



Московский государственный университет имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Кафедра автоматизации систем вычислительных комплексов

Князев Егор Иванович

**Разработка серверной части приложения
интернета вещей на основе принципов передачи
репрезентативного состояния**

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:

к.ф.-м.н.

А.Г. Бахмуров

Научный консультант:

А.А. Любицкий

Москва, 2023

Аннотация

В рамках данной выпускной квалификационной работы выполнено исследование существующих моделей взаимодействия клиентских приложений с сервером. Проведено сравнение различных подходов к организации интерфейса, предоставляемого сервером, а также для серверной составляющей системы сбора телеметрии Интернета Медицинских Вещей был разработан RESTful интерфейс, реализующий переход от нулевого уровня по модели зрелости Ричардсона, ко второму уровню. Также в качестве экспериментального исследования полученного интерфейса был разработан клиент импорта данных из облака производителя умных жилетов Hexoskin.

Оглавление

Аннотация	2
Оглавление	3
Введение	4
Основные понятия	4
Формулирование проблемы	4
1 Постановка задачи	6
2 Обзор подходов к построению архитектуры	7
2.1 REST	7
2.2 RPC	9
2.3 GraphQL	10
2.4 Сравнение подходов	10
3 Проектирование интерфейса	16
3.1 Ресурсы	16
3.2 Методы	17
3.3 Регистрация	18
3.4 Аутентификация	19
4 План перехода на RESTful API	20
4.1 Нулевой уровень зрелости по Ричардсону	20
4.2 Первый уровень зрелости по Ричардсону	21
4.3 Второй уровень зрелости по Ричардсону	21
4.4 Третий уровень зрелости по Ричардсону	21
5 Описание реализации	23
5.1 MQTT	24
5.2 HTTP	24
5.3 rest_api	24
6 Экспериментальное исследование	27
Заключение	28
Список литературы	29

Введение

В данной работе приведено исследование существующих подходов к унификации прикладных интерфейсов, а также разработан альтернативный интерфейс для системы Интернета Медицинских Вещей (Internet of Medical Things), называемой далее IoMT. Основной целью серверной части этой системы является сбор, хранение и обработка специфичных для медицины данных.

Виды учётных записей пользователей системы

Система поддерживает два типа аккаунтов: “операторы” и “пользователи”.

“Оператор” — тип аккаунта предназначенный для медицинских сотрудников имеющих доступ к portalу, предоставляемому веб интерфейсом, на котором те могут получить данные собранные с носимых устройств пользователей.

“Администратор” — оператор, имеющий расширенный набор прав, для администрирования системы. Под администрированием системы подразумевается создание, удаление и редактирование других операторов и типов устройств.

“Пользователь” — тип аккаунта предназначенный для людей использующих носимые устройства для сбора данных, передающий эти данные на сервер.

Актуальность решаемой задачи

При анализе наработок полученных из курсовой работы Князева Егора по развитию данного сервиса и выпускной квалификационной работы Юлии Аникевич было выявлено многократное дублирование кода, отсутствие систематизации и, как следствие, критическое усложнение, вплоть до невозможности, разработки иных клиентских приложений, в дополнение к

существующему веб-интерфейсу операторов и мобильному приложению сбора данных.

Фактически, кодовая база делилась на две части , для веб интерфейса операторов и для мобильного приложения пользователей. Анализ исходного кода показал, что в некоторых случаях одна и та же функциональность описывалась в двух разных местах, причем такое дублирование кода не получается устранить, оформив код в виде функции. Изучение имеющегося исходного кода показало, что написание нового клиента (в частности, для доступа к данным, для администрирования пользователей, резервного копирования) может вызвать затруднения.

В работе предпринята попытка унификации всех программных интерфейсов между компонентами системы. Проведён анализ известных архитектурных стилей построения программных интерфейсов, обоснован выбор стиля REST. Переработана архитектура системы в соответствии с принципами REST. Для демонстрации результативности разработан дополнительный клиент импорта данных из облака производителя умных жилетов Hexoskin.

1 Постановка задачи

В данной работе поставлена цель унифицировать существующий интерфейс предоставляемый сервером для реализации клиентских приложений для IoT платформы для достижения логической связности существующих моделей и методов, а также повышения надежности системы и упрощения разработки клиентских приложений и сервисов. Под унификацией в данном контексте подразумевается применение одного из рассмотренных архитектурных стилей для организации клиент-серверного взаимодействия.

Для достижения поставленной цели должен быть выполнен следующий список задач:

1. Исследовать существующие архитектурные стили.
2. Выделить основные достоинства и недостатки каждого подхода. Сравнить их между собой.
3. Выбрать архитектурный стиль для унификации интерфейсов и обосновать сделанный выбор.
4. Спроектировать новый интерфейс сервера, следуя принципам заложенным в выбранном архитектурном стиле.
5. Построить план разработки нового интерфейса и перехода существующих клиентов на новый интерфейс.
6. Провести экспериментальное исследование.
7. Сделать выводы.

2 Обзор подходов к построению архитектуры

На текущий момент в данной сфере наблюдается развитие трех основных подходов, это REST, RPC и GraphQL.

2.1 REST

Передача репрезентативного состояния (Representational State Transfer, REST[7]) — самый популярный из архитектурных стилей при построении интерфейсов приложений. Впервые данный термин был предложен Роем Филдингом в диссертации «Архитектурные стили и дизайн сетевых программных архитектур». Суть данного подхода заключается в передаче серверу исчерпывающих данных о запрашиваемом ресурсе и отсутствии хранимого на сервере какого либо состояния клиента.

В основе REST лежит 6 принципов, описывающих то, как должна быть спроектирована архитектура сервиса:

1. Модель взаимодействия основана на клиент-серверной архитектуре.

Одно из базовых ограничений накладываемых подходом REST, заключается в разграничении клиентских и серверных приложений. Данное ограничение, как и большая часть принципов REST, лежит в основе протокола HTTP и автоматически выполняется при использовании такового.

2. Отсутствие хранимого состояния клиента.

Одно из ключевых отличий сервисов, построенных по принципам REST и без учета данных принципов, заключается в отсутствии хранимого на сервере состояния клиента.

3. Кэширование

Принцип, способный увеличить производительность сервисов в разы, призван вместе с полезной нагрузкой в ответе передавать информацию о возможности кэширования этого ответа. Конечно это не единственный способ кэширования, также существуют различные варианты кэширование ответов на промежуточных узлах между

клиентом и сервером. Но именно сервер вправе определять возможность кэширования данного ответа.

4. Унифицированный интерфейс

Принцип определяющий единообразие представляемого интерфейса. Под единообразием понимается структурированность и логичность существующих методов, путей, параметров и т.д. И вновь, протокол HTTP представляет удобные инструменты для удовлетворения этого принципа.

5. Многослойность подключения

Под многослойностью понимается возможность размещения посредников между клиентом и сервером. Этот принцип позволяет реализовывать надстройки над сервером, позволяющие масштабировать сервис, устанавливать балансировку нагрузки, реализовывать кэширование на промежуточных серверах, а также дополнительно разгружать сервер, перекладывая простые проверки на посредников.

6. Код по требованию

Единственный опциональный принцип, предлагает использовать такой способ расширения функционала как отправка интерпретируемого кода по запросу клиента. Данный метод позволяет существенно расширить возможности сервиса, не обязывая клиенты поддерживать каждое последующее серверное обновление.

Закладывая перечисленные принципы в основу сервиса, REST позволяет сервису достичь следующих свойств:

- Производительность

Как уже было сказано, многие принципы в своей основе содержат установку на повышение производительности.

- Масштабируемость

Вопрос масштабируемости уже поднимался в работе Антона Бодрова в рамках автоматической масштабируемости части сервиса [5], работающей по протоколу MQTT. Архитектуры построенные по принципам REST позволяют реализовать аналогичные возможности и с HTTP составляющей проекта.

- Надежность

RESTful сервисы получают дополнительно к производительности повышенную надежность, обеспечивающую устойчивость к повышенным нагрузкам.

2.2 RPC

Удаленный вызов процедур (Remote Procedure Call, RPC[11]) — ещё один вектор развития архитектуры, предложенный в 60-70-х годах двадцатого века. Если в REST клиент и сервер обменивались состояниями объектов, то в понятиях RPC обращение клиента к серверу рассматривается как вызов функции, с передачей параметров и получением значения. В отличие от REST, где диктуются принципы построения сервисной архитектуры и понятие “реализация” попросту не применимо, RPC объединяет в себе набор технологий, выстроенных по диктуемому этим подходом стилю. Примерами таких технологий могут служить gRPC, DCE/RPC, XML RPC и другие.

В силу специфики RPC не рекомендуется строить реализации RPC поверх протокола HTTP. Хотя есть такие технологии как JSON-RPC, которые используют HTTP как базу для реализации вызовов RPC методов.

RPC стремится размыть границу между вызовом локальных процедур на клиентской стороне и вызовом удаленных процедур на серверной стороне. В связи с этим, характерными чертами взаимодействия по RPC является асимметричность, здесь наблюдается схожесть с принципом

клиент-серверной архитектуры для REST, и синхронность по аналогии с вызовом локальных функций.

2.3 GraphQL

В последнее время набирает популярность такой способ организации программных интерфейсов как GraphQL [12]. В действительности GraphQL — язык запроса данных, который в свою очередь навязывает свой, специфический, подход к построению интерфейсов. Таким образом GraphQL предлагает простейшую реализацию интерфейса из одного метода, принимающего запрос данных, составленный в заданном синтаксисе и возвращающий всю запрашиваемую информацию в едином ответе сервера.

2.4 Сравнение подходов

В силу различной природы сравнение этих подходов является весьма нетривиальной задачей, тем не менее возможно выделить несколько характеристик.

- Гибкость разрабатываемой системы;
- Простота разработки нового функционала;
- Простота использования клиентскими приложениями;
- Надежность;
- Производительность;
- Простота перехода с исходной схемы на новую.

Рассмотрим сильные и слабые стороны каждого подхода по каждой из приведенных характеристик. Для сравнения с исходным вариантом без какого либо подхода введем четвертый подход под названием “Без унификации” и так же оценим его по перечисленным характеристикам.

2.4.1 REST

- Гибкость данного подхода уступает варианту без использования каких либо ограничений и принципов. Однако данный архитектурный стиль

остаётся весьма гибким, так как диктует только принципы которым сервис должен соответствовать, не давая жестких реализаций.

- Так как кроме базовых знаний работы сетевых протоколов, языка и библиотек программисту необходимо знать и учитывать принципы REST, что иногда усложняет разработку.
- Разрабатывать клиентские приложения опираясь на принципы REST проще и быстрее чем без их соблюдения.
- Переход с исходной системы на новую заключается в смене клиентами конечных точек и формат передаваемых и принимаемых данных, но протокол запросов остаётся прежним - HTTP.
- Производительность системы является одним из основных преимуществ сервисов построенных по принципам REST, так как многие принципы ориентированы на увеличение пропускной способности и скорости ответа сервиса.
- Сервисы построенные по принципам REST можно считать более надёжными, нежели системы разработанные без применения данного подхода, так как такие принципы REST отсутствие хранимого на сервере состояния клиента, дополняют устойчивость сервиса к возможным поломкам и последствиям при поломках.

2.4.2 RPC

- Гибкость системы построенной по принципам RPC ниже чем по другим принципам из-за жестких ограничений реализаций этого протокола и отстранение программиста от сетевого взаимодействия, что может привести не только к различного рода ограничениям, но и к ухудшению производительности.
- Новый серверный функционал для архитектур с интерфейсом построенным на базе RPC разрабатывать проще при знании

соответствующей технологии реализующей этот подход. Так как он заключается в реализации обычной функции.

- Аналогично предыдущему пункту, при знании соответствующего фреймворка интеграция серверных вызовов в код происходит почти незаметно.
- Переход старых клиентов на взаимодействие через RPC процесс весьма не тривиальный из-за отказа от стандартного использования протокола HTTP и необходимости использовать соответствующую реализацию выбранной технологии.
- Производительность серверной части при взаимодействии методом RPC можно считать средней относительно остальных подходов, однако клиентская часть вносит дополнительные расходы.
- Взаимодействие клиента и сервера методом вызова удаленных процедур не предполагает дополнительных указаний для повышения надежности системы.

2.4.3 GraphQL

- Гибкость архитектуры построенной на базе GraphQL стоит под вопросом, так как диктуемый этим языком стиль интерфейса может привести к нежелательным последствиям. Также, по сравнению с REST, этот подход явно использует не все возможности предоставляемые протоколом HTTP на базе которого он работает. Что также можно учесть как недостаточную гибкость.
- Новый функционал на серверной стороне по сложности разработки сопоставим с REST. Так как прослеживаются аналогичные структуры и шаги при обработке GraphQL запроса.
- При использовании GraphQL в качестве языка запросов от клиента серверу. У клиента появляется возможность запросить большое

количество различных данных одним запросом в достаточно простой форме.

- Переход на GraphQL это сложный и долгий процесс затрагивающий как клиентские так и серверные части сервиса.
- Производительность системы основанной на подходе с использованием GraphQL можно считать повышенной из-за возможности запрашивать в одном обращении к серверу несколько ресурсов, что позволяет получать одинаковое количество информации через меньшее количество запросов по сравнению с подходами без использования GraphQL.
- GraphQL не предполагает дополнительных механизмов предполагающих повышение надежности системы.

2.4.4 Без унификации

- Гибкость архитектуры - одна из сильных сторон сервиса реализованного без специальных подходов к реализации интерфейса. Здесь ничего не ограничивает программиста и есть возможность взаимодействовать самыми разными способами.
- Простота разработки нового функционала на сервере также является преимуществом у данного подхода. Реализовать необходимую функцию не оглядываясь на общий стиль интерфейса гораздо проще нежели задумываться над тем как лучше написать тот или иной метод.
- Разработка клиентских приложений при подходе без унификации терпит существенные сложности с взаимодействием с сервером из-за отсутствия четкой системы и правил вызовов.
- При выборе варианта без унификации необходимость в переходе на новый архитектурный стиль пропадает.
- Система построенная без использования каких-либо подходов по унификации интерфейсов не повышает производительность сервиса.

- Надежность - слабое место отсутствия унификации интерфейсов. При таком подходе, велика вероятность дублирования кода, что ведет к неминуемым ошибкам и проблемам безопасности.

Приведенный разбор описывает сильные и слабые стороны каждого подхода, при этом заметно преимущество REST перед другими подходами, что позволяет выбрать его в качестве архитектурного стиля для нового интерфейса.

3 Проектирование интерфейса

Следующий этап достижения поставленной цели - это проектирование предполагаемого интерфейса. Для этого выделим основные ресурсы с которыми предполагается взаимодействие.

3.1 Ресурсы

Ресурсы — это объекты с которыми взаимодействует клиент. Опишем существующие ресурсы.

3.1.1 User

User — пользователь сервиса, предоставляющий информацию с носимых устройств. Имеет такие характеристики:

- id - автогенерируемый идентификатор
- login - уникальный логин пользователя, по которому он способен аутентифицироваться
- password - пароль пользователя, хранится в базе данных как два поля, хеш и соль используемая при хешировании
- name - имя
- surname - фамилия
- patronymic - отчество
- born - дата рождения
- email - подтвержденная электронная почта
- allowed - список операторов кому выдан доступ на просмотр данных пользователя

3.1.2 Operator

Operator — оператор, имеет доступ к веб portalу с данными пользователей.

- id - автогенерируемый идентификатор

- login - уникальный логин оператора по которому он способен аутентифицироваться
- password - пароль оператора, хранится в базе данных как два поля, хеш и соль используемая при хешировании
- is_admin - флаг указывающий на доступ к администрированию сервиса Authentication — конкретная аутентификация пользователя.
- Принимает либо JWT токен, либо пару логин-пароль.

3.1.3 Device

Device — определенный тип устройства.

- id - автогенерируемый идентификатор
- name - название девайса
- type - тип девайса (jacket или bracelet)
- characteristics - список характеристик передаваемых датчиком

3.1.4 UserDevice

UserDevice — конкретное устройство, зарегистрированное пользователем.

- id - автогенерируемый идентификатор
- user_id - привязка к конкретному пользователю
- device_id - привязка к типу датчика
- mac - MAC адрес устройства

3.2 Методы

В основе REST прочно закреплен протокол HTTP, соответственно методы из этого протокола активно используются в данном API.

GET - метод, выдающий информацию о запрашиваемом объекте, если таковая имеется и у пользователя есть доступ к этой информации.

POST - метод, создающий новый объект. Вызывается исключительно для создания новых объектов, а не для изменения старых.

PUT - метод редактирования уже имеющихся объектов.

DELETE - метод удаления объекта.

Благодаря такому четкому определению допустимых ресурсов и доступных для них методов, стало возможно представить унифицированный запрос к серверу в следующем виде:

Метод вызова /тип_запрашиваемого_объекта/{его_id}
{тело вызова, если необходимо}

Отдельно разберем ресурс Authentication и метод POST ресурса User.

3.3 Регистрация

Фактически метод POST ресурса User реализует регистрацию пользователя. И данный метод при реализации поставил интересную задачу: необходимо зарегистрировать пользователя с подтверждением электронной почты.

Ранее данный процесс был реализован через хранение информации о состоянии регистрации пользователя. Сейчас же принципы REST не позволяют реализовать подобный функционал через хранение состояния. В связи с чем процесс регистрации изменился. Теперь вся информация, введенная пользователем, кодируется в токен и отправляется ссылкой на указанный пользователем почтовый ящик. После чего пользователь переходит по указанной ссылке и вместе с этой ссылкой передает еще раз всю введенную им информацию, но в зашифрованном виде. Это финальный шаг регистрации. С получением этого запроса сервер действительно создает пользователя и сохраняет в базе данных.

3.4 Аутентификация

Все конечные точки сервера кроме регистрации доступны исключительно при наличии в запросе JWT токена в заголовках. Получение этого токена и называется аутентификацией.

Процесс аутентификации заключается в создании нового токена. Метод POST принимает в заголовке запроса информацию о логине и пароле пользователя и возвращает токен вместе с информацией о времени его устаревания. Важным является факт того, что после генерации токена на сервере не остается информации о выданном токене. Это позволяет избавиться от еще одного хранимого на сервере компонента состояния пользователя, что предписывается одним из принципов REST.

Дополнительно был реализован метод GET, который возвращает информацию о времени устаревания токена.

4 План перехода на RESTful API

Имея спроектированный интерфейс, необходимо разработать план перехода серверной части и клиентских приложений на новую архитектуру. Модель зрелости Ричардсона поможет определить, в каком состоянии находится интерфейс в данный момент, и что необходимо сделать, чтобы интерфейс стал по настоящему RESTful.

Модель зрелости Ричардсона (Richardson Maturity Model, RMM) — модель, определяющая четыре уровня, каждый из которых соответствует определенным этапам при переходе сервиса на архитектурный стиль REST.

Определим, на каком уровне зрелости находится спроектированный интерфейс.

4.1 Нулевой уровень зрелости по Ричардсону

Нулевой уровень соответствует интерфейсу, не достойному назваться RESTful. На этом уровне конечные точки определены, не опираясь на существующие ресурсы, из методов HTTP используется один или два метода, в ответах содержится смешанная неструктурированная информация.

Единственное требование, возникающее на нулевом уровне - это использование протокола HTTP для обмена данными.

Примером интерфейса нулевого уровня является интерфейс предыдущей версии системы, в котором не были явно выделены ресурсы, использование методов ограничивалось методами POST и GET, и все взаимодействие происходило по протоколу HTTP.

Интерфейсы нулевого уровня по модели зрелости Ричардсона не считаются RESTful.

4.2 Первый уровень зрелости по Ричардсону

На первом уровне в интерфейс вводится само понятие ресурса. Теперь запросы к разным ресурсам соответствуют разным конечным точкам. Но активное использование HTTP методов в интерфейсе ещё не наблюдается.

Примером интерфейса первого уровня является часть исходного интерфейса предназначенная для взаимодействия с проложением. Как пример в этой части наблюдаются такие конечные точки как `/devices/get/` и `/devices/delete/`, работающие через метод GET. Что идеально вписывается в описание первого уровня интерфейса по модели зрелости Ричардсона.

4.3 Второй уровень зрелости по Ричардсону

Второй уровень модели зрелости Ричардсона предполагает активное использование различных методов HTTP при работе с ресурсами.

Вообще говоря, протокол HTTP не закрепляет строго набор допустимых методов, и если клиент и сервер способны обрабатывать какой-то специфический метод, то протокол этого не запрещает. Но есть набор общепринятых методов для общения клиентов с сервером. Самыми частыми из них являются GET, POST, PUT и DELETE.

Второму уровню по модели зрелости Ричардсона соответствует спроектированный в данной работе интерфейс, так как в нем четко определены существующие ресурсы и методы взаимодействия с ними.

Интерфейсы, соответствующие второму уровню зрелости, уже могут считаться RESTful.

4.4 Третий уровень зрелости по Ричардсону

Последний уровень в модели зрелости Ричардсона вводит понятие представления гипермедиа. На данном уровне предполагается использование в интерфейсах архитектурные ограничения, называемые HATEOAS.

HATEOAS - Hypermedia As The Engine Of Application State, суть данного ограничения заключается в возвращении вместе с ответом гипермедиа определяющее возможные дальнейшие изменения состояния клиента. Таким образом сервер берет на себя логику определения допустимых состояний в которые может перейти клиент из текущего состояния, определяя эти состояния в возвращаемом гипермедиа. Например добавляя ссылки на объекты допустимые к получению текущему пользователю.

При проектировании не были заложены принципы HATEOAS, вследствие чего невозможно отнести обновленный интерфейс к третьему уровню по модели зрелости Ричардсона. Но фундамент, заложенный при проектировании интерфейса позволяет развить его до третьего, последнего уровня зрелости.

5 Описание реализации

Перед описанием реализации стоит описать общую архитектуру системы, начиная с самого верхнего уровня опускаясь до самого нижнего.

Весь проект разбит на серверную часть и клиентские приложения в соответствии с Рисунком 5.1. Архитектура серверной части спроектирована специально таким образом, чтобы добавление новых клиентских приложений было простым и не требовало доработок в серверной части. Минимальный предполагаемый набор клиентских приложений - это мобильное приложение сбора данных (исследуется в другой выпускной работе А. Фролова) и веб интерфейс операторов.

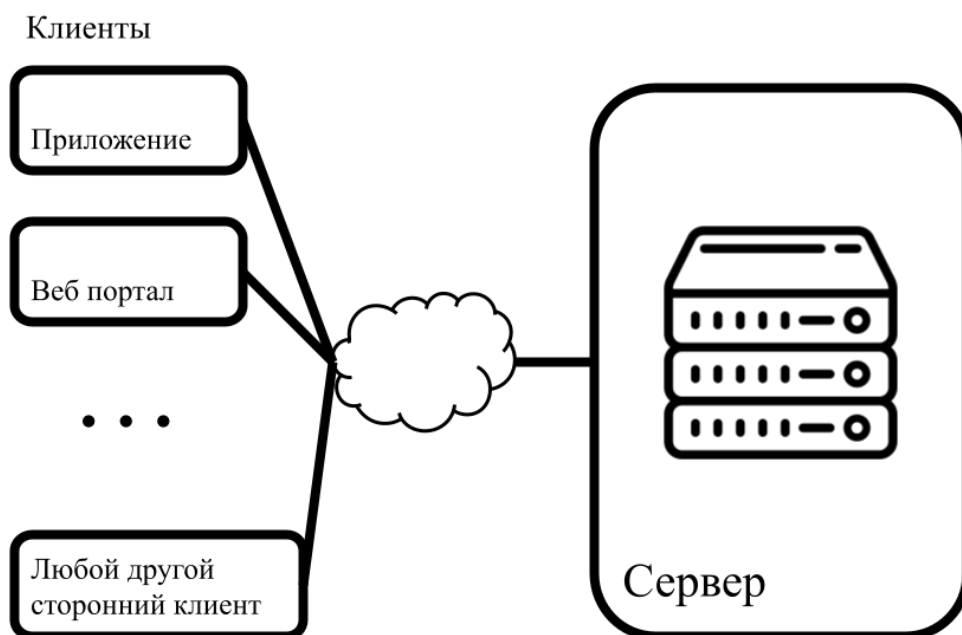


Рисунок 5.1

Рассмотрим подробнее серверную часть проекта. Она реализована в виде микросервисной архитектуры на базе технологии docker compose, позволяющей организовывать и конфигурировать набор docker контейнеров, каждый из которых представляет изолированную среду запуска отдельного микросервиса. Микросервисная архитектура использована с целью обеспечить горизонтальное масштабирование создаваемого сервиса сбора телеметрии - порождение новых экземпляров сервиса при росте числа клиентов.

Перечислим и опишем кратко каждый из существующих микросервисов. Общая схема взаимодействия микросервисов указана на Рисунке 5.2. Все сервисы можно разбить на два класса: обслуживающие MQTT составляющую проекта и обслуживающие HTTP составляющую проекта.

5.1 Микросервисы для MQTT

К потоку данных, передаваемому по протоколу MQTT, относятся следующие сервисы:

- mosquitto - основной контейнер, содержащий два процесса, mqtt брокер mosquitto и клиент, читающий из топики, в который клиенты пишут данные, и записывающий их в базу данных ClickHouse
- clickhouse - контейнер с базой данных записей данных, собранных с носимых устройств пользователей
- Набор сервисов, реализующих автомасштабирование контейнера mosquitto, подробнее про них можно прочитать в работе Антона Бодрова [5].

5.2 HTTP

- web_app - вся функциональность в одном контейнере, включая рендеринг страниц web приложения для операторов, старый интерфейс API
- mongo - база данных, используемая в исходном интерфейсе
- rest_api - новый API
- mysql - база данных для нового интерфейса
- nginx - отдельный контейнер, проксирующий запросы из открытого порта в нужный контейнер (web_app или rest_api)

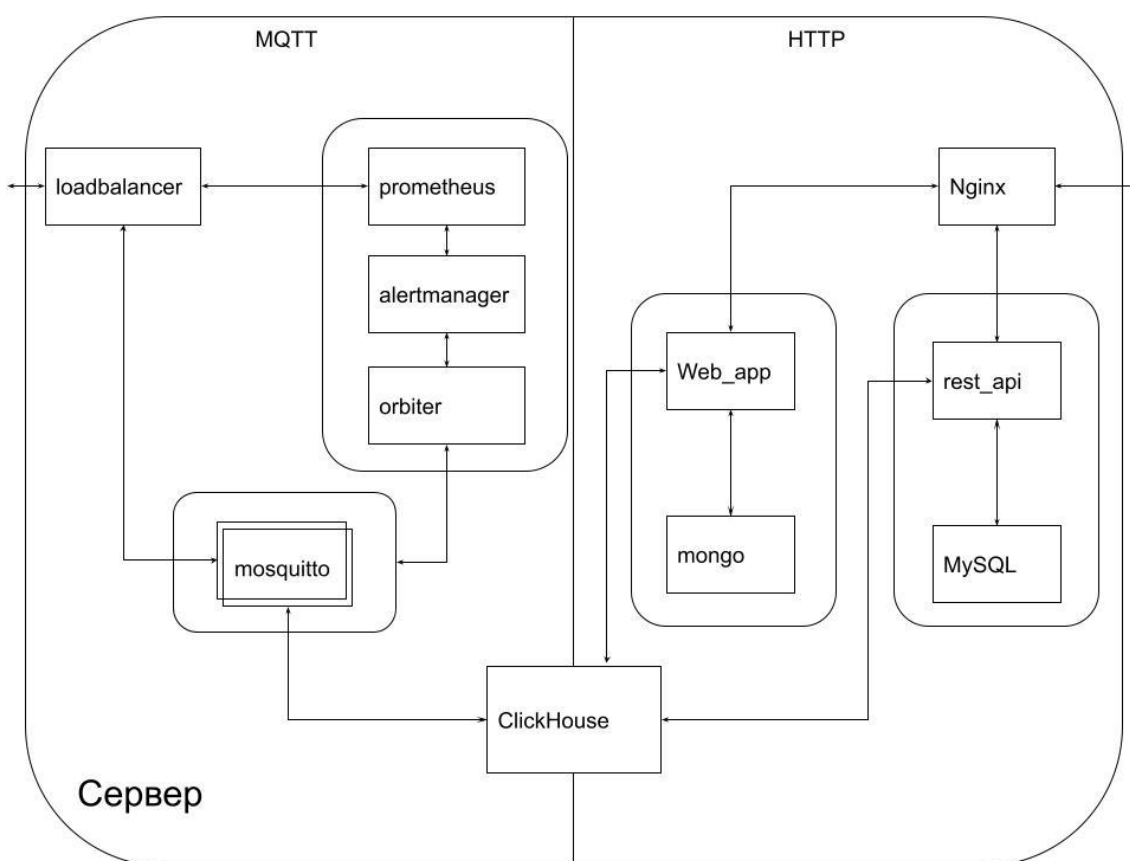


Рисунок 5.2

5.3 rest_api

Рассмотрим внутреннее устройство обновленного интерфейса. Схема взаимодействия внутренних компонентов этого микросервиса указана на Рисунке 5.3.

Для взаимодействия nginx с сервисом, написанном на языке Python существует стандарт называемый WSGI (Web Server Gateway Interface), в данный момент развивается асинхронная версия данного стандарта названная ASGI (Asynchronous Server Gateway Interface).

В сервисе rest_api применен актуальный стандарт ASGI, реализованный веб сервером под названием uvicorn. Именно он запускается внутри контейнера, принимая в качестве параметра приложение, в котором реализована логика.

В основу сервиса положен фреймворк connexion, который в свою очередь построен на базе популярного фреймворка Flask. Основной особенностью при выборе connexion в качестве основного фреймворка для разработки была возможность связывания имеющейся конфигурации OpenAPI и реализации в виде классов Class-based View, предлагаемых фреймворком Flask.

OpenAPI - это спецификация языка, на котором можно описать интерфейсы, начиная с HTTP-методов для конечной точки, и заканчивая типами принимаемых и возвращаемых параметров. Стандарт вырос из проекта под названием Swagger и способен гибко описывать самые разные конфигурации интерфейсов. Дополнительным достоинством использования OpenAPI является возможность, имея только спецификацию, автоматически генерировать код клиентских приложений, способных отправлять запросы и принимать ответы в указанном формате. Конечно, логика взаимодействия описывается дополнительно. Также OpenAPI предоставляет автоматическую

генерацию документации к описанным конечным точкам, с возможностью их вызова.

Фреймворк connexion, кроме связывания спецификации и логики работы той или иной конечной точки, способен выполнять роль валидатора как входных параметров, так и выходных следуя описанным в спецификации типам.

Файловая структура сервиса состоит из следующих основных компонентов:

- `openapi` - набор спецификаций
 - `paths` - описания существующих конечных точек
 - `schemas` - описания схем предоставляемых ресурсов
- `views` - набор файлов описывающих логику обработки запросов
- `models` - набор файлов, описывающих ресурсы, хранимые в базе данных

Еще одним нововведением обновленного сервиса стало использование технологии объектно-реляционного отображения (Object-Relational Mapping, ORM). Эта технология позволяет программисту абстрагироваться от написания запросов в базу данных и работать с ней как со стандартными объектами языка программирования, дополнительно ORM берет на себя задачу защиты от таких типов атак как SQL-инъекции. В исходном интерфейсе использовалось нечто похожее на ORM, но заточенный на конкретную базу данных `mongo`. В новом интерфейсе используется ORM `peewee`, поддерживающая 4 базы данных: `sqlite`, `mysql`, `postgresql` и `cockroachdb`, что позволяет при необходимости легко переключаться между ними.

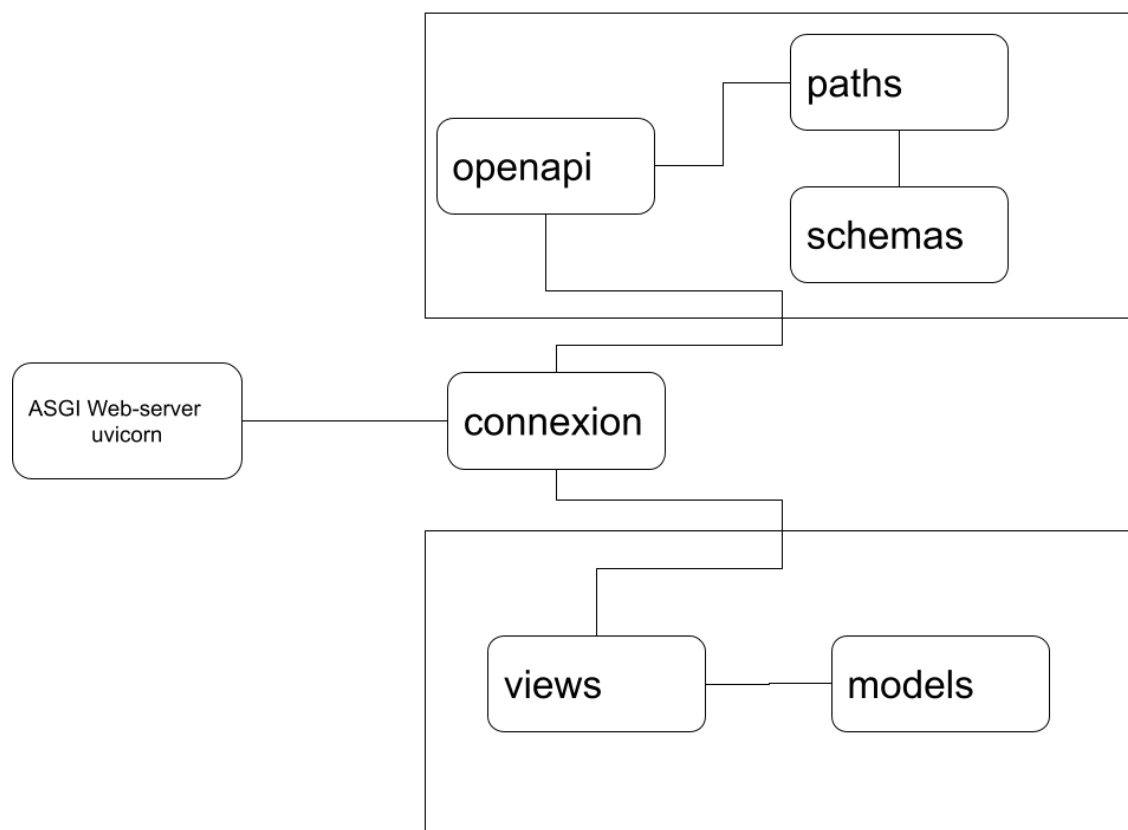


Рисунок 5.3

6 Экспериментальное исследование

Одной из фундаментальных целей работы была унификация интерфейса взаимодействия клиентов с сервером для обеспечения разработки клиентов без внесения изменений в код сервера.

В ходе экспериментального исследования был реализован новый клиент, загружающий данные с сервера компании Hexoskin (производителя умного жилета) и отправляющий эти данные на сервер IoT.

Общая схема работы программы включает всего два шага:

- загрузить данные из хранилища Hexoskin;
- отправить данные в собственное хранилище на сервере IoT.

6.1 Загрузка данных

Интерфейс взаимодействия с сервером Hexoskin как и разработанный интерфейс построен по принципам REST, благодаря чему, взаимодействие с ним устроено схожим образом.

Из документации интерфейса Hexoskin было получено описание структуры предоставляемых ресурсов. После чего, в целях проверки работоспособности интерфейса, была произведена попытка в ручном режиме через командную строку аутентифицироваться и получить данные за определенный период.

6.2 Отправка данных на сервер

Вторая часть программы состояла в отправке данных на сервер. Сами данные посылаются по протоколу MQTT. Но для их отправки необходимо обладать дополнительной информацией, такой как id пользователя, MAC адрес устройства и название считываемой характеристики, для формирования mqtt-топика и авторизационный токен для доступа к записи в этот топик.

Получение JWT токена реализуется по протоколу HTTP через разработанный интерфейс в конечной точке `“/api/v1/auth/user”`. Для этого достаточно предоставить в авторизационном заголовке строку из логина и пароля зашифрованные в base64, так называемое `“Basic authorization”`.

После успешного получения токена программа обращается к ещё одной конечной точке `“/api/v1/user”` предоставляющей информацию о пользователе. Из этой информации получен id текущего пользователя. Далее производится запрос зарегистрированных пользователем устройств доступных в конечной точке `“/api/v1/devices”`. Из этой информации удаётся получить MAC адреса устройств. Последний необходимый для составления названия топика элемент - это название характеристики устройства (в терминологии Bluetooth LE), т.е. показателя, передаваемого устройством. Его можно получить, составив запрос на получение информации о типе выбранного устройства в конечной точке `“/api/v1/device_type/{id}”`.

Получив всю необходимую информацию, генерируется название топика в формате `“c/{user_id}/{MAC}/{char_name}”`. После чего становится возможным отправка MQTT запроса на сервер и успешное завершение программы.

6.3 Результат

В итоге, за короткое время удалось реализовать полезный инструмент для интеграции внешнего сервиса с сервисом IoT, что было невозможно для исходного интерфейса в силу отсутствия удобной внешней документации и ограничений или полного отсутствия аналогичных конечных точек. Исходный текст клиента приведен в Приложении Б

В процессе разработки активно использовалась документация Hexoskin API и документация IoT API, где подробно описываются конечные точки и способы взаимодействия с ними.

Заключение

В результате проделанной работы удалось разработать сервис записи телеметрии с интерфейсом, открытым к дальнейшему масштабированию архитектуры сервиса.

Теперь сторонним разработчикам доступен открытый API и документация к нему, что заметно ускорит скорость разработки клиентов для системы. Также из-за распространенности интерфейсов, спроектированных следуя тем же принципам REST, разработчик, уже знакомый с ними, с легкостью освоит и данный интерфейс.

Кроме удобства для разработчиков клиентов, благодаря унифицированному подходу, строгому определению структуры сервиса и заложенным принципам в основе интерфейса расширение функциональности самого сервиса так же упростилось, что открывает новые возможности для интеграций со сторонними сервисами и разработки различных пользовательских интерфейсов. Что было показано в работе на примере получения записей медицинской телеметрии из частного облака Nexoskin.

Однако, по полученный интерфейс соответствует второму уровню по модели зрелости Ричардсона..


Цель поставленная в начале работы достигнута.

Список литературы

1. Князев Е.И. Развитие сервиса сбора и обработки медицинской телеметрии
2. Аникевич, Юлия. Разработка и реализация серверной части платформы для сбора и обработки медицинской телеметрии
3. Лебедев Г.С. Интернет медицинских вещей: первые шаги по систематизации [Электронный ресурс] URL: https://jtelemed.ru/sites/default/files/jtm_9-15.pdf (Дата обращения: 14.05.2023)
4. The OpenAPI Specification. [Электронный ресурс] URL: <https://www.openapis.org> (Дата обращения: 14.05.2023)
5. Бодров А.О. Построение масштабируемого сервиса сбора и обработки данных на примере медицинской телеметрии [Электронный ресурс] URL: <https://istina.msu.ru/publications/article/515128316/>
6. Документация по фреймворку connexion [Электронный ресурс] URL: <https://connexion.readthedocs.io> (Дата обращения: 14.05.2023)
7. Филдинг Р. Architectural Styles and the Design of Network-based Software Architectures — 2000 г. — [Электронный ресурс] URL: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> (Дата обращения: 14.05.2023)
8. Спецификация Asynchronous Server Gateway Interface [Электронный ресурс] URL: <https://asgi.readthedocs.io> (Дата обращения: 14.05.2023)
9. Микросервисная архитектура [Электронный ресурс] URL: <https://en.wikipedia.org/wiki/Microservices> (Дата обращения: 14.05.2023)
10. Без Серверные вычисления [Электронный ресурс] URL: https://en.wikipedia.org/wiki/Serverless_computing

11. Нельсон Б. Remote Procedure Call — 1981 г — Xerox Palo Alto Research Center. PhD thesis
12. Спецификация GraphQL [Электронный ресурс] URL: <https://graphql.org/>

Приложение А

 **IoMT**

[Получить данные](#) [Информация о пациентах](#) [Панели мониторинга](#) [Админ-панель](#) [Выход](#)


Операторы

1	admin	Удалить
2	sanyavertolet	Удалить

[Создать оператора](#)

Доступные устройства

[Добавить устройство](#)

 **IoMT**

[Получить данные](#) [Информация о пациентах](#) [Панели мониторинга](#) [Админ-панель](#) [Выход](#)

Шаг 1

Выберите пользователя:

Пользователь:

 **IoMT**

[Получить данные](#) [Информация о пациентах](#) [Панели мониторинга](#) [Админ-панель](#) [Выход](#)

Шаг 1

Выберите пользователя:

Пользователь:

Приложение Б

```
from datetime import datetime
import json
import requests
from requests.auth import HTTPBasicAuth
import base64
from pprint import pprint
import paho.mqtt.publish as publish
from paho.mqtt.client import MQTTv5

login, password = input("Enter login: "), input("Enter password: ")
basic = HTTPBasicAuth(login, password)

r = requests.get(
    #f"https://api.hexoskin.com/api/data/?record=241228&datatype=19",
    f"https://api.hexoskin.com/api/record/",
    auth=basic,
)

login, password = input("Enter login: "), input("Enter password: ")
basicc = HTTPBasicAuth(login, password)

rr = requests.post(
    #f"https://api.hexoskin.com/api/data/?record=241228&datatype=19",
    f"https://iomt.lvk.cs.msu.ru/api/v1/auth/user",
    auth=basicc,
)
```

```

token = rr.json()['token']
rr = requests.get(
    #f"https://api.hexoskin.com/api/data/?record=241228&datatype=19",
    f"https://iomt.lvk.cs.msu.ru/api/v1/user",
    headers={'Authorization': f'Bearer {token}'},
)
user_id = rr.json()['id']
topic = f'c/{user_id}/EC:2E:98:73:A1:6B/heartRate'

for obj in r.json()['objects']:
    id = obj['id']
    r = requests.get(
        f"https://api.hexoskin.com/api/data/?record={id}&datatype=19",
        auth=basic,
    )
    data = r.json()[0]['data']['19']
    msgs = [
        (topic, json.dumps({'timestamp': datetime.fromtimestamp(obj[0] /
256).isoformat(), 'value': str(obj[1])}), 2, False)
        for obj in data
    ]
    publish.multiple(
        msgs,
        hostname='iomt.lvk.cs.msu.ru',
        port=1883,
        client_id='grabber',
        auth=dict(
            username=token,

```

```
        password=token,  
    ),  
    protocol=MQTTv5,  
)
```