет использоваться в качестве передаваемого по сети. В методе *handleException*() указывается, что делать при возникновении исключения. В *handleTransportException*() указывается, что делать при возникновении ошибки, связанной с транспортом, то есть сокетом.

Главным методом в этом классе можно назвать метод *handleFrame*(). В нем, при получении данных, происходит конвертация объекта *Object* в *Action*, полностью совпадающий с классом *ActionDTO*, отправляемый сервером. В метод *handleAction* класса *StompService* передается действие, которое требуется выполнить хабу. Например, если в поле action находится «*TURN\_ON*», то запустится метод *turnOnLights*(), который отправит запрос на IP-адрес, принадлежащий умной лампе.

Метод для запуска требуемой функции определен в *switch* блоке метода *handleAction*. В нем перечислены методы, которые должны запускаться при определенном значении поля action класса *Action*.

Отправкой запросов на устройство занимается метод *sendCommand*. В нем используется класс *RestTemplate*, используемый для отправки *REST*-запросов на устройства и получение от них ответов. При необходимости хаб может передать обратно серверу по *WebSocket* данные от умного устройства.

Для работы хаба не требуется вмешательство пользователя. Он полностью автономен благодаря *Spring* *Framework*, который автоматически запускается и самоконфигурируется.

Таким образом, был реализован хаб для системы умного дома, который позволяет отправлять команды от сервера и обмениваться сообщениями с ним.

3.4. Добавление шифрования и деплой системы

Для того, чтобы система умного дома работала по сети Интернет, необходимо ее запустить на компьютере, имеющий публичный *IP*-адрес. Связано это с тем, что публичные *IP*-адреса маршрутизируются в Интернете и доступны без *DDNS* сервисов, что упрощает развертывание приложения.

Для работы сервиса в сети Интернет был выбран *VDS* – провайдер «*VDSina»*, предлагающий услуги аренды частного сервера. В нем был создан сервер на базе *Ubuntu* 20.04, на котором будет работать база данных и серверная часть.

Также в данном сервисе был оформлен новый домен *smarthome-controls.ru* и настроен на перенаправление запросов на *IP*-адрес сервера.

Серверная часть и база данных будут контейнерезированы в *Docker* для удобного управления сервисами.

Для запуска системы умного дома был создан файл *Dockerfile* в котором содержится информация о параметрах запуска контейнера серверной части умного дома, *SQL*-код для инициализации базы данных, контейнер *certbot* для создания сертификата шифрования, контейнер nginx для проксирования и файл *docker-compose.yml*. Рассмотрю эти файлы подробнее и опишу их значение.

В *Dockerfile* для запуска *Java*-кода серверной части системы умного дома прописаны параметры, по которым будет создаваться *jar*-файл с кодом. Для сборки этого файла используется *Maven –* инструмент для автоматизации сборки программы.

В данном файле описано, что при помощи пакета *maven:3.8.3-openjdk-17* будет произведена сборка *jar*-файла с профилем prod. В данном профиле указаны данные для подключения *Hibernate* к базе данных. После сборки *jar*-файла он копируется в корневую папку с названием *app*.*jar* и запускается, попутно делая доступным порт 8082 контейнера.

Для инициализации базы данных необходим *SQL*-код, с помощью которого будут созданы таблицы и внесены первоначальные данные. Для этого в файле *init*.*sql*, в котором хранится код для инициализации базы данных с названием “*SmarthomeDB*”, содержатся методы для создания таблицы при помощи процедур «*CREATE TABLE*». Сущности, которые используются в системе, были рассмотрены ранее в данной работе. Данный файл копируется в контейнер *PostgreSQL* и исполняется в нем.

Для обеспечения безопасности передаваемых данных в данной работе используется бесплатный сертификат безопасности «*Let’s Encrypt*». Данный сертификат позволяет верифицировать сервер и шифровать данные для их безопасной передачи по сети. Для получения сертификата используется контейнер *certbot*, с его помощью можно получить данный *SSL/TLS* сертификат. Это - архиважная часть системы, которая защищает данные от несанкционированного чтения.

Стоит отметить, что данный сертификат нужно обновлять раз в 3 месяца, но при помощи скриптов можно автоматизировать этот процесс. Чтобы получить сертификат и начать использовать его, необходимо указать несколько параметров, при помощи которых будет выдан сертификат:

* --*webroot* – плагин для автоматической настройки сертификата
* --*webroot*-*path* – путь для создания сертификатов
* --*email* – почтовый адрес владельца
* --*agree*-*tos* – соглашение с условиями использования
* --*no*-*eff*-*email* – не соглашаться на взаимодействие с *EFF* (*Electronic Frontier Foundation*).
* --*force-renewal* – выпустить новый сертификат такому же домену, что и у существующего сертификата
* -*d* – указание адреса, которому выдается сертификат.

Данная команда прописывается в блоке инициализации в файле *docker-compose.yml.*

Выданный сертификат необходимо применить к системе. Для этого будет использоваться *nginx*, который будет проксировать запросы к локальному *Java*-контейнеру и обеспечивать работу системы шифрования данных. Для этого напишу конфигурационный файл.

В нем указано два *location* – для *http*s и *wss* запросов (шифрованный веб-сокет).

В блоке *server* указано, что *nginx* будет слушать только 443 порт, а среди поддерживаемых протоколов будут использоваться только *TLSv1*, *TLSv1*.1, *TLSv1*.2 и *TLSv1*.3. В поле ssl\_*certificate* и ssl\_*certificate*\_key установлены пути, по которым находятся ранее выданные сертификаты от «*Let’s* *Encrypt*».

В поле ssl\_dhparam необходимо вставить ключ, необходимый для работы сертификата шифрования. Для этого он генерируется командой «*openssl dhparam -out dhparam-2048.pem 2048*». После этого путь до данного ключа необходимо указать в настойке ssl\_*dhparam* в *nginx*.

В том же конфигурационном файле для nginx указываются настройки проксирования и параметры заголовков.

Для защиты сервера от несанкционированных атак были закрыты все порты, кроме 443 и 22.

Для старта всех компонентов системы достаточно написать в терминале *docker*-*compose* up. После загрузки всех компонентов система заработает и будет доступно *TLS* шифрование.

Для эксперимента проведу тестирование шифрования при помощи сервиса *ssllabs*.*com*. Для проверки достаточно указать адрес системы, после чего начнется тестирование. По результату тестирования была выставлена оценка «*А*», то есть «отлично».

Таким образом, был рассмотрен вопрос безопасности передаваемых данных, реализовано шифрование и запущен сервис умного дома на удаленной машине. Все поставленные задачи были выполнены.

# **4 ВОЗМОЖНОСТИ РАСШИРЕНИЯ УМНОГО ДОМА**

Система умного дома может автоматизировать практически любой процесс или добавить функциональности привычным вещам. Например, можно создавать автоматизации при включении умного прибора или настраивать устройства через программируемые пользователем функции.

Возможные программные функции

Среди возможных программных нововведений можно добавить возможность делиться доступом к устройствам умного дома. Это добавит удобства в управлении системой умного дома, так как позволит пользователю давать доступ доверенными людям без необходимости предоставления своих данных для входа в систему и управления ею.

Чтобы реализовать данный функционал, необходимо разработать логику предоставления доступа к устройствам другим людям.

Все управление умным домом происходит в приложении пользователя. Это значит, что необходимо добавить новый пункт настроек в пользовательском интерфейсе. В техническом смысле, это можно описать следующим образом.

В проекте мобильного приложения список настроек находится в классе *SettingsPage*. Его можно модифицировать, добавив в список возвращаемых виджетов метод *returnSettingCart*(). В него передам имя будущего виджета «*Manage* *device* *sharing*» и класс *DeviceSharing*, который будет отвечать за работу виджета открытия доступа к устройствам другим людям.

Виджет *DeviceSharing* будет содержать два поля: поле выбора устройства, к которому владелец может открыть доступ и логин пользователя, которому будет открыто управление устройством. По нажатию кнопки сохранения эти данные придут на эндпоинт серверной части приложения, который передаст данную информацию сервису по управлению доступами к устройствам.

Для того, чтобы сохранить сведения о доступах пользователей к устройствам, необходимо создать отдельную таблицу в базе данных. В ней должно содержаться несколько полей: ссылка на владельца устройства, ссылка на само устройство, к которому открыт доступ и ссылка на человека, который имеет доступ к устройству.

Чтобы использовать данный функционал, в методе *returnDevicesList*() класса *DeviceServiceImpl*, который возвращает список устройств, доступных к управлению, можно добавить метод, задача которого будет добавлять в вышеприведенный список устройства, доступ к которым открыт пользователю, и отправлять пользователю. В самом приложении подобное устройство будет помечено как чужое, но доступ к нему будет полным.

Чтобы закрыть доступ к устройству, владельцу достаточно будет в меню настроек нажать на кнопку «Закрыть доступ к устройству». Запустится метод, который удалит строку из новой таблицы. Таким образом, доступ к устройству будет прекращен.

Добавление поддерживаемых устройств

Широкий спектр поддерживаемых устройств – важный аспект при выборе системы умного дома. Для собственного решения умного дома можно рассмотреть существующие умные устройства, которые возможно встроить в собственную систему.

Среди таких устройств можно отметить релейный модуль от компании *Sonoff*. Он поддерживает подключение по *WI-FI* и позволяет использовать его через *REST*-запросы, по такому же принципу, как и *Wi-Fi* лампа.

Его можно использовать как переключатель для любых устройств. Для этого необходимо установить его в провод питания, подключить к *Wi*-*Fi* и настроить.

Для адаптации данного реле к системе не требуется дополнительные правки: *API*-интерфейс данного устройства уже поддерживается системой умного дома, так как совпадает с *API* умной лампы.

Для данной реализации умного дома подойдет любое устройство, которое работает на протоколе *Wi-Fi* и имеет *REST*-интерфейс для взаимодействия с ним. Можно также самостоятельно сделать собственное устройство при наличии навыков пайки.

Добавление новых протоколов

Как было выяснено ранее, умные устройства взаимодействуют по совершенно разным протоколам. Логичным шагом для расширения системы является добавление поддержки новых протоколов для расширения списка поддерживаемых устройств.

Существуют библиотеки, которые добавляют поддержку протокола *ZigBee* в *Java*-приложение. Такой библиотекой является «*zigbee4java*». При помощи данной библиотеки возможна передача данных на *ZigBee*-устройства с хаба. Для работы этой библиотеки необходим специальный передатчик, который подключается к хабу по *USB*, и имплементация кода, который будет взаимодействовать с устройствами.

Таким образом, можно добавить новый протокол общения с умными устройствами.

# **5 ТЕХНИКО-ЭКОНОМИЧЕСКИЕ РАСЧЕТЫ**

Расчет стоимости разработки программного обеспечения является важным этапом в процессе его создания. Этот процесс позволяет определить, сколько ресурсов, времени и денег потребуется для создания ПО, а также оценить его экономическую эффективность. Оценка стоимости разработки программного продукта зависит от многих факторов, таких как сложность проекта, объем работ, сроки выполнения, квалификация специалистов и т.д.

Для определения стоимости разработанной системы необходимо посчитать затраченное на разработку время, определить квалификацию программиста для расчета стоимости его работы и посчитать траты на работу системы. Также важно получить прибыль с разработанной системы.

Для данной работы общие времязатраты составили 5 месяцев. В данное время включены все этапы создания системы: разработка архитектуры, выбор устройств для умного дома, написание серверного, клиентского приложений и хаба, проверка работоспособности и развертывание приложения на сервере. Для ускорения процесса разработки возможен найм нескольких разработчиков.

Для расчета заработной платы программиста необходимо принять во внимание его квалификацию и опыт работы. У автора имеется годовалый опыт работы, в который входили написание кода в существующей системе согласно техническому заданию, работа с базами данных, проверка написанного кода и написание тестовых наборов, работа с системами версионного контроля и анализ производительности кода. Данный специалист, согласно профстандарту Министерства труда и социальной защиты Российской Федерации, имеет уровень квалификации 6 [10] и, согласно анализу зарплат на ресурсе Habr [11], он может получать заработную плату в размере от 80.000 до 120.000 рублей в месяц. Возьмем нижний порог. Таким образом, за 5 месяцев работы разработчику необходимо заплатить 400.000 Рублей.

Также необходимо купить беспроводные лампы, которые будут управляться системой. Одна лампа стоит 1081 рубль.

Важным аспектом является подсчет трат на работу сервера. В данной конфигурации сервер стоит 6.6 рублей в день, 198 рублей в месяц и соответственно 2376 рублей в год. Данную сумму также необходимо заложить в стоимость продукта.

ПО, используемое для работы системы и ее разработки в данном проекте, было получено бесплатно, поэтому включать его в стоимость не целесообразно.

Таким образом, стоимость разработки составила 402061 рублей. В эту сумму входит 5 месяцев оплаты хостинга, зарплата программиста и одна беспроводная лампа.

Таблица 1 – Стоимость разработки системы умного дома

|  |  |
| --- | --- |
| Наименование | Количество |
| Затраченное время | 5 мес |
| Зарплата разработчика | 80.000 \* 5 = 400.000 рублей |
| Лампа Sonoff | 1081 за 1 шт. |
| Аренда хостинга (5 месяцев) | 6.6 \* 30 \* 5 = 980 рублей |
| Итого | 402061 рублей |

ЗАКЛЮЧЕНИЕ

По завершению выполнения данной выпускной квалификационной работы можно сделать следующие выводы.

В первой главе была рассмотрена концепция умного дома и проанализирована актуальность данной темы. Были рассмотрены популярные производители умных устройств на рынке СНГ и проанализированы методы передачи данных между устройствами.

Во второй главе была разработана архитектура системы умного дома. Были выбраны фреймворки и языки программирования вместе с пояснениями о сделанном выборе. Также был рассмотрен вопрос безопасности передаваемых по сети Интернет данных.

В третьей главе был описан процесс разработки системы умного дома на базе выбранной архитектуры. Были описаны методы имплементации каждого компонента системы.

В четвертой главе был рассмотрен вопрос о расширении возможностей умного дома и создана карта развития системы.

В пятой главе был проведен экономический расчет стоимости разработки системы умного дома.

Таким образом, поставленные цели и задачи в данной работе были достигнуты. Реализованная система работает отзывчиво, мобильное приложение ясно показывает состояние устройств в сети и управляет ими, а хаб бесперебойно выполняет свои функции. Результаты работы отвечают поставленным задачам.