Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра автоматизации систем вычислительных комплексов

Фролов Александр Валерьевич

Развитие клиентскои части сервиса сбора и обработки медицинскои телеметрии

Курсовая работа

Научный руководитель:
к.ф.-м.н.
Бахмуров Анатолий Геннадьевич
Научный консультант:
Любицкий Александр Андреевич

Москва, 2022

Аннотация

Развитие клиентскои части
сервиса сбора и обработки медицинскои телеметрии

Фролов Александр Валерьевич

Данная курсовая работа нацелена на улучшение существующей клиентской части сервиса сбора и обработки медицинской телеметрии с устройств интернета медицинских вещей. В ходе работы рассмотрена возможность подключения новых устройств при помощи использования конфигурационных файлов, проведены испытания стабильной работы приложения, реализован пользовательский интерфейс для отображения данных, полученных с произвольного датчика, подключенного при помощи конфигурационного файла.

В результате работы получено приложение, поддерживающее подключение новых устройств при помощи создания конфигурационного файла. Данное улучшение приложения обеспечивает масштабируемость клиентской части сервиса на большее число устройств интернета медицинских вещей.

Abstract

Development of medical telemetry service's client part

Abstract

Содержание

1 Введение

Интернет вещей стал неотъемлемой частью нашей жизни. Все мы слышали про умные колонки, которые служат центром умного дома, умные лампочки, весы. Более того, стремительно набирает популярность подвид интернета вещей - интернет медицинских вещей (IoMT, Internet of Medical Things). Интернет медицинских вещей - множество устройств, программ и сервисов, объединенных в одну систему, сеть.

Практически каждый из нас видел умные часы и браслеты, которые могут считывать важные показатели жизнедеятельности человека в реальном времени. Такие браслеты позволяют измерять пульс, сатурацию, суточную активность и даже качество сна. Область применения устройств умного дома крайне широка. Людям, которым необходимо находиться под наблюдением врачей, такие датчики могут позволить эффективнее следить за своим здоровьем. Профессиональные спортсмены, которые каждую тренировку находятся на пике своих возможностей, могут контролировать свое состояние во избежание травм. Обыкновенному человеку такие устройства помогают улучшить качество жизни и держать свой организм в тонусе.

С ростом числа различных устройств возникает вопрос: как агрегировать несколько разных устройств? Сейчас каждый производитель устройств интернета медицинских вещей создает свою платформу для получения, обработки и хранения информации. Все эти сервисы зачастую имеют закрытый исходный код, что не дает возможность каким-то образом изменить существующее решение, доработав его. Более того, в такие сервисы невозможно внедрить какой-либо свой алгоритм диагностирования заболевания. Еще одним минусом таких решений является отсутствие поддержки устройств других производителей. Купив умный браслет *Huawei* мы обязаны использовать фирменное приложение *Huawei Health*, которое будет хранить наши данные на своих серверах.

Одной из актуальных задач является создание универсальной платформы сбора и обработки медицинской телеметрии с открытым исходным кодом. Было принято решение создать платформу мониторинга показателей жизнедеятельности человека на факультете вычислительной математики и кибернетики Московского Государственного Университета. Выпускная квалификационная работа Чайчица Даниила [1] служит основой для данной курсовой работы. Поддержка единственного устройства *Hexoskin* [2] компании *CARRÉ TECHNOLOGIES INC.*, реализованная Даниилом сильно ограничивает возможности сервиса, в связи с чем было принято решение изучить масштабируе-

мость платформы на большее число устройств.

Данная работа посвящена улучшению мобильного приложения для сбора и обработки физиологических данных о человеке, написанного в рамках выпускной квалификационной работы студена Чайчица Даниила. Проведено исследование способов взаимодействия с устройствами интернета медицинских вещей. Предложен способ подключения новых устройств. Улучшен пользовательский интерфейс. Произведено экспериментальное исследование корректной работы платформы с новыми устройствами.

2 Постановка задачи

Целью данной работы является улучшение клиентской части платформы для сбора и обработки физиологических данных о человеке путем добавления поддержки новых устройств. Требуется реализовать подключение новых устройств путем написания конфигурационных файлов.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1. Изучить методы подключения устройств на основе спецификации Bluetooth Low Energy;
2. Разработать язык описания устройств и величин, которые они могут считывать;
3. Реализовать чтение конфигурационного файла, генерацию пользовательского интерфейса и обработку данных, полученных с устройства;
4. Провести экспериментальное исследование корректной работы приложения, в том числе при помощи специального тестового приложения, симулирующего устройство IoMT.

3 Краткое описание спецификации Bluetooth low energy

3.1 Базовые понятия

Bluetooth был задуман как технология связи ближнего диапазона, призванная заменить провода, необходимые для работы различных периферийных устройств таких как компьютерные мыши, наушники или клавиатуры.

В настоящее время существует два типа устройств с поддержкой Bluetooth[3]:

1. Bluetooth Classic, который используется, например, в автомобильных звуковых системах и наушниках;

2. Bluetooth Low Energy (BLE), то есть Bluetooth с низким энергопотреблением. Он чаще всего применяется в приложениях, чувствительных к энергопотреблению или в устройствах, передающих небольшие объемы данных с большими перерывами между передачами.

Поскольку во многих устройствах Интернета Вещей (IoT) используются небольшие устройства и датчики, BLE стал наиболее часто используемым протоколом связи в приложениях Интернета Вещей.

3.2 Архитектура BLE

Рисунок 1 иллюстрирует различные уровни, присущие архитектуре BLE. Три главных блока в этой архитектуре – приложение, хост и контроллер.

3.3 Протокол атрибутов (ATT)

Протокол атрибутов (или ATT) определяет, в каком виде сервер представит свои данные клиенту и как эти данные будут структурированы. Существует две роли, связанные с ATT:

- **Сервер** - это устройство, которое предоставляет данные и получает входящие команды от связанного устройства и отправляет ответы, уведомления и индикации.

  В данной курсовой работе в роли сервера выступает устройство интернета медицинских вещей, например, умный браслет. Он предоставляет данные о пульсе, активности и своем уровне заряда батареи.

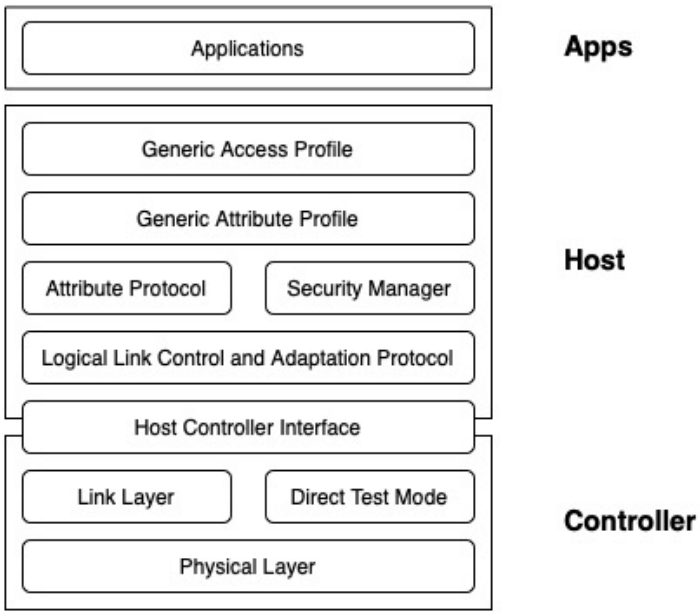

Рис. 1: Архитектура BLE

- **Клиент** - это устройство, которое взаимодействует с сервером с целью считать предоставляемые сервером данные и/или для того, чтобы контролировать его поведение.

  В данной курсовой работе мобильный телефон, на котором запущен клиент платформы сбора медицинской телеметрии выступает в роли клиента.

Данные, предоставляемые сервером, сгруппированы в атрибуты. Атрибут - это общий термин для любых типов данных, предоставляемых сервером, определяющий структуру этих данных. Например, сервисы и характеристики (будут описаны позднее) являются атрибутами. Ниже состав атрибута:

Тип атрибута (универсальный уникальный идентификатор, UUID) - это 16-битное (в случае стандартных атрибутов Bluetooth Special Interest Group [4]) или 128-битное число (в случае атрибутов, определенных разработчиком устройства, vendor-specific UUID). Например, UUID для одобренного консорциумом атрибута значения температуры:

0x2A1C

Одобренные консорциумом типы атрибутов имеют один общий (за исключением 16 бит) специальный 128-битный UUID:

0000XXXX-0000-1000-8000-00805F9B34FB

16-битный UUID будет подставлен вместо символов XXXX в базовом UUID. Собственный UUID может быть любым 128-битным числом, не совпадающим ни с одним из одобренных Bluetooth Special Interest Group базовых UUID. Например, разработчик может создать свой собственный UUID для показаний температуры, такой как:

FF321A4E-78DE-2B5A-F1E4-11D0D640FD6E

Дескриптор атрибута - это 16-битное число, которое сервер присваивает каждому из своих атрибутов. Это число используется клиентом как ссылка на конкретный атрибут, и сервер гарантирует, что эта ссылка будет уникальной для атрибута, которому она присвоена, в течении всего времени существования соединения между устройствами.

Права атрибута определяют, может ли атрибут быть прочитан или записан, может ли он посылать уведомления или индикации, и какие уровни доступа требуются для каждой из этих операций.

3.4 Сервисы

Сервисы это группа из одного или большего числа атрибутов, некоторые из которых являются характеристиками. Он предназначен для группировки связанных атрибутов, удовлетворяющих специфической функциональности сервера. Например, одобренный SIG сервис батареи содержит одну характеристику под названием “уровень заряда”.

3.5 Характеристики

Характеристика - часть сервиса, предоставляющая часть данных, которые сервер предоставляет клиенту. Например, характеристика уровня заряда батареи показывает оставшийся уровень заряда батареи в устройстве, который может быть прочитан клиентом. Характеристика содержит набор атрибутов, которые облегчают работу с содержащимися в характеристике данными:

- **Свойства** представлены набором бит, определяющих то, каким образом значение характеристики может использоваться. Пример свойств: чтение, запись, запись без ответа, уведомление, индикация.

- **Дескрипторы** используются для хранения информации, связанной со значением характеристики. Примеры использования: расширенные свойства, пользовательское описание, поля, используемые для подписки на уведомления и индикации, поля, описывающие представление характеристики, такие как формат или единица измерения.

3.6 Промежуточные итоги

На основании архитектуры спецификации BLE следует заключить, что, для того чтобы описать какое-либо устройство интернета вещей, необходимо знать список характеристик (в особенности их универсальные уникальные идентификаторы), хранящих значения измеряемых величин. Для того чтобы получить доступ к характеристике, необходимо знать универсальный уникальный идентификатор сервиса, предоставляющего данную характеристику.

4 Описание устройства IoMT

На основании предыдущей главы можно утверждать, что множество `Characteristics`, состоящее из пар UUID сервиса(`ServiceUUID`) и характеристики(`CharacteristicUUID`), описывает множество измеряемых устройством величин.

Помимо этого, для каждого элемента множества `Characteristics` необходимо предоставить название, которое будет отражать смысл считанной величины.

Таким образом, предоставив множество `Characteristics`, можно описать устройство интернета вещей, подключение которого происходит при помощи спецификации BLE. Предоставить множество `Characteristics` можно при помощи создания класса, реализующего интерфейс, свойственный данному устройству или использованием конфигурационных файлов. Написание класса является более понятным способом с точки зрения программиста, так как читаемость кода значительно выше читаемости конфигурационных файлов. Более того, часть ошибок реализации конкретного класса может быть выявлена на этапе компиляции. С другой стороны, при добавлении новых устройств необходимо реализовывать новые классы, что увеличивает размер проекта и, конечно, требует компиляции, в результате чего пользователи приложения будут вынуждены ждать обновлений, а программисты - поддерживать проект. К тому же, написание классов может привести к дублированию кода.

Использование конфигурационных файлов, напротив, решает проблему необходимости компилировать код всякий раз, когда добавляется поддержка нового устройства. Достаточно лишь написать конфигурационный файл или загрузить с сервера уже написанный конфигурационный файл.

4.1 Структура конфигурационного файла

Как было сказано ранее, необходимо предоставить множество вида: `Characteristics = { ServiceUUID, CharacteristicUUID, CharacteristicName }` где `ServiceUUID` - универсальный уникальный идентификатор сервиса, `CharacteristicUUID` - универсальный уникальный идентификатор характеристики, `CharacteristicName` - название величины, отражающее ее смысл. Конфигурационный файл должен включать в себя одну или несколько секций, состоящих из этой четверки параметров. Для удобства работы с конфигурационным файлом требуется определить секцию General, содержащую в себе

дополнительную информацию об устройстве интернета медицинских вещей, такую как название устройства, тип устройства (браслет, жилет, какое-либо иное устройство).

4.2 Поддержка конфигурационных файлов на стороне клиента

Для того чтобы корректно отображать данные на экране телефона, необходимо отказаться от статического окна приложения, реализованного в предыдущей версии приложения в пользу динамически генерируемого окна. В таком окне должен быть слой, отображающий информацию об устройстве, и несколько слоев, отображающих информацию о каждой отдельной характеристике (Рис. 2).

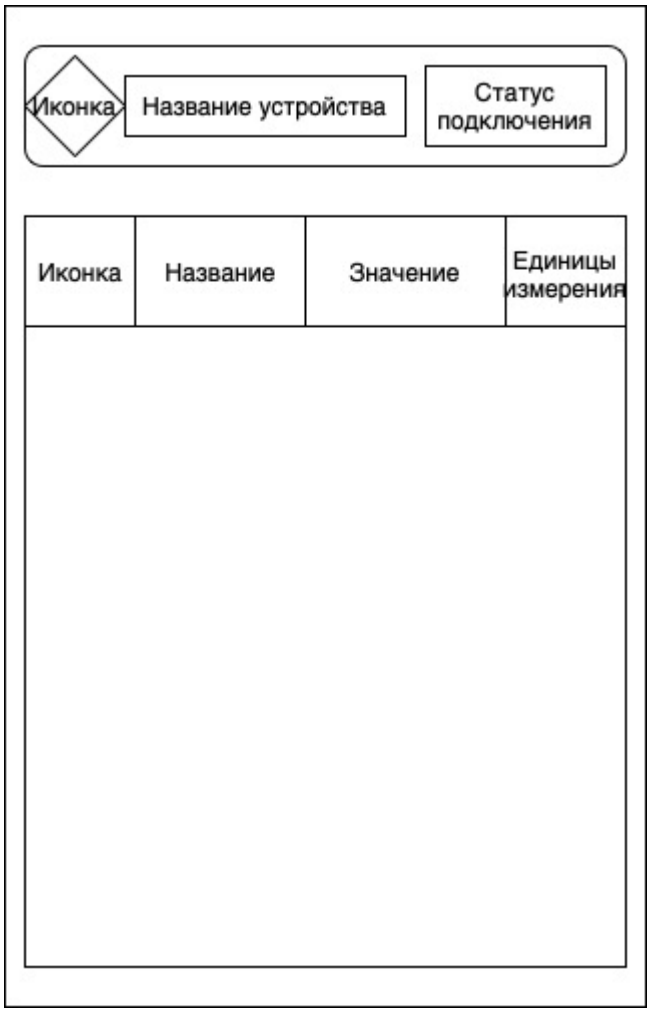

Рис. 2: Прототип графического пользовательского интерфейса

4.3 План работ

1. Разработать парсер конфигурационных файлов;
2. Написать тесты для парсера конфигурационных файлов;
3. Разработать графический интерфейс, необходимый для отображения полученных данных;
4. Реализовать подключение к устройству при помощи конфигурационного файла и считывание с него данных;
5. Разработать тестирующее приложение, которое будет эмулировать устройство интернета медицинских вещей и провести испытание корректной работы клиента при помощи этого приложения;
6. Провести экспериментальное исследование корректной работы подключения и считывания данных различных реальных устройств интернета медицинских вещей.

5 Краткое описание Kotlin

5.1 Сравнение Java и Kotlin

Предыдущая реализация клиента сервиса сбора и обработки медицинской телеметрии была написана на языке программирования Java - строго типизированном объектно-ориентированном языке программирования общего назначения, разработанном компанией Sun Microsystems, в том числе использующимся для создания мобильных приложений под операционную систему Android.

Java имеет свои недостатки. Для того чтобы реализовать простую программу, требуется написать достаточно много кода. Вот пример реализации вариации программы `"Hello world"`, написанной на Java:

```
public class Main() {
        public static void printJava() {
                System.out.println("Goodbye Java!");
        }
}
```

Видно, что для вывода на печать простой строчки требуется создать класс `Main`, в котором будет реализован статический метод `printJava`, который в свою очередь обращается к полю класса `System`, чтобы напечатать на экран заветную строчку.

Рассмотрим, что нужно сделать в Kotlin [5], чтобы напечатать вывести строку на печать:

```
fun printKotlin() = println("Hello Kotlin!")
```

Пять строчек превратились в одну, показывая, как Kotlin решает проблему избыточности Java, что делает код на Kotlin более читаемым.

5.2 Обнуляемые типы данных

По умолчанию все типы в языке программирования Kotlin не могут принимать значение `null` - константе, которая обозначает, что объект не существует. Для того чтобы использовать обнуляемые типы данных, необходимо добавить знак вопроса к типу данных:

```
val nonNullable: String = ""     // non-nullable
val nullable: String? = null     // nullable
```

Таким образом, происходит четкое отделение обнуляемых типов данных от необнуляемых, что повышает безопасность и снижает риск `NullPointerException`.

5.3 Data class

Другой удобной особенностью языка является возможность создавать так называемые дата-классы. Дата-класс (или `data class`) - класс, содержащий в себе лишь данные и методы, необходимые для манипуляций с этими данными. Приведем пример дата-класса на Java:

```
public class Person {
        private String name;
        private String fatherName;
        private Integer age;

        public Person(String name, String fatherName, Integer
           age) {
            this.name = name;
            this.fatherName = fatherName;
            this.age = age;
        }

        public String getName() {
                return this.name;
        }

        public String getFatherName() {
                return this.fatherName;
        }

        public Integer getAge() {
```

```
                return this.age;
        }

        @Override
        public String toString() {
            return "Person(name=" + name + ", fatherName=" +
               fatherName + ", age=" + age.toString() + ")";
        }
}
```

Эквивалентный код на языке Kotlin:

```
data class Person(val name: String, val fatherName: String,
   val age: Int)
```

5.4 kotlinx.serilization

В дополнение к Kotlin data class стоит отметить библиотеку *kotlinx.serilization*, которая позволяет конвертировать объект дата-класса в строку нужного формата - Json, HOCON и другие. Более того, благодаря активной поддержке сообщества, существует поддержка и других форматов - toml, yaml и так далее. Для того чтобы пометить дата-класс как класс, который может быть использован для сериализации или десериализации, необходимо перед объявлением класса добавить аннотацию **@Serializable**.

В этой курсовой работе используется библиотека *kotlinx.serilization*[6] и сторонняя библиотека *ktoml*[7], реализующая сериализацию и десериализацию toml-файлов.

5.5 Промежуточный итог

На основании всего вышесказанного было принято решение переписать весь существующий код с Java на Kotlin.

6 Реализация использования конфигурационных файлов

Как было сказано в главе 3, конфигурационные файлы могут быть использованы для подключения новых устройств к сервису IoMT. Для этого необходимо задать название устройства, регулярное выражение, по которому будет происходить фильтрация устройств и список считываемых величин со всеми необходимыми универсальными уникальными идентификаторами. Рассмотрим реализацию работы с конфигурационными файлами.

6.1 Формат и вид конфигурационного файла

Конфигурационные файлы, используемые в рамках данной курсовой работы, было принято писать в формате toml [8], так как он является одним из наиболее читаемых форматов благодаря его понятному синтаксису. Toml так же поддерживает комментарии, что может упростить написание документации.

Как было сказано ранее, конфигурационный файл должен состоять из $n + 1$ блока, где n - количество величин, которые способно отдавать на чтение устройство интернета медицинских вещей (1 дополнительный блок хранит информацию об устройстве, с которого считываются данные).

Ниже приведен пример конфигурационного файла, описывающего устройство *Mi Band 6* производителя *Xiaomi*:

```
[general]                                  // General block
    name = "Mi Band 6"                     // Device name
    nameRegex = "Mi.*6"                    // Regex for a name
       to filter while connecting
    type = "band"                          // Type of a device
    characteristicNames = ["heartRate"]    // List of
       characteristics
[heartRate]                                // Characteristic
   block for heartRate characteristic
    name = "Heart Rate"                    // Characteristic name
```

```
    serviceUUID = "0x180D"                  // Service UUID
    characteristicUUID = "0x2A37"           // Characteristic UUID
```

Как видно, данный конфигурационный файл описывает устройство с именем *Mi Band 6*, которое удовлетворяет регулярному выражению *Mi.*6*, по которому будет осуществляться фильтрация устройств при поиске. Тип *Mi Band 6* - браслет (это используется для отображения нужной иконки), а список характеристик состоит всего из одного параметра - частоты сердечных сокращений. После блока general следует блок, описывающий взаимодействие с сервером, который работает на браслете и отвечает за обмен считанными величинами.

6.2 Дата-классы, описывающие сущности конфигурационных файлов

Дата-класс, отвечающий за блок general на ЯП Kotlin:

```
@Serializable
data class GeneralConfig(
    var name: String,
    var nameRegex: String,
    var characteristicNames: List<String>,
    var type: String? = null,
)
```

Дата-класс, отвечающий за блок характеристики на ЯП Kotlin:

```
@Serializable
data class CharacteristicConfig (
    var name: String,
    var serviceUUID: String? = null,
    var characteristicUUID: String? = null,
    var descriptorUUID: String? = null,
)
```

Чтобы не хранить отдельно список характеристик и общую часть конфигурационного файла, был написан дата-класс DeviceConfig:

```
data class DeviceConfig(
    val general: GeneralConfig,
    val characteristics: Map<String, CharacteristicConfig>,
)
```

6.3 Класс ConfigParser

Класс ConfigParser должен получать строку - содержимое корректного toml-файла и возвращать объект класса DeviceConfig. Реализация класса ConfigParser на ЯП Kotlin:

```
class ConfigParser {
  private val tomlConfig = TomlConfig(
    ignoreUnknownNames = true,
  )
  private var tomlLines: List<String> = emptyList()

  fun parseGeneralConfig(): GeneralConfig =
    Toml(tomlConfig).partiallyDecodeFromString(serializer(),
       tomlLines, "general")

  private fun parseCharacteristicConfig(characteristicName:
     String): CharacteristicConfig =
    Toml(tomlConfig).partiallyDecodeFromString(serializer(),
       tomlLines, characteristicName)

  fun parseAllCharacteristicConfigs(characteristicNames:
     List<String>): Map<String, CharacteristicConfig> =
    characteristicNames.associateWith {
       parseCharacteristicConfig(it) }

  override fun toString(): String =
    tomlLines.joinToString(separator = "\n")
```

```
  fun parseFromString(tomlString: String): DeviceConfig {
    tomlLines = tomlString.split("\n")
    return parse()
  }

  fun parse(): DeviceConfig =
      parseGeneralConfig().let { DeviceConfig(it,
         parseAllCharacteristicConfigs(it.characteristicNames))
         }
}
```

В дополнение к классу `ConfigParser` были написаны тесты, в которых проверяется корректность работы класса как на правильном конфигурационном файле, так и на неполном или неправильном.

7 Результаты работы и тестирование

В данной главе рассматриваются результаты реализации класса `ConfigParser`, описанного в предыдущей главе, и динамически генерируемого пользовательского интерфейса.

7.1 Внешний вид приложения

По схеме (Рис. 2) был реализован графический интерфейс, при помощи которого пользователь приложения может получать информацию о значениях величин, считанных с устройства IoMT. Для динамической генерации окна, отображающего считанные величины, был использован `LinearLayout`, состоящие из `ImageView` и двух `TextView` [9]. В дополнение к этому, были добавлены различные иконки, отображающие тип устройства и тип величины, которая была считана.

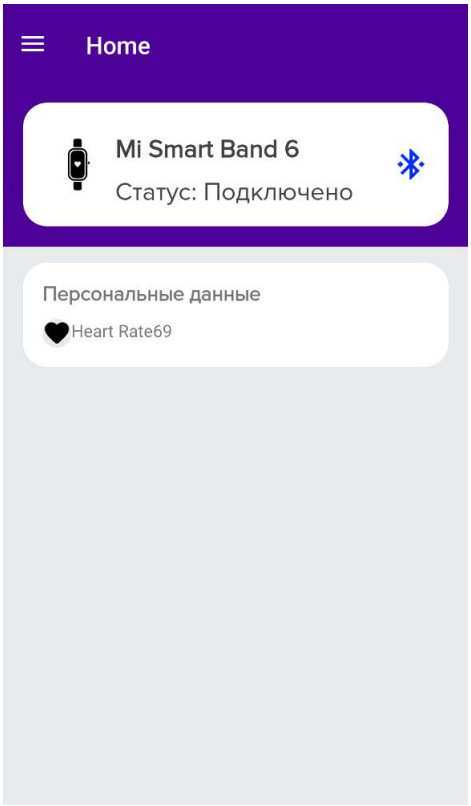

Рис. 3: Графический пользовательский интерфейс

7.2 Тестирование

Для тестирования работы приложения был приобретен фитнес-браслет *Mi Band 6* компании *Xiaomi*. Подключение произошло успешно (Рис. 3), данные были считаны без ошибок, после чего сохранены в локальную базу данных и отправлены на сервер при помощи протокола MQTT.

Так же в ходе курсовой работы было написано приложение, *Virtual Device* эмулирующее работу устройства интернета медицинских вещей, выступая в роли BLE сервера. В тестовом приложении можно выбрать интервал, в течение которого клиент получает уведомление об обновлении характеристики, после чего клиент считывает эту характеристику.

Были написаны конфигурационные файлы для *Hexoskin*, *Virtual Device* и *Mi Scale 2* - умных весов компании *Xiaomi*.

8 План дальнейшей работы

В данный момент существует несколько направлений, которые можно проработать в рамках выпускной квалификационной работы.

В первую очередь необходимо развивать приложение *Virtual Device*. Сейчас тестовое приложение поддерживает симуляцию только одной величины - частоты сердечных сокращений. Требуется реализовать создание сервера BLE при помощи таких же конфигурационных файлов, что используются для подключения к устройствам. Таким образом, в тестовом приложении можно будет создать конфигурацию из характеристик, чтобы проверить корректность работы со специфическими величинами и нестандартными UUID. Более того, требуется рассмотреть поведение устройства в критических ситуациях, в том числе при отсутствии связи, непредвиденном выключении устройства, ошибках передачи данных.

Другим направлением развития данного приложения является поддержка "многозначных"характеристик. Возможна ситуация, когда одна характеристика содержит в старших битах одну величину, а в младших - другую. Для поддержки таких составных характеристик требуется дополнить блоки характеристик необязательным полем, обозначающим диапазон битов, отвечающих непосредственно за данную величину.

Еще одним вариантом развития является поддержка считывания данных при через другие средства связи. Так как некоторые устройства (например браслеты и часы компании *Huawei*) не позволяют пользовательским приложениям получать данные, необходимо развивать конфигурационные файлы с целью поддержки получения данных при помощи REST API. Более того, некоторые медицинские датчики способны считывать кардиограмму. Так как кардиограмма подразумевает достаточно высокую частоту обновления, отправка такого объема данных невозможна по протоколу BLE. Отправляемые данные выглядят как обычный .wav файл, а для работы со звуковыми файлами используется классический Bluetooth.

9 Заключение

В рамках данной курсовой работы было рассмотрено масштабирование клиентской части сервиса сбора и обработки медицинской телеметрии путем написания конфигурационных файлов, реализован парсер конфигурационных файлов, пользовательский интерфейс, отображающий считанные с подключенного устройства данные.

Было разработано приложение для тестирования обмена данными с клиентом, произведено подключение к различным устройствам интернета медицинских вещей при помощи конфигурационных файлов, намечены планы на дипломную работу.

Список литературы

[1] *Чайчиц, Д. А.* Разработка и реализация клиентской части платформы для сбора и обработки медицинской телеметрии. — 2021.

[2] *INC, CARRE TECHNOLOGIES.* Hexoskin. — 2022. — Последнее посещение 20.12.2021. `https://www.hexoskin.com`.

[3] *Bluetooth SIG, Inc.* Bluetooth. — 2022. — Последнее посещение 23.05.2022. `https://www.bluetooth.com`.

[4] *Bluetooth SIG, Inc.* 16-bit UUID Numbers Document. — 2022. — Последнее посещение 24.02.2022. `https://www.bluetooth.com/specifications/assigned-numbers/`.

[5] *JetBrains.* Kotlin Docs. — 2022. — Последнее посещение 20.05.2022. `https://kotlinlang.org/docs/home.html`.

[6] *JetBrains.* kotlinx.serilization. — 2022. — Последнее посещение 14.04.2022. `https://github.com/Kotlin/kotlinx.serialization`.

[7] *Kuleshov, Andrey.* ktoml. — 2022. — Последнее посещение 15.04.2022. `https://github.com/akuleshov7/ktoml`.

[8] *Preston-Werner, Tom.* TOML. — 2021. — Последнее посещение 13.04.2022. `https://toml.io`.

[9] *Google.* Android Developer. — 2022. — Последнее посещение 15.05.2022. `https://developer.android.com`.

[10] *Dmitrii, Bainak.* IoT service based on BLE technology. — 2018.