M.Sc. Thesis
Master of Science in Engineering

DTU Health Tech
Department of Health Technology

The Gardener Framework

An open-source programming framework for collection of wearable activity and health data from web-based services

János Richárd Pekk (s192617)

Kongens Lyngby 2022

DTU Health Tech
Department of Health Technology
Technical University of Denmark

Ørsteds Plads
Building 345C
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
healthtech-info@dtu.dk
www.healthtech.dtu.dk

Abstract

Smart, wearable devices (wearables) have seen a rapid increase in popularity in recent years. Among many others, health and fitness monitoring devices such as smartwatches and bio-multifunction wearable glucose sensors are on the rise. These wearable technologies can greatly facilitate patient monitoring for the healthcare sector. Various types of data can be collected about the user, such as physiological data (e.g., heart rate, Oxygen Level in Blood (SpO2), skin temperature, blood glucose, a.o) and behavioral data (e.g., physical activity, sleep, location, a.o), which could be used to detect medically significant events and improve the quality of life [1], [2].

However, each wearable device collects data in a different way, in a different data schema and format, and the data is uploaded to the vendor's warehouse. As such, it raises several difficulties, i.e., communicating with specific third-party APIs (e.g., Fitbit, Garmin, Dexcom), handling the authentication process with the API, retrieving specific data from it, and mapping that data into a specific format.

The goal of the project is to design and implement a software framework that can collect data from various third-party devices and convert it into a particular format. The designed framework must be easily extensible, which means that integrating new devices into the existing ecosystem should be quick and easy (plug-n-play). The framework should also be able to persist the information needed for data collection and publish the processed results to a message queue (such as RabbitMQ[1]).

The project will be developed in collaboration with Copenhagen Center for Health Technology (CACHET)[2] and aims to provide a solution for their data integration needs for their current platform.

[1] `https://www.rabbitmq.com/`
[2] `https://www.cachet.dk`

Preface

This M.Sc. thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a Master of Science in Engineering degree in Computer Science and Engineering.

Kongens Lyngby, January 2, 2022

János Richárd Pekk (s192617)

Acknowledgements

First and foremost, I would like to express my sincerest gratitude to my supervisors Alban Maxhuni and Jakob E. Bardram for their priceless guidance not just trough the thesis, but through my entire time at Copenhagen Center for Health Technology (CACHET). Next to them, I would also like to thank my entire family and friends for the endless support throughout my entire studies.

Contents

CHAPTER 1

Introduction

This chapter provides an introduction to the use of wearable devices to harness human behavior from commercial and open source frameworks. It also explains the problem statement, the research goals and methods, and finally the impact for innovation and application.

1.1 Background

"This is really an exciting result. It shows that passive, continuous monitoring with devices like Fitbit, Garmin and the Apple Watch might turn out to be an important public health surveillance tool for COVID-19."

Eric Topol, MD
Founder and Director, Scripps Research Translational Institute

Due to the increasing demands and urgent needs in the global healthcare sector, the healthcare industry has constantly changed and transformed over time. Since the beginnings of modern medicine in the 18th century, healthcare has become dependent on technology due to regular demands, and at the beginning of the 21st century, healthcare has become more and more dependent on technology [3]. Rapid growth in medical knowledge, need for informed decision-making, unavailability of informatics tools, rising cost of healthcare, shortage of medical professionals, financial unsustainability of healthcare systems, patient empowerment and democratisation of care were some of the driving forces that highlighted the immediate need for a paradigm shift in healthcare and led to looking at digital technologies and disruptive innovations in healthcare [4].

Until recently, the study of human behaviour has been hampered by the ability to accurately quantify its components. However, technological advances in wearable devices and smartphones are increasingly facilitating the collection of large amounts of multimodal data in an unobtrusive and seamless manner. In particular, the use

of data passively generated by these devices enables the scalable measurement of human behaviour in the wild. This data can be used for digital phenotyping. Digital phenotyping can be defined as "movement-by-movement quantification of the human phenotype in situ at the individual level using data from personal digital devices" [5]. This new field has already aroused research interest in a clinical medicine. At mental health, for example, the objective, multimodal, continuous quantification of behaviour using proprietary devices can lead to clinically useful markers, which can then be used to improve diagnosis, adjust treatment or develop new intervention models [6, 7]. Similarly, real-time feedback in conjunction with with artificial intelligence (AI) models enables new possibilities for health and wellbeing applications. For example, it could be possible to to develop personalised intervention feedback that is automatically generated based on based on physiological, environmental and social cues from mobile and wearable devices [8].

The decreasing cost and increasing capabilities of sensors embedded in mobile and wearable devices, and the increasing number of data sources from social media, environmental factors, and other sources, have new concepts and techniques for quantifying well-being, mobility and social interaction [9]. Copenhagen Center for Health Technology (CACHET) is already developing a platform to collect data from smartphones and plans to expand its current system to include wearable technologies such as Fitbit[1] and Garmin[2] smartwatches. To do this, they need software that is able to collect data from third-party vendors, convert it to their own "*data points*" format, and store the converted data in their database. The goal of this thesis work is to design and implement a solution that meets the needs and fits into the existing ecosystem.

1.2 Prior Work

After cloud computing and its innovations such as instant data access and high availability dominated the market, it opened up new opportunities for researchers to explore and combine with the latest technologies such as wearable and mobile devices, making real-time remote patient monitoring and data collection widely available. Much research has been conducted in the field of Mobile Health (mHealth)[3] and the innovations it could bring to modern healthcare. One problem in this broad topic is particularly relevant, namely the integration of heterogeneous data from many different devices[10].

There are a considerable number of commercial solutions on the market that solve the problem of integrating different devices into one's system, such as HumanAPI[4] and Validic[5], which reduce the burden of managing the different authorisation mechanisms

[1] https://www.fitbit.com
[2] https://www.garmin.com
[3] `https://en.wikipedia.org/wiki/MHealth`
[4] `https://www.humanapi.co/wearables-network`
[5] `https://www.validic.com/solutions/`

and data processing of different devices, such as FitBit[6] and Garmin[7], and provide an API for accessing patients' health data. Further, there are also open source projects that aim to solve the same problem, but may have a smaller number of supported devices and platform features compared to their paid counterparts.

In this line, SHIMMER[8] provides a platform for collecting health data from popular third-party APIs such as FitBit and converting the collected data into an Open mMealth-compliant format. In addition, AWARE[9] is an open source framework for collecting sensor data on Android[10] and iOS[11] mobile devices and also has plugins that work with Fitbit devices.

Similarly, RADAR-base[12] is also a data collection platform for mobile and wearable devices. It provides active data collection through surveys on mobile devices in addition to passive data collection solutions for wearable devices. It is a ready-made platform with a scalable architecture that researchers can deploy on their own servers and start their own projects. It also has its own Fitbit integration.

1.3 Problem statement

As it can be seen from the previous section, there are a handful of commercial and open source software available to solve the problem of collecting data from various types of third-party sources. However, apart from the closed, commercial alternatives, the majority of available solutions are ready-made products with established ecosystems that severely limit customisability in certain aspects. From CACHET's perspective, none of them meets their exact needs, because the solutions are either too heavy-weight and force the user to use their entire platform or there is no option to customise the data format the collected data will be transformed to. On the other hand, CACHET's unique requirements will not be present in other cases, therefore the final solution should be generic enough that it could be reused by other individuals to satisfy their problems. To address these shortcomings, the following problem statement can be formulated:

How can a software be designed that collects data from various Web APIs, while it offers customisability and extensibility?

This question can be broken up into the following subproblems.

- How can the software be designed that capable of collecting data from various wearables?

[6] `https://www.fitbit.com/`
[7] `https://www.garmin.com/`
[8] `https://github.com/openmhealth/shimmer`
[9] `https://awareframework.com/`
[10] https://www.android.com/
[11] https://www.apple.com/ios/ios-15/
[12] `https://radar-base.org/`

- How can the design offer customisability/extensibility?
- How can the solution solve CACHET's unique requirements?

1.4 Research goals and methods

The main objective of the project is to develop a system that meets the requirements defined in the previous section and to provide software that overcomes the weaknesses of the existing solutions. The following points explain the questions posed in the Problem Statement section and provide a way to verify their correctness and fulfilment at the end of the project.

1.4.1 How can the software be designed that capable of collecting data from various wearables?

One of the main tasks is to recognize and handle the different access requirements of the various Web APIs. The protocols used for authorization/authentication to protect the sensitive data differ from each other in many ways, so the solution must be able to handle their unique requirements and provide the same functionality for each case, such as user authorization and protected resource access. In addition, almost every data provider offers a different set of data that can be queried through its Web API, and even if they have the same data type, their presentation formats differ significantly, which means that each data type must be handled in a different way by each organization. In summary, the framework must be able to handle different authorization protocols and the capture of different data types. Verifying that this requirement is met, the project should be carried out with different types of approved/authoized wearables and validate that the implementation can handle the different devices simultaneously and retrieve the data over time.

1.4.2 How can the design offer customisability/extensibility?

Previously introduced softwares have already solved the issue of collecting data from wearables, however, the customisation is heavily limited in each case. This statement will be further discussed in the following Related Work Chapter 2. In order for this project to propose unique value, the solution must be easily extensible and customisable.

On the one hand, the main purpose of extensibility is to create the possibility of integrating additional wearable devices into the framework over time. The design of the framework should aim to ensure that the extension requires the least amount of work for developers and minimal changes to the existing software. To achieve this, even if the main CACHET requirements are Fitbit and Garmin, more than two

devices should be analysed and their unique requirements identified to find out how a generic solution could provide a way to meet most of them. The comparison of the wearable devices is described in section 3.2. The verification of this function is done by integrating Dexcom and Withings devices into the system to determine the amount of work needed to add new devices to the system.

On the other hand, the solution should not focus exclusively on solving the problem of data collection from the CACHET point of view. It should remain universally applicable and at the same time offer possibilities to adapt it to one's own needs. One of the most important aspects of this is that the user of the software must be able to choose the technologies that best suit their needs. For example, the project will store information in order to function properly, such as authentication information. Persistence technology must not be hard-wired. It should also be up to the user to decide how to convert each retrieved data type into a different format and what to do with the converted data. One of the tasks of the solution is to provide convenient and flexible ways to customise these aspects with different implementations. This is reviewed using the example of CACHET by implementing its requirements and outlining how these implementations can be interchanged with other types of implementations.

1.4.3 How can the solution solve CACHET's unique requirements?

CACHET shall be able to register users, retrieve and convert their data into the required format, and store the converted data in its main system. The system needs to collect data about the user over time, so the system needs to be able to update authentication information when possible. The solution must be customised in a way that meets these requirements and also fits into the organization's existing ecosystem. This is verified by deploying the implementation on the company's servers, connecting it to the company's infrastructure, and conducting a study that captures users in the way they want and stores the data converted to the format they want.

1.5 Impact - innovation and application

Smart wearable devices offer the possibility to remotely and continuously monitor patients anywhere in the world. With proper analysis of data collected from humans or machines, various health problems could be detected and treated before the first symptoms appear. This project aims to solve the need for data integration in such a scenario. Its purpose is to collect data from devices and convert it into a unified format so that it can be analyzed later. In addition, the framework is intended to be generic and easily extensible to include newer healthcare devices in the future. Developers will be able to leverage the project and develop their own project or use the implementation provided to CACHET to integrate Fitbit and Garmin devices into their CANS[13] system. Great emphasis is placed on extensibility and adaptability, i.e.,

[13] `https://cans.cachet.dk/`

genericity, as developers should not be constrained by technological choices made by today's standard that may not be valid in the future.

1.6 Thesis outline

The following chapters will describe how a software was designed and implemented that resolves the question stated in the Problem Statement. The solution was given the name *Gardener*, so later the project will be references as this.

First of all, the next chapter will give a more detailed look at the existing technologies introduced in Section 1.2 and identify the common problems this thesis will try to solve. Furthermore, the Analysis Chapter will formulate the requirements and explain technological choices made to meet those. After the technologies are chosen, the Design Chapter will show the architecture and software components with various UML[14] Diagram illustrations. The following Implementation Chapter will build upon the Design and explain how the different components were implemented. By the end of the Implementation, a completed solution should be established and the Evaluation Chapter will introduce how the software was tested. Lastly, in the Discussion Chapter the questions formulated previously will be taken to the foreground again and a reasoning will be given how those are solved by the project. In the end, the Conclusion will give a summary about the thesis.

Furthermore, the Appendix will contain useful information about the project, such as a deployment guide, which details how the software should be deployed to a web server and how credentials should be acquired from different wearable providers and moreover An API documentation that describes the currently available endpoints in CACHET's implementation.

[14] https://www.uml.org/

CHAPTER 2

Related Work

This chapter discusses some projects that aim to solve similar problems to those encountered in this project. A brief description is provided for each of the frameworks and their architectures. Finally, their advantages and disadvantages are outlined, and a comparison between the existing solutions and the current project is drawn at the end.

2.1 SHIMMER

Shimmer [1] is an application that allows collecting data from a handful of third-party Web APIs such as **Fitbit**, **Withings** and many more. Its main goal is to convert the different types of data into a unified Open mHealth format. The project is fully open source and comes with a deployment script using Docker.

First of all, the architecture of the project is modular and consists of three main components: Shims, Resource server and Console. Shims are libraries that can communicate with third-party APIs. Authorization, data collection, and conversion to an Open mHealth-compliant format are performed in these packages. A MongoDB[2] instance is used to store authentication information needed to contact the APIs. The resource server is responsible for providing an API to query the ”data points”. Data points are the data collected by the third-party APIs and converted into Open mHealth compliant format. The storage of these data points is the responsibility of the user; the software provides only the query logic. The console is a web-based user interface that helps interact with the resource server.

Moreover, the way the wearable devices are integrated into the system are worth studying. On an authorization level, OAuth1 and OAuth2 based APIs are supported. These are modelled in an abstract way that means the extension of the software with new APIs is possible. Furthermore, the Fitbit implementation supports an impressive amount of data types: body weight and mass, heart rate, physical activity, step count and sleep. However, there is no polling mechanism or a subscription service present to pull data. To initiate data collection, it must be requested by making a HTTP

[1] https://github.com/openmhealth/shimmer
[2] https://www.mongodb.com/

request to a specific endpoint.

All in all, Shimmer is a great tool for managing third-party libraries. It only requires minimal configuration, and you can start collecting health-related data with many wearable integrations. However, it is a ready-made software that lacks extensibility in some aspects. MongoDB is hardwired into the software to manage status and authentication information. To access the collected information, the user must query the Shimmer API. Furthermore, data conversion is limited to the pre-configured Open mHealth conversion and does not allow for any customisation.

2.2 RADAR-base

RADAR-Base is an open source platform for data collection and management. It has a plugin-based, scalable architecture that relies heavily on Apache Kafka[3] and the Confluent platform[4]. The claim of scalability is backed up by the use of Kafka to process real-time data streams. Apache Kafka is an open-source distributed event streaming platform widely used in the industry to implement high-performance data pipelines. Kafka connectors represent the plug-in nature of the architecture and facilitate the connection of new data sources or consumers to the existing Kafka cluster by serving as a bridge to connect and copy data from other systems to Kafka. Figure 2.1 shows their complete architecture.

As for the data collection capabilities, the platform has data collection methods: passive and active collection. First, active data collection is based solely on an An-

[3]https://kafka.apache.org/
[4]https://www.confluent.io

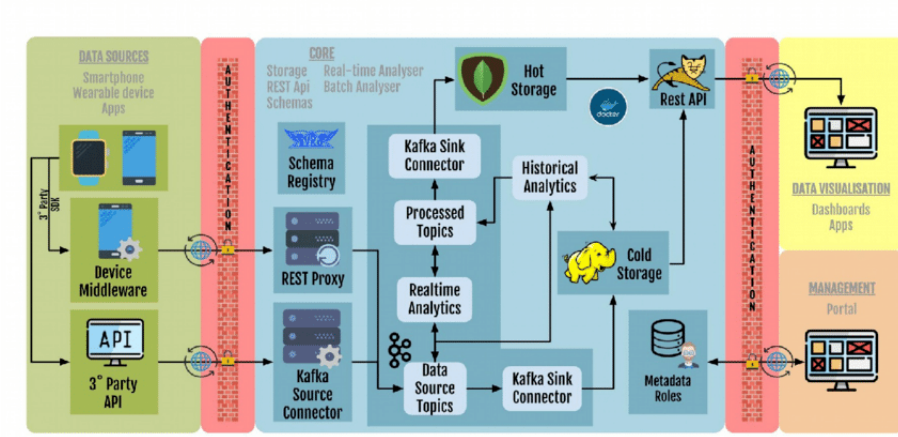

Figure 2.1: Architecture of RADAR-base[5].

droid/iOS application called "*Questionnaire App*". Its purpose is to provide surveys to the user and store the results. Written and spoken questionnaire options are supported. Second, passive data collection consists of two main parts: background sensor data collection on smartphones and data collection from a handful of wearable devices. Sensor data collection is currently available in a stable version for Android, and work has already begun to extend it to the iOS platform. Motion and position sensor data collection is supported, but in addition, various data about the user's interaction with their device can also be collected. As for wearable devices, the software supports a total of six devices, of which only four are continuously supported. **Fitbit** and **Gamin** integrations are available, but the selection of supported data types for Fitbit is rather poor. Only step, heart, and sleep data are available. In contrast, Garmin's offering is sufficient.

The official RADAR-base Github page[6] lists a variety of projects, among which is one of particular interest from the point of view of this project, the REST connector[7]. Its goal is to create a *Kafka Connect source connector* for a generic representation of a third-party REST source that can be used to retrieve data. Once collected, the data is published to existing Kafka themes. The implementation for data retrieval is a polling solution where the actual polling rate is configurable.

In summary, RADAR-base is a completed project that provides data collection capabilities for mobile phones and some portable devices that come with their own smartphone application. The solution has a well-designed architecture that relies heavily on Kafka, but the plug-in based feature offers a degree of extensibility thanks to the use of Kafka Connectors. Therefore, the integration of new wearables is possible, but the implementation must comply with the requirements set by the connectors. Furthermore, the Fitbit integration is tightly coupled to the requirements of the platform, although the abstract REST connector library serves as a good example for implementing a third-party data source. Another shortcoming of the system is that, like SHIMMER, you have to query the RADAR base's REST API to get the data, and there is no way to automatically publish it to external messaging systems. Finally, the data in the system is also stored in Avro[8] formats. While it has its advantages as it is a rich data serialization schema, if the data is required in a different format, the data transformation has to be implemented.

2.3 AWARE

AWARE [9] is an Android and iOS framework for the acquisition of smartphone-specific sensory data. Hardware, software and interaction-based information collection is sup-

[6] https://github.com/RADAR-base
[7] https://github.com/RADAR-base/RADAR-REST-Connector
[8] https://avro.apache.org/docs/current/
[9] https://awareframework.com/

ported by predefined data collection processes. Most modern smartphone sensors are already integrated, and since AWARE is open-source, the platform is constantly evolving by developers to support the latest developments. The framework also includes several plugins, including a Fitbit integration.

This framework has a variety of functions when it comes to notifying another entity when new data is available: Broadcasts, Providers and Observers. Broadcasts allow other applications running on the same phone to be notified. Providers store data directly on the device in SQLite[10] databases or remotely on a MySQL[11] server when using AWARE's Dashboard project. Observers are basically listeners for changes, such as the availability of new data. Personalised subscriptions can be created for these changes, allowing for customisation.

The Fitbit integration [12] is only an Android implementation. There is no generality, it is solely focused on Fitbit integration and only that. It is able to track calories, steps, heart rate and sleep with a granularity of one day. Data collection is done via a scheduled job. The frequency of data retrieval is configurable.

In summary, AWARE is an excellent open-source solution for health monitoring on mobile devices, but is not suitable for primarily collecting health data from other third-party APIs. The Fitbit integration is there, but it was developed mainly for their needs and is not extensively extensible.

2.4 Commercial solutions

So far, only open-source projects have been discussed, but there are many solutions on the market that offer a comprehensive list and high quality of interactions with wearable devices, such as Human API[13], Validic[14] and WeFitter[15], to name a few. They have in common that they operate as companies manner and offer their services for a fee. The source code is not available to the users of the services.

A look at the Human API shows that the company offers integration services for more than 300 activity trackers, wearables and fitness apps, including Fitbit, Garmin and Withings. User authorisation and data collection is handled entirely by the solution. The software has its own dashboard that provides an easy way to connect users and share their health data. The collected data can be accessed via the RESTful API. Through the API, the service is deployed in their ecosystem, the lifecycle is managed

[10] https://www.sqlite.org/index.html
[11] https://www.mysql.com/
[12] https://github.com/denzilferreira/com.aware.plugin.fitbit
[13] https://www.humanapi.co/wearables-network
[14] https://validic.com/solutions/
[15] https://www.wefitter.com/en-us/

by them and the data is stored on their servers.

Similarly, Validic also offers integration with the previously mentioned wearable devices and additionally supports Dexcom. The only difference with the Human API is the selection of wearable technologies available. They also host their services and provide an API for data retrieval.

Finally, WeFitter competes with the previous entries by offering the same services but with an additional aspect, namely the gamification part. For the gamification part, there is a dedicated API with the aim to increase user engagement and health data collection. An example: A game could be one of the individual challenges offered by the company. Users have an individual goal to reach before others, or a time frame, and their achievements are compared. This part alone helps WeFitter stand out from the crowd and offer unique value.

Ultimately, a single thesis project cannot match the functionality of a commercial platform backed by an entire company. But what they have in terms of functions, they lack in terms of customisation options. All of the aforementioned projects are closed ecosystems, meaning that the user has no way to expand or customise their services.

2.5 Summary

In this section, the aforementioned projects are compared in terms of their functional scope, extensibility and customisability. The commercial solutions are left out, as they offer unrivalled support for wearables, but can neither be extended nor adapted to the specific requirements of the user.

Starting off with **SHIMMER**:

- *Feature set*: They have support for Fitbit, Google Fit[16], iHealth[17], Misfit[18], Runkeeper[19] and Withings.
- *Extensibility*: The impressive abstraction of the OAuth1 and OAuth2 protocols makes it easy to implement new wearable technologies, making the software highly extensible.
- *Customisability*: However, in terms of adaptability, the project was developed with the synchronous Spring Boot[20] framework, so it is not possible to change

[16] https://developers.google.com/fit/?hl=en
[17] https://ihealthlabs.com/
[18] https://www.misfit.com/
[19] https://runkeeper.com/cms/
[20] https://spring.io/projects/spring-boot

technology without rewriting the software. Furthermore, MongoDB is hard-wired, the database technology is not interchangeable. Furthermore, the data is strictly converted into an Open mHealth compliant format, which is also not interchangeable. As a final note, SHIMMER is delivered as a complete solution with containerisation using Docker and is therefore easy to deploy and use. However, the data is retrieved by querying the API. All in all, there is a lack of customisation options.

Secondly, **RADAR-base**:

- *Feature set*: Support is given for Fitbit, E4[21], Faros[22], Biovotion[23] and Garmin. Apart from the available wearable technologies, they offer a complete solution with their own infrastructure and mobile applications for data collection.
- *Extensibility*: This can be achieved with Kafka Connectors. An abstraction for OAuth-based APIs has already been developed, therefore the integration of new APIs is possible.
- *Customisability*: Their biggest disadvantage is the technological dependencies and the complicated infrastructure shown in figure 2.1. The components there are not interchangeable, customisation with arbitrary dependencies is not possible. The focus is rather on providing a platform for conducting studies and collecting data from many different sources, not only from portable technologies. As for the data, just like SHIMMER, it is stored internally and has to be queried via the API.

Lastly, **AWARE**:

- *Feature set*: They don't have premium support for wearable devices, the interactions are developed by the community. At the moment, only one Fitbit integration is available. Their purpose is to collect sensory data from smartphones rather than collecting data from web APIs for wearables. AWARE also offers its own platform for conducting studies.
- *Extensibility*: The platform itself is expandable, just as the Fitbit integration was done.
- *Customisability*: They are too integrated into their own technology stack and offer no way to change the dependencies, let alone the format of the data they convert third party data into.

As could be seen, although the solutions offer possibilities to extend them with new additions, they are severely lacking in adaptability. Apart from RADAR-base

[21] https://www.empatica.com/en-int/research/e4/
[22] https://www.bittium.com/medical/bittium-faros
[23] https://biofourmis.com/

and AWARE, which are too heavyweight and offer a complete platform for data collection, SHIMMER is the main competitor and the only one whose main purpose is to meet the data collection needs of wearable technologies. This thesis project aims to address the previously mentioned shortcomings and provide a solution that solves the problem of data collection from third-party APIs while remaining technology-independent, lightweight and providing options for customisation of data processing. If these claims are ultimately met, the project could be a serious contender in the field of health data management solutions.

CHAPTER 3

Analysis

This chapter explains the requirements and then provides analysis of the related fields.

3.1 Requirements

Before the analysis is conducted, functional and non-functional requirements need to be detailed. The first two subsections introduce the general requirements and the third one details what CACHET requires.

3.1.1 General functional requirements

- **User authorization**: The Gardener must be able to provide a way for users to authorize the application to collect data on their behalf from the third-party Web APIs.
- **Data collection**: The Gardener must be able to collect the data from the Web APIs.
- **Data transformation**: The collected data needs to be transformed into a specific format.

These items are the most important functionalities the software must provide. However, the *User authorization* and *Data collection* points need to be narrowed down. There are numerous authorization/authentication mechanisms applied nowadays and not necessarily all of them are utilised in Web APIs, where a delegation based authorization protocol is required. Therefore, only certain protocols need to be supported in the project. Additionally, data representation can also be done in many different ways. An analysis in the field of Web APIs in Section 3.3 and Wearable technologies in Section 3.2 reveals that there are prevalent technologies and trends used widely in the industry. Thus, those aspects should be considered.

3.1.2 General non-functional requirements

- **Customisability of technological dependencies**: The Gardener needs to be designed in a way that the technologies are interchangeable.

- **Customisability of data publication**: The handling of the transformed data needs to be customisable.

- **Customisability of data transformation**: Custom data transformation logic should be able to be configured.

- **Extension with more devices**: Developers should be able to extend the framework with additional wearable devices.

As it can be seen, a high emphasis has been put on the customisability property. To achieve this, a proper software architecture needs to be chosen. The choice is detailed in Section 3.4.

3.1.3 CACHET specific requirements

As discussed briefly in Section 1.1, CACHET requires a solution for data collection from wearable devices. Their main requirements were mentioned in Section 1.4.3 and those are mainly fulfilled by the general requirements. However, the following points detail what customisations are necessary for the application to meet their unique needs.

- **Data transformation into the "*data points*" format**: *Data points* is a special data format CACHET uses to unify and store heterogeneous data. The collected data needs to be transformed into this format before it is published.

- **Data publication to RabbitMQ**: The transformed data needs to be published to the RabbitMQ instance their main system connects to. That will be the connection between the two components.

- **Fitbit and Garmin integration**: Fitbit and Garmin integration excepted to be ready by the end of the project.

- **Adjustment of their main system**: Their main system, Copenhagen Center for Health Technology Research Platform (CARP), needs to be adjusted to accommodate the new functionality. The integration itself is detailed in Section 4.5.

The customisation of the Gardener is further discussed in Section 3.5 and 3.6 as they describe how the project can fit into their existing ecosystem.

The following Sections provide an insight to the before-mentioned topics and outlines the technologies chosen to design and implement the project.

3.2 Wearable devices

First, some of the major wearable technology vendors are presented in detail. Analysis of these vendors reveals common patterns they all follow and unique requirements where they differ. CACHET's main requirement was to integrate Fitbit and Garmin. However, in order to understand the common needs and thus design a solution that can be extended with new technologies in the future, some other vendors are also described in detail. To make the discussion more efficient, the vendors are characterized according to the following categories: Authorization Protocol, Data Collection, and Notification System. Regarding the authorization protocol, each vendor uses a version of Open Authorization (OAuth). An in-depth analysis of both versions is provided in the Appendix Chapter B. In addition, the Web API and its requirements in data collection will be addressed. Finally, the notification system will address whether the company has a system that can notify the application when new data is available for download or whether such a service does not exist.

3.2.1 Fitbit

The first of the bunch is Fitbit. The company makes many different smartwatches. Each watch connects via Bluetooth to a companion mobile app that uploads data to the company's web server. The data can be accessed through the company's API[1]

Authorization protocol: Fitbit uses the OAuth2 authorization protocol and implements the following flows[2]: Authorization Code Grant Flow, Authorization Code Grant Flow with PKCE, and Implicit Grant Flow. Their implementation of the protocol follows the specification and does not add any additional requirements. The only thing they extend is the *Access Token response*, which includes the ID of the user in Fitbit's system.

Data collection: Their Web API provides a respectable amount of different data types that can be queried. All of them have a unique data format. Also, the API URL remains the same for all data, however, some of them may have different API versions that are required in the full URI. In addition, it is possible to customize a query by specifying query parameters in the query URI, such as specifying a time period. Finally, the data is presented in JSON format.

Notification system: The company offers a decent subscription service [3]. Subscriptions are available per user, and each subscription can be customized with the desired data types. Creating a new subscription consists of several steps. First, the subscription service must be verified for each new application by Fitbit calling the

[1] https://dev.fitbit.com/build/reference/web-api/
[2] https://dev.fitbit.com/build/reference/web-api/authorization/
[3] https://dev.fitbit.com/build/reference/web-api/developer-guide/using-subscriptions/

registered application with a unique verification code. Second, after successful verification, subscriptions can be created for each user by making an HTTP request to a specific URI. The format of the notification pings can also be seen in the official documentation. This format is unique to Fitbit and includes the collection that was updated by the user and the Fibit-based ID of the user to whom the data belongs.

3.2.2 Garmin

Garmin also offers a variety of smartwatches and uploads data the same way Fitbit does, through a companion app on the phone. The official documentation of their API is confidential and cannot be linked to in the thesis. Their API is only for business use and the access to the protected resources have to be approved by the company beforehand.[4] For this thesis, CACHET's account was used.

Authorization protocol: Interestingly, Garmin uses the OAuth1 protocol. The implementation of this protocol is fully standardized, so no special requirements are needed in any of the authorization steps. Finally, Garmin does not provide the user ID in its system in the token response, so the application does not know it.

Data collection: Data collection is done through their Web API. The base URL remains the same for each data type, but the API through which the data is available may be different. The queries can be customized using the available query parameters. Again, the data is represented in JSON.

Notification system: Garmin's subscription system is different. Previously, there are two types of notification systems: Ping and Push. In the push system, the Web API sends the new data directly to the application through the configured endpoint. The payload contains the actual data and the user it belongs to. On the other hand, the ping system works like a normal subscription-based service. In the developer portal, custom URIs can be configured for each data type, and every time this specific collection is updated by the user, the Web API sends a notification to this URI. The ping contains a payload containing the user's *Long Lived OAuth1 Token*, which can be used to access the resource and the URI under which the data is available.

3.2.3 Withings

Withings offers a variety of smart products in addition to smartwatches, such as sleep trackers and scales. Their products communicate with smartphones, as in the previous cases, and upload data to their cloud via the phone.

[4] https://developer.garmin.com/gc-developer-program/program-faq/

Authorization protocol: Withings relies on OAuth2 for user authorization[5] and they implement the Authorization Code Grant Flow. Their implementation differs from the original specification in one thing. An extra header[6] is required to be present in the HTTP request to request and refresh *Access tokens*. Other than this, just like Fitbit, the *Access Token response* is also expanded with the user's ID on their system and some other values.

Data collection: A high variety of data types are available for each of their devices[7]. They are mostly available on the same base URL and the path has to be modified based on which data is required. However, the HTTP requests made towards their API have to posses certain mandatory headers. These are indicated in their official API documentation. Despite that, the queries are also customisable with query parameters. Their data is represented as JSON.

Notification system: A subscription based service is available[8]. The subscription process involves two steps. Firstly, the application has to get a random number from their Web API. Secondly, with the random number a new request has to be made that will result in the creation of a subscription. These steps both involve cryptographic operations, because the data required in each step is transferred in the query parameter list in the HTTP request. This requires a signature to be secure, which is a SHA-256 hash of the parameters. Furthermore, their notification is unique, but contains the usual field such as the data type that has been updated and the id of the user the data belongs to.

3.2.4 Dexcom

Dexcom offers a wearable devices to monitor glucose. The data transmission to their Web API happens trough a mobile application.

Authorization protocol: Dexcom uses a plain implementation of the OAuth2 Authorization Code Grant Flow[9]. Everything is how the protocol dictates it, even the *Access Token response* contains the bare minimum. The identification of the user in their system is not received.

Data collection: The list of available data types is short. Every data type is available on the same URL and the path is different based on the type. The query is customisable with query parameters. Other than this, there is nothing specific about their API. Their data representation also utilises JSON.

[5]https://developer.withings.com/developer-guide/data-api/authentication
[6]https://developer.withings.com/api-reference/#operation/oauth2-getaccesstoken
[7]https://developer.withings.com/developer-guide/data-api/index-data-api
[8]https://developer.withings.com/developer-guide/data-api/data-update-notifications
[9]https://developer.dexcom.com/authentication

Notification system: Dexcom does not own any kind of notification system.

3.2.5 iHealth

iHealth also offers a high variety of wearable smart devices. The connection and data upload happens through an application.

Authorization protocol: OAuth2 Authorization Code Grant Flow[10], as almost in every other case. According to their documentation, no additional parameters are required during the authorization steps. In their *Access Token Response* the identifier of the user in their system is present, next to another extra fields.

Data collection: Their options of their data collection are also common. However, they offer highest level of customisation when it comes to queries, so far. The concrete list of available parameters are listed in the official documentation[11]. It is also has to be noted, that mandatory parameters ought to be present in the HTTP headers when making a request. Interestingly, their data is available JSON and XML format.

Notification system: Subscription based system is available. The subscription are configured on their portal, the application itself does not have to make any calls towards their API. Once the service is enabled, it will push notification to the application through the registered endpoint. The notification contains the updated data type and the id of the user in their system.

3.2.6 Misfit

As the last analysed provider, Misfit also provides a Web API to collect data about users. The data upload happens through mobile applications. As a side-note, Misfit's documentation[12] seems to be vague and faulty. Their actual API description was during writing this chapter and their website hardly works. This was presented just to see an other example of a subscription service and an overall data provider.

Authorization protocol: As the second representative of providers who utilise OAuth1, Misfit uses OAuth 1.0 instead of OAuth 1.0A, according to their documentation[13]. Even though, the name *OAuth1* is not mentioned on their page. Their documentation is also confusing and short. Presumably, OAuth1 is utilised, yet they use the naming conventions of OAuth2, such as *scopes*.

[10] https://developer.ihealthlabs.com/dev_documentation_Authentication.htm
[11] https://developer.ihealthlabs.com/dev_documentation_RequestfordataofBloodPressure.htm
[12] https://build.misfit.com/docs/cloudapi/overview
[13] https://build.misfit.com/docs/cloudapi/get_started#authAPI

Data collection: Data collection happens the same way as previously. Same base URL and the path decides the resource. JSON format.

Notification system: Subscription service is provided. On their developer portal, webhooks have to be defined, which will be called upon data updates. The notification, next to other fields, will contain the updated data and the id of the user in their system.

3.2.7 Devices overview

As it could have been seen in the previous analysis of six different Web APIs, the following trends can be discovered:

- Every manufacturer uses OAuth version 1 or 2 as their authorization protocol. The state of these protocols in today's landscape will be discussed on Section 3.3 to further back up this claim.
- Most of them follow the protocol guidelines, however, there can be cases when they require unique parameters to be present in different steps, such as in Withing's case. The project has handle the presence of extra parameters.
- At the end of the authorization flow, the *Access Token Response* contained additional fields in the majority of the cases. The project must be able to provide a way to persist the provider specific fields.
- The data collection seemed to be straight forward in every case. The data is available on a specific path on the Web API of the company and the retrieval can be customized with query parameters.
- The data notification services differ in every scenario. Their presence is not guaranteed. Even if they are available, they vastly vary in almost every property. Different ways of creating a subscription and completely different ping notification formats. One common aspect can be noticed, however. In the ping notifications the data that is updated and the user that the data belongs to are always present. These two information correctly points out what needs to be downloaded, thus it makes sense why they are always there. The application should provide a way to download data that is usable with and without the presence of a notification service.

3.3 Web APIs

Prior to this paragraph, Section 3.2 frequently used the term Web APIs, where the wearable device manufacturers made their data available. This leads to the question of what exactly a Web Application Programming Interface (API) is. APIs provide

a way for developers to open up their data and services to the public. Hence, this enables services and applications to communicate with each other and make use of each other functionality through a documented interface[14]. APIs that are available on the public internet and can be accessed using the HTTP protocol are the Web APIs.

To find out more of today's landscape about the state of the Web APIs, it is worth taking a look at the website *ProgrammableWeb*[15] due to its archive containing roughly around 25.000 API descriptions. This set of APIs is also used by a handful of papers that aims to survey the field[11][12]. According to their article[16] from 2017 details, that the most popular API type out of 3,620 APIs is the Web API type with a staggering 82%. Out of those, the RESTful APIs are by far the most popular choice when it comes to architectural styles with 81%. As a brief description, REST is an architectural style for distributed hypermedia systems that was designed to work with the basics of the Web architecture. The most essential representation of a set of data in REST is a resource. It uses resource identifiers (such as a URI) to identify resources during a communication and declares generic interfaces (REST connectors, such as Web APIs) for manipulating the indicated resources[13]. The detailed description of the architectural style is detailed in the cited dissertation.

One more interesting aspect of the available APIs, despite their architectural style, is the authentication/authorization mechanisms they utilise. According to this research[11] conducted in 2019, OAuth based authorization is applied in 72% out of 45 carefully selected APIs. *ProgrammableWeb*'s research[17] showcases a different result where OAuth is forced to the background, however, it is noted that the research involved many legacy API's with outdated security mechanisms.

Last item to mention in this chapter is the type of the data representation the APIs use. *ProgrammableWeb*'s article[18] surveyed the field and the APIs starting from 2018 mainly used JSON as their main data representation format. Recently, JSON was used almost five times more than the second place's data format, XML. The research paper[11] also proves this theory, as 89% of their analysed APIs had support for JSON.

It can clearly be seen that REST accompanied by JSON is the staple of the API space. OAuth is also important when it comes to authentication/authorization,

[14] https://www.ibm.com/cloud/learn/api

[15] https://www.programmableweb.com

[16] https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26

[17] https://www.programmableweb.com/news/spotting-api-security-trends-programmablewebs-api-directory/research/2018/01/02

[18] https://www.programmableweb.com/news/json-clearly-king-api-data-formats-2020/research/2020/04/03

especially when the requirement is identity delegation. This data is significant, when it comes to the decision of what kind of third-party APIs should be supported by the project. In addition, Section 3.2 analysed the wearable APIs and it could have been seen how OAuth prevailed as the main authorization mechanism and JSON as the main data representation format. Therefore, the project can narrow down the support for OAuth based APIs with JSON.

3.4 Software architecture

With the conclusions of the previous sections and the requirements mentioned in the introduction of the chapter, it is known that the system has to support various kinds of wearable devices with OAuth based APIs and JSON data formats, while staying free of technological dependencies as much as possible. With the problem given, the available architecture patterns should be taken a look at in order to choose the most suitable one. The architecture patterns are general *blueprints*, which give solutions for commonly occurring problems. There are a handful of established patterns[19] available. Detailing each of them is out of the scopes of this project.

As mentioned in the beginning, the software has to solve the integration of wearable device APIs. When it comes to the functional requirements in Section 1.4.3, most of them can be considered event-based. During the authorization, the user will be redirected to the third-party website to give consent. At some point later in time, the application will be called with the result and it has to react to this *event*. When the authorized users eventually upload some data, the application will be called with a payload specifying the details of the upload and it has to react to this *event* accordingly. In addition, the process of enrolling a user and collecting data are separate concerns, which means they could be decoupled. Therefore, a **Publish-Subscribe/Event-bus** pattern could be a good choice, as it is one of the most used patterns[14] and could satisfy the requirements. With the event-based approach, there would be a central agent acting as a mediator, providing a communication channel for different components of the system. Using the central agents, components could subscribe for different *events* defined by the software. Once those events happened, the subscribers would be invoked to handle it. In summary, the Event-bus pattern provides loose coupling between the components due to the communication happening trough the central agent and also ensures extendability, because arbitrary subscriptions could be made by for different events. These benefits are gained at the cost of more complicated implementation, since the events and subscription handling with the central agent also have to handled.

[19] https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013

Apart from the advantages that the Event-bus pattern brings, it does not in itself solve technological independence. It is important that the technologies are not hard-wired into the software and the user could choose them and inject their own choices. Therefore, the software should define generic functionalities which later could be customized with the user's own components. This requirement exactly matches the definition of **Software frameworks** on Wikipedia[20]. By following an abstract software framework design and relying on the use of interfaces instead of actual implementation, the project stays free of dependencies and it can be made sure that those interfaces could be substituted with components that perfectly meets the requirements of the current user. Furthermore, the Hexagonal architecture pattern[21] could also be utilized when developing the framework. As Netflix's blog describes it[22], the pattern dictates that the business logic must not depend on the technology used. Therefore, the technologies could be changed without impacting the code-base. This is exactly what is needed to achieve complete technological independence of the framework.

However, the framework itself is not a ready-made solution, its required interfaces need to be implemented. By adopting the Event-bus pattern and following the software framework, there are numerous possibilities for implementation. On the one hand, the framework could be implemented as part of an existing monolithic software. Furthermore, it could also be implemented as an independent microservice. On the other hand, the Event-bus allows the authorization and data collection to be split into two separate modules. From then on, it is up to the user of the framework to decide whether to implement them together, split them into different services or run one as part of a monolith and the other as a standalone service.

3.5 CANS Implementation

The implementation of the framework aims to fulfil all the dependencies that the framework needs to be operational. As per CACHET's requirements, a solution should be in place where data collection from the portable APIs is possible and the collected data can be transmitted to their existing system. There are a handful of ways in which this could be implemented. First, the framework could be integrated into the existing monolithic application, provided that it remains truly free of any hard-wired technological dependency. In this case, persistence is through the existing database and data is pulled directly into the system, so no transmission of the collected data is required. Consequently, the existing database schema will need to be modified to meet the requirements of the framework, and the code base will also need to be adapted to integrate it. In addition, the framework and programming

[20] https://en.wikipedia.org/wiki/Software_framework
[21] https://en.wikipedia.org/wiki/Hexagonal_architecture_(software)
[22] https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749

language are chosen, namely Spring Boot[23] and Kotlin[24]. This solution is viable, but requires a lot of changes to the main system and leaves little room for customisation and experimentation.

As a second solution, the framework could also be implemented to function as a completely independent microservice. In this approach, with the exception of the RabbitMQ message broker, the application is not restricted by the technologies used in the main system. Thus, the web framework, programming language, database technologies, etc., can be freely chosen. The only real restriction is that the project must be able to transmit the collected wearable data to the RabbitMQ broker. The main system remains untouched and nothing changes from its point of view, as it is already able to process and store data from the broker. In the microservices-based approach, the following three aspects of the system should be selected: Programming language, web framework and persistence technology. As mentioned earlier, RabbitMQ as a message broker is a prerequisite.

3.5.1 RabbitMQ

Before explaining the technical choices/possibilities, RabbitMQ should be briefly introduced. RabbitMQ is one of the most widely adopted open source message broker, implementing the Advanced Message Queuing Protocol (AMQP) protocol[25], but in addition other protocols are also supported. Its message distribution, according to the AMQP protocol, is based on *exchanges* and *queues*. Messages are published to exchanges, which distribute copies of the messages to the *queues* using certain rules, which are called *bindings*. From the queues the messages will either be delivered to subscribers or fetched by them[26]. Using this communication protocol, the project will be able to communicate with the main system through the broker if there is a queue that is being observed by it. Nevertheless, the broker itself proves to be reliable, however, its scalability is questionable[15]. If it will ever be replaced due to its weak point, only the RabbitMQ specific implementation has to be changed, the other parts of the project will not be affected thanks to the hexagonal architecture, detailed in Section 3.4.

3.5.2 Programming language

Firstly, the **programming language**/development environment has a decent amount of options available. According to *Developers Nation*'s[27] latest report[28], which details the current trends in the developer community from 2021. The chart from the report

[23] https://spring.io/projects/spring-boot
[24] https://kotlinlang.org/
[25] https://www.amqp.org/
[26] https://www.rabbitmq.com/tutorials/amqp-concepts.html
[27] https://www.developernation.net/
[28] https://slashdata-website-cms.s3.amazonaws.com/sample_reports/_TPqMJKJpsfPe7ph.pdf

is attached on Figure A.10. While JavaScript based languages are the undisputed kings, other technologies, such as Python, are catching up rapidly. The third place is taken by Java, one of the most significant general purpose languages. Event tough Java has been around for almost 20 years, its community is still steadily growing, according to the report. CACHET's main system is developed in Kotlin, which is a close relative of Java and also sits among the top ten on the list. Kotlin is a fairly new programming language, first launched in 2016 and developed by JetBrains[29]. According to their report[30] of the current technology landscape of 2021, Kotlin is among the top five fastest growing languages. It can be considered as the modern version of Java by introducing a whole new syntax that does not require writing huge amount of boilerplate code Java is infamous for and by bringing modern programming paradigms[16]. In addition, Kotlin can be combined with Java code. Due to the increasing popularity as well as the above-mentioned advantages of Kotlin and based on the experience gained so far, we have decided to write the Gardener framework using the Kotlin in the context of this thesis work.

3.5.3 Web Framework

Secondly, a **web framework** is necessary to being able to handle HTTP requests. In CACHET's example, Spring Boot serves this purpose. Spring would be a perfectly valid choice for implementation, its extensive ecosystem of plugins mostly cover most of the common concerns that can come up during development. However, Spring Boot is a synchronous framework. Probably the most significant drawback of the synchronous application in today's world is the fact that they block until a certain operation is completed. It is especially prevalent in I(input)/O(output) heavy applications, such as web applications, where the number of blocking operations, such as network calls and database operations, is significant. Blocking threads while waiting for long running I/O operations wastes resources and can take its toll on the scalability of the applications. If all threads are occupied by blocking operations, additional requests cannot be served, resulting in latency and poor usability. To counter this problem, asynchronous computing aims at *eliminating* the blocking part. When a blocking I/O operation occurs, instead of blocking the current thread, it can move to perform other tasks and the initial task will resume once the blocking operation is completed. This is done through the use of callback functions. These functions are configured to be invoked when the blocking operation is completed. On the other hand, writing asynchronous code is more complicated than its synchronous counterpart. Deeply nested callback functions can easily lead to *callback hell*[31], which makes the code hard to comprehend and maintain, thus leads to unintended bugs. However, the positives outweigh the negatives and the asynchronous programming model can lead to great scalability benefits[17]. In the Java Virtual Machine (JVM) ecosys-

[29] https://www.jetbrains.com
[30] https://www.jetbrains.com/lp/devecosystem-2021/
[31] http://callbackhell.com/

tem there are plenty of asynchronous web frameworks, such as Spring Webflux[32] and Vert.x[33]. Out of the two mentioned projects, both of them have an extensive amount of plugins and great documentation and both of them are capable to satisfy the requirements of CACHET. Both have their own disadvantages and advantages, and neither can be superior to the other. Thus, Vert.x was chosen to implement the project purely out of the desire of learning new technologies.

3.5.4 Database Technology

Lastly, a **database technology** has to be chosen, as the application requires the storage of data to manage the authentication formation for each user. There are a wide variety of available database technologies one can choose from, however, the first decision is the question of relational or non-relational databases. Relational databases have been around from the 1970s and still remained one of the primary choices for data storage due to their powerful and well-established Relational Database Management System (RDBMS) and Structured Query Language (SQL). RDBMS systems store the data in tables consisting of columns in a normalized form[34] in predefined, strict schema and defines clear relationships among them. RDBMS systems usually honor the ACID principles[35]. Nevertheless, today's requirements demand, as more and more people have access to the internet and generate data, scalable solutions that are capable of storing and retrieving huge amount of data in an efficient way. Relational databases usually struggle with scalability[18], even though there are ways to make them scalable. On the other hand, Not Only Structured Query Language (NoSQL) aims to provide an alternative, which solves the problem of handling big data on a Web scale. NoSQL does not have a formal definition. It represents a a fundamentally different data storage technology compared to the RDBMS systems. Generally, it stores the data in *documents* in a flexible schema and honors the BASE principles. By venturing away from the strict restrictions and locking strategies of ACID[36], with BASE, NoSQL databases are able to achieve scalability by design[19]. The way they store the data and their principles are one of the most significant differences among them. While ACID tries to lock the resources, therefore causing a bottleneck, BASE by relying on the concept of eventual consistency allows a more scalable approach. One more advantage of the NoSQL databases is that no predefined schema is required. This is highly beneficial during the development phase, when the domain model is constantly evolving. Of course, it cannot be stated that one technology is better than the other, it all comes down to the use-case. SQL remains better and handling complex data relationships and complex queries. NoSQL on the other hand, excels at handling a huge amount of data in an efficient way. In this project, complex domain models and need for complicated query logic are not excepted, since the only things

[32] https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html
[33] https://vertx.io/
[34] https://en.wikipedia.org/wiki/Database_normalization
[35] https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions
[36] https://en.wikipedia.org/wiki/ACID

need to be persisted are the authentication data for users and some state information. For the current user-case, NoSQL with its flexible schema handling offers a great solution to the problem. In the batch of technologies that implement NoSQL based data storage systems, there are also a handful of different data storage models, such as *key-value stores*, *document stores*, *graph databases*, etc. MongoDB is one of the most famous[18] and well-performing[20] option in the space of Document Stores space. while it is open-source. Therefore, it is chosen as the database technology for the implementation.

3.6 Deployment

Once the software architecture and the required technologies have been selected, the application itself must be made executable and executable instances of the dependencies must be provided.

3.6.1 Build automation tools

First of all, dependency manager and build automation tool should be selected. Build automation tools are essential for modern software development to automatically manage the life cycle of the software that encapsulates compiling the source code, packaging dependencies, running tests and more. The three most used build automation tools[21] are: Ant[37], Maven[38] and Gradle[39]. Ant and Maven are both XML based configurations and they are limited and inflexible when custom operations need to be defined. Gradle focuses more on improving the shortcoming of the others by providing great options for extensibility[22]. Additionally, its configuration file can be written with Kotlin DSL. One more important aspect that has to be outlined is the need for multi-module builds. The framework and the CACHET specific implementation are two different projects with different dependencies, where the implementation depends on the framework, but not the other way around. Multi project builds make it possible to separate the two and define different dependencies and build logic for each project. All of the before mentioned building tools are capable of handling multi-project builds and each of them is able to handle the requirements of the project. As Gradle being more extensible and customisable, and according to their report[40] outperforms Maven, it will be chosen as the build automation tool for the project.

[37] https://ant.apache.org/
[38] https://maven.apache.org/
[39] https://gradle.org/
[40] https://gradle.org/maven-vs-gradle/

3.6.2 Containerization

Nowadays the two most frequently used techniques to host and isolate applications as well as optimize the use of hardware resources are *virtualization* and *containerization*. The main point of virtualization is that the physical resources of the host computer are divided up and the parts can be utilised by different virtual environments called the Virtual Machine (VM). The VMs define a complete Operating System (OS) independent from the host OS, the application it is supposed to run and necessary libraries, therefore the instances of VMs are usually heavy-weight and takes up a lot of space. Furthermore, the host operating system must have a so-called *Hypervisor software* that allocates the resources to the VMs and manages them.

On the other hand, containerization offers a light-weight alternative to its counterpart. The essential building blocks in this context are the containers. The containers contain the application and its own operating environment. On the contrary, it does not run a completely different OS, the host OS is utilised. Therefore, the resulting containers are much more light-weight than the VMs, as only the necessary dependencies are bundled together. With the *Container engine* installed on the host OS, it can take care of spinning up containers. Not having to handle multiple kernels as compared to VMs, containerization offers to reduce the overhead on deployment times, portability and physical resource usage[23].

When it comes to containerization market, Docker[41] and LXC[42] are the leaders of the industry. Docker originates from the concepts of LXC and improves upon its formula. It is more scalable and creating containers is much more simpler[24]. Docker can also be run on different Host OSs, compared to LXC which can only be run on Linux systems. Docker has a big impact on the industry. Most major softwares have their official Docker images, such as MongoDB and RabbitMQ, which are essential for the project, and can be easily set up. Due to its popularity, ease of use and its overall influence on the industry, Docker will be used to containerize the project.

3.7 Analysis overview

This chapter has analysed the available technologies and principles that can be used to develop software that meets the requirements presented in the first section. In order to allow a high degree of customisation and not impose any technology on the user, it was decided that the development would follow the hexagonal architecture and define an application core that describes the business logic. In this way, the framework can be implemented and customised with technologies that are required for the user's current project. According to the survey conducted in the wearables vendor space, the framework will focus on supporting Web APIs from the beginning

[41] https://www.docker.com/
[42] https://linuxcontainers.org/

using OAuth authorization protocols with JSON Data representation focus. Finally, the core will be written in the Kotlin programming language, as will the implementation.

The technologies and operational strategy for CACHET were also selected. MongoDB is used for the persistence of the data. RabbitMQ is also used as a message broker that connects the project to CARP. The project is developed to function as a standalone microservice, so changes to CARP are kept minimal. The operation will be handled via Docker.

With all the necessary technologies and principles chosen, the next Chapter will explain how the project was designed.

CHAPTER 4

Design

This chapter describes the design process of the framework and its implementation for CACHET in detail and illustrates it with UML diagrams. The chapter contains three types of structure diagrams to illustrate the structure of the software components, namely deployment diagrams, component diagrams, class diagrams and a type of behaviour diagram, the sequence diagram. The class diagrams are presented from the conceptual perspective so as not to bloat the screen with details that would only make the diagrams more difficult to understand. For more expandable details of the source code and functionalities, we provide a fully documented code with JavaDoc[1].

4.1 Framework architecture

First of all, according to the principles of the Onion architecture analysed in Section 3.4, the application core must be set up. The core is the heart of the application as it encapsulates the domain objects and the business logic. Before we look at the domain and its associations, the main components of the core should be highlighted. The Component Diagram 4.1 shows the 3 main parts of the system. The required interfaces are discussed in the corresponding sections of the components.

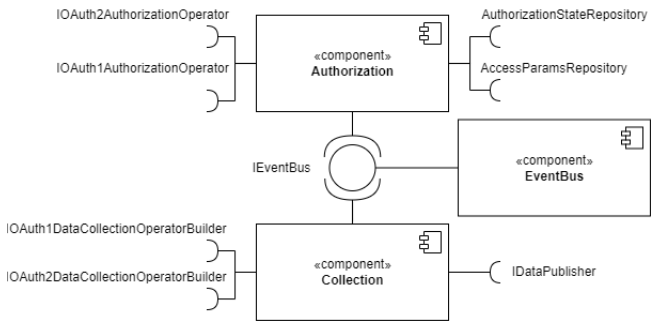

Figure 4.1: Component diagram of the framework.

[1] https://github.com/cph-cachet/carp.gardener

Starting from the top, the first main component is the "Authorization" component. There are three main tasks here: User authorisation, token handling and the initiation of the data collection flow. In each scenario, authorisation can be summarised in two main steps. First, the new user must be redirected to the device's official web page, where the user gives consent to the application to collect data on their behalf according to the OAuth1/2 standard. Second, the third-party Web API contacts the app by calling its webhook and sends data depending on the success of the authorisation process, and the app acts accordingly by either storing the data or handling the errors. When handling tokens, both OAuth1 and 2 use tokens that need to be stored. However, in the case of OAuth2, token refreshment should also be handled as access tokens are short-lived. Furthermore, initiating the data capture flow means that the module collects all the information required for data capture (tokens, client settings, etc.) and passes it to the Collection Module for execution.

Speaking of the Collection Module, once the user is registered and the application has the authentication tokens, the data can be captured by the third-party Web API. This is the main responsibility of the **Collection** component. It should retrieve the data from the provider's API, convert the retrieved data into the specified format and publish the result.

Handling user authorisation and retrieving data are matters that can be well separated. By defining these boundaries, a modular structure is achieved. To ensure a loose coupling between the modules, the component **EventBus** is additionally introduced. It serves as an intermediary in the middle that takes over the message exchange between the two components. The advantages of this modular architecture become apparent during the implementation of the framework. Each of the modules can be developed independently without affecting the other and could be split into its own microservice, they do not need to be coupled together. The only thing they depend on is the EventBus, despite some common concerns that will be discussed later in the chapter. The loose coupling and separation of concerns are also advantageous during development, as changes in the modules do not directly affect the others if the common dependencies are not changed.

4.2 Events

As described in the previous section, the EventBus component plays an important role in the application. Its purpose is to serve as a messaging component. It provides functions to subscribe to various changes in the system and publish the changes as soon as they happen. These changes are the so-called *Events*. The different types of events can be seen in the Class Diagram 4.2.

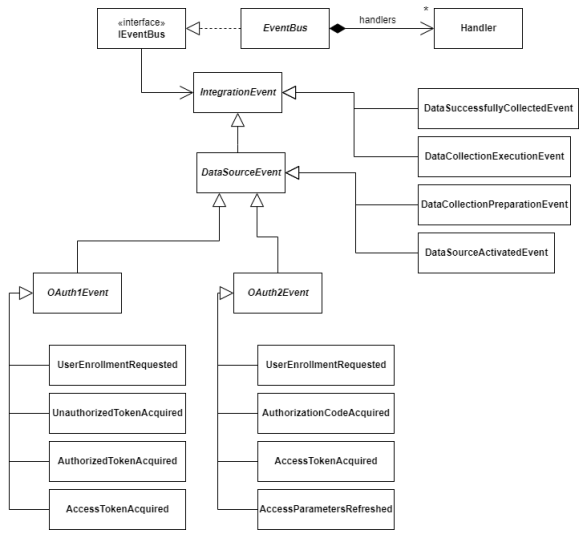

Figure 4.2: Class diagram of the event hierarchy.

Every event in the system is derived from the common superclass, **IntegrationEvent**. It provides common fields, such as an ID and a creation date. The ID field is used to give each event a unique identifier that can be used to provide idempotent event handling [2]. Handling the events in an idempotent way ensures that multiple processing of the same message has the same effect as excepted. The class **DataSourceEvent** is a specialised *IntegrationEvent* as it also ensures the presence of a data source ID field. These events are associated with the specified data source (e.g. Fitbit) and return data that is unique to that specific data source, unlike the *IntegrationEvent* which is a general event type. When subscribing to a *DataSourceEvent*, the desired data source is also specified by its ID, so once such an event is triggered, only subscribers subscribed to that specific event for that specific data source should be notified. For example, during the Fitbit authorisation process, the Fitbit data source is only interested in processing the Fitbit-specific *AuthorisationCodeAcquired* events, rather than processing every event of this type, such as Dexcom or Withings events. In addition, there are also two abstract classes, **OAuth1Event** and **OAuth2Event**, which are there to encapsulate OAuth1 and 2 protocol-specific events. The concrete event types that can be triggered in the system are the descendants of these classes.

The EventBus hierarchy can be seen in the upper part of the image. The interface **IEventBus** provides the functionality to subscribe and publish events and should be used throughout the application to exchange messages. Injecting only the interface

[2]https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/subscribe-events

into the components ensures that it is interchangeable for different implementations. The abstract class **EventBus** implements the interface and provides an in-memory implementation for the subscription functionality, i.e. the subscriptions are stored in memory in the **Handler** classes.

4.3 Authorization module

As briefly discussed in Section 4.1, the Authorization module is one of the three main modules and responsible for user authorization, token management, data collection initiation. In order to elaborate more on how the module provides these responsibilities, a more fine grained Component diagram is displayed by Figure 4.3.

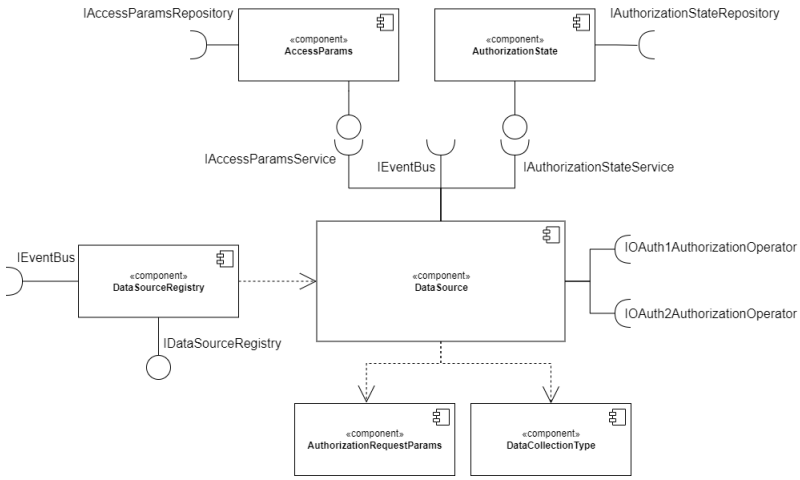

Figure 4.3: Component diagram of the Authorization component.

Starting with the most conspicuous component on the diagram, the **DataSource** defines the whole process of user authorization in case of OAuth1 as well as of OAuth2. It requires three interfaces that are not provided in the Authorization component. The **IEventBus** which is used for messaging. **IOAuth1AuthorizationOperator**, which declares the main steps in the authorization process of OAuth1 that involves communication over the network, such as getting the tokens. Similarly, the **IOAuth2 AuthorizationOperator**, which handles the same thing, just for the OAuth2 protocol. The two operators are not implemented in the framework, it is the responsibility of the developer using the framework to provide an implementation of their choice that fits their needs. Implementing them would have required many different dependencies, such as web clients or cryptographic libraries for OAuth1, which would have limited the flexibility of the framework.The remaining two interfaces that are required

but also provided by different parts of the component are the **IAccessParamsService** and the **IAuthorizationStateService**.

The **AccessParams** component manages the persistence and retrieval of the tokens acquired by completing the authorization process. It requires the **IAccessParamsRepository**, which declares some functions that are needed for persistence. This is supposed to be implemented by the user, just as the operators, to not to limit the users to a specific persistence technology.

The **AuthorizationState** component is used for the management of states in the authorization process. For the capability of the two authorization protocols for state management in-between calls is limited and can vary between implementations, the framework uses this component to save state information. It requires the **IAuthorizationStateRepository** for persisting the state objects and that is not provided by the framework for the same reason as the *IAccessParamsRepository*.

Moreover, the **AuthorizationRequestParams** component, at the bottom of the picture, is being used the *DataSource* to provide customisability for different steps in the authorization process, for the reasons mentioned in section 3.2.7. **DataCollectionType** lists the supported data types of different Web APIs and stores information about them for the Data Sources, so they can use the types when it comes to data specific tasks, such as constructing the URIs where the data can be collected from. Lastly, **DataSourceRegistry** is a storage unit which is used to store activated Data Sources, meaning that the Data Source object (e.g. Fitbit) is instantiated and configured, so they can be retrieved later on by their ID. After the establishment of the major subcomponents and their purposes in the module, the following Class Diagrams will detail the actual classes and their relationships. Each diagram aims to focus on one component and its immediate dependencies to avoid having a complicated structure.

Starting off with the *AccessParams* and *AuthorizationState* components, since their structures are identical, their class diagrams can be beheld on Figure 4.4 and Figure 4.5. **AccessParams** is the main abstract class that will hold the authentication information for one user per Data Source. Its descendants are the **OAuth1AccessParams** and the **OAuth2AccessParams** and they store their protocol specific data. **IAccessParamsService** declares functionality to retrieve/save authentication data for a user and Data Source. **AccessParamsServiceHost** is the main class that realizes the interface and provides implementation. It uses the **IAccessParamsRepository** to retrieve and save the data. Following the principles of the Hexagonal Architecture in Section 3.4, the repository is only used by its service class throughout the application and the service class itself is used to data related operations. Likewise, the *AuthorizatioState* Class diagram follows the same principles. The notable difference is the domain object itself. The **AuthorizationState** abstract class holds state information for a user and Data Source. Its children are the **OAuth1AuthorizationState**

and **OAuth1AuthorizationState**. This hierarchy was made to accommodate the requirement that in case of OAuth1, a *Request Token* and *Token Secret* has to be saved during the authorization process. On the contrary, in case of OAuth2, there is no necessary information to be persisted.

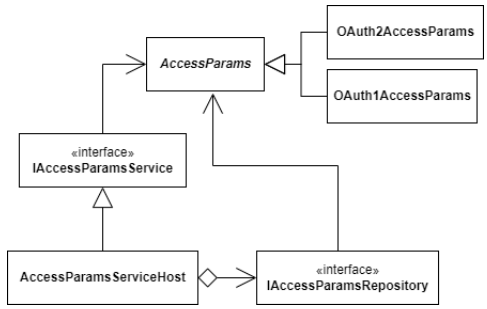

Figure 4.4: Class diagram of the AccessParams component.

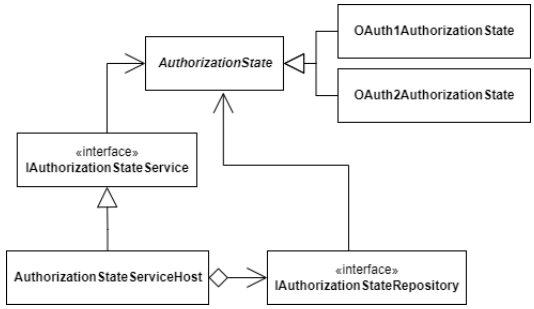

Figure 4.5: Class diagram of the AuthorizationState component.

The separation between OAuth1 and OAuth2 is a common theme through the design of the framework. Even though the share an almost identical name and both of them utilise tokens to access protected resources, they are vastly different. The next representative of this separation is the **AuthorizationRequestParams** hierarchy presented on Figure 4.6. This abstract class' main purpose is to provide a way to customize the authorization requests made towards the third-part API, because different vendors require different parameters to be present in the request. Nevertheless, when one property is required by one of the protocols it might not be mandatory by the other. For instance, in case of OAuth2, the scopes should be present when requesting authorization from the user. Scopes, however, have no concept in the world of OAuth1, thus, that field is only present in the **OAuth2AuthoizationRequestParams**. On

the contrary, **OAuth1RequestParams** has dedicated fields for requesting Unauthorized Tokens, which step is not present in the second version of the protocol.

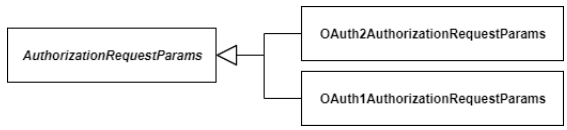

Figure 4.6: Class diagram of the AuthorizationRequestParams component.

Continuing the discussion with the second component that does not provide interface for the main DataSource component but being used by it is the **DataCollectionType** hierarchy, showed on Figure 4.7. This structure represents the supported data types for one Data Source and provides information about them, such as data identifiers and collection URIs. As a practical example in Fitbit's case, one data type would be Fitbit's Heart Rate. Each specific Data Source should declare one class with their data types and the main Data Source will use these classes. The *ThirdPartyData* and *DataTypeTransformer* classes will be discussed later on in this chapter in section 4.4.

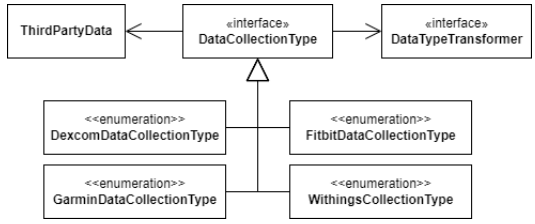

Figure 4.7: Class diagram of the DataCollectionType component.

The last subcomponent before the DataSource is the **DataSourceRegistry** component, presented on Figure 4.8. As stated before, the whole point of this hierarchy is to save and retrieve every concrete Data Source in a polymorphic way. **IDataSourceRegistry** is the interface that declares the functionality to satisfy this dependency and **DataSourceRegistryHost** is the class that realizes the interface and provides implementation with an in-memory storage.

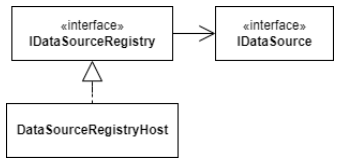

Figure 4.8: Class diagram of the DataSourceRegistry component.

With every subcomponent discussed from the Authorization module's Component Diagram, the highlight is placed on the DataSource component itself. The exhaustive Class diagram is depicted on Figure 4.9.

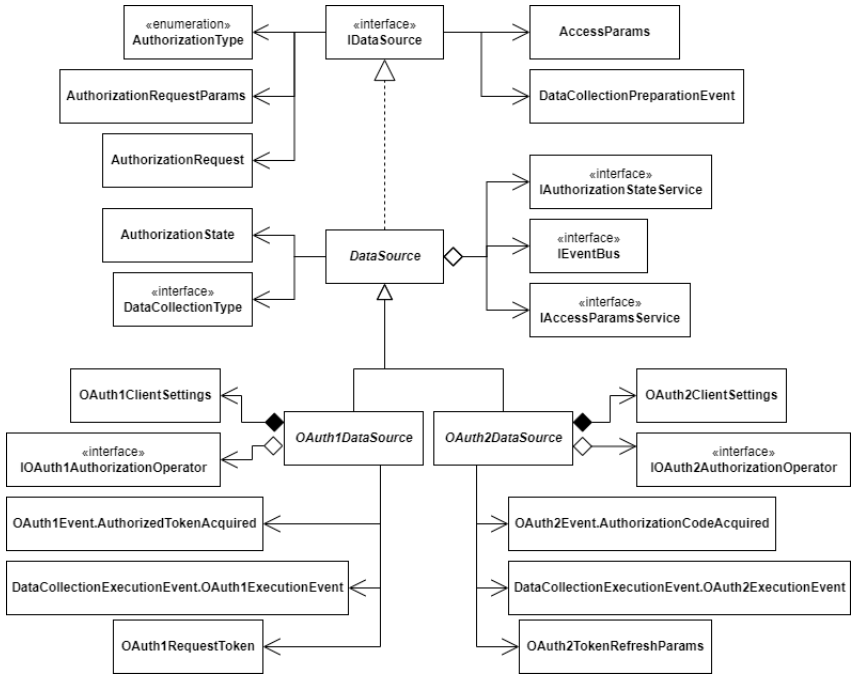

Figure 4.9: Class diagram of the DataSource component.

This is the parent of every wearable device's implementation in the framework. It contains the business logic for both OAuth1 and OAuth2 authorization protocols. Starting the description from the top of the diagram, the **IDataSource** interface declares common functionalities applicable to each Data Source, such as initiating user authorization process, returning the authorization protocol's type, returning the ID, etc, but the most important functionality there is the user enrollment. Regardless of the authorization protocol, each Data Source must be able to register users. In both

cases, the enrollment mainly consist of two steps. Receiving the registration call and redirecting the user to the vendors website and handling the callback made my the vendor. The following two Sequence Diagrams attempt to illustrate the OAuth1 flow of registration. These diagrams are made with CACHET's implementation, however, despite the *WebserverVerticle* object, it is all core concern. Figure 4.10 displays the first phase, which is redirecting the user to the vendor's website.

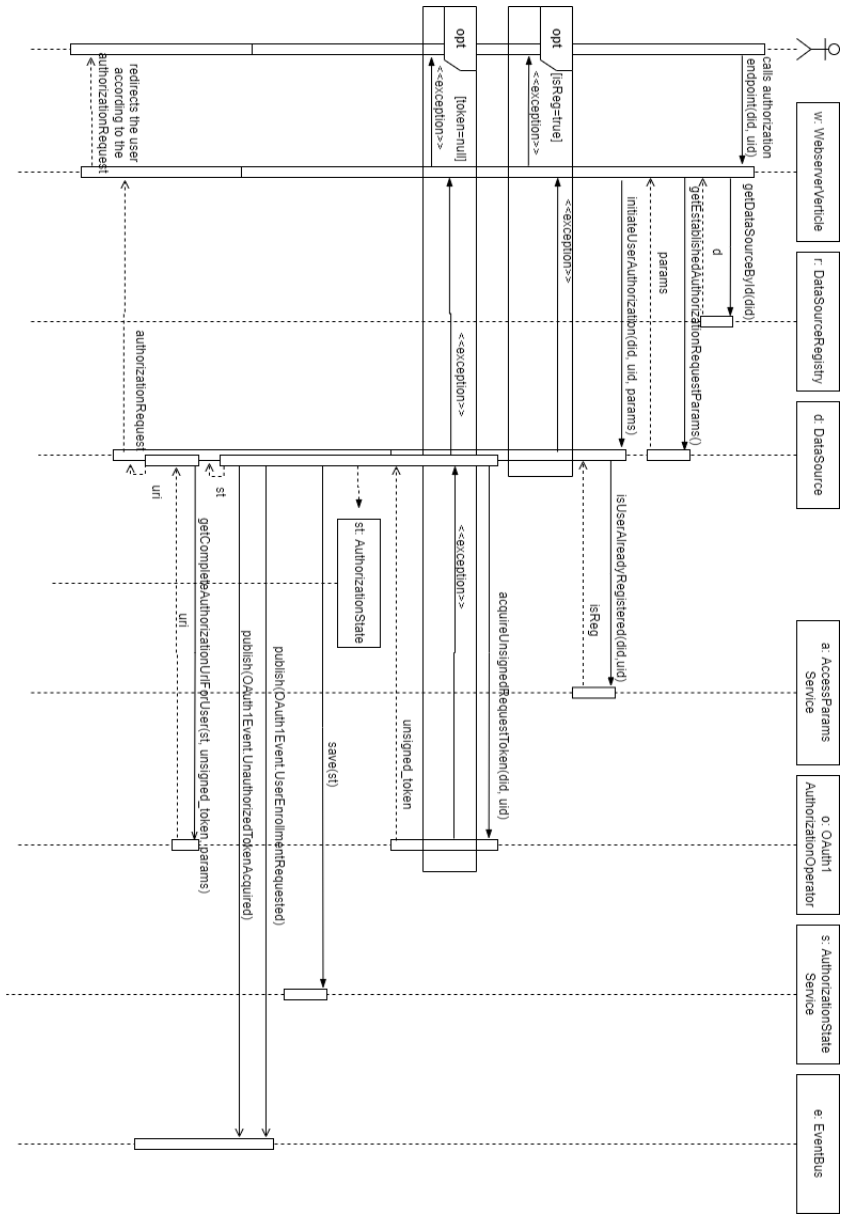

Figure 4.10: Sequence diagram of the OAuth1 User Enrollment process.

The flow starts from the point where the user calls the authorization endpoint. At that point the user's name (indicated by the *uid* variable) and the Data Source's ID (indicated by the *did* variable) the user wants to authorize to is known (the end-

point contains these parameters as path variables or query parameters). Once the required DataSource is retrieved from the registry and the required *AuthorizationRequestParams*, the *initiateUserAuthorization* call starts the flow. By the end of it, there is a new **AuthorizationState** object created and saved (*OAuth1AuthorizationCase* in this scenario) for the user and Data Source and the complete authorization URI (noted by *uri*) is established for the user, parametrized with the new state's ID. This is the URI where the user should be redirected to. In the end, the *initiateUserAuthorization* calls returns with a **AuthorizationRequest** object, which contains the mentioned URI and the state object. The implementation's responsibility is to answer the user's call with the proper redirection. Phase two of the enrollment process begins once the user authorized the application on the vendor's official website the third-party server called the registered callback. This is shown in Figure 4.11.

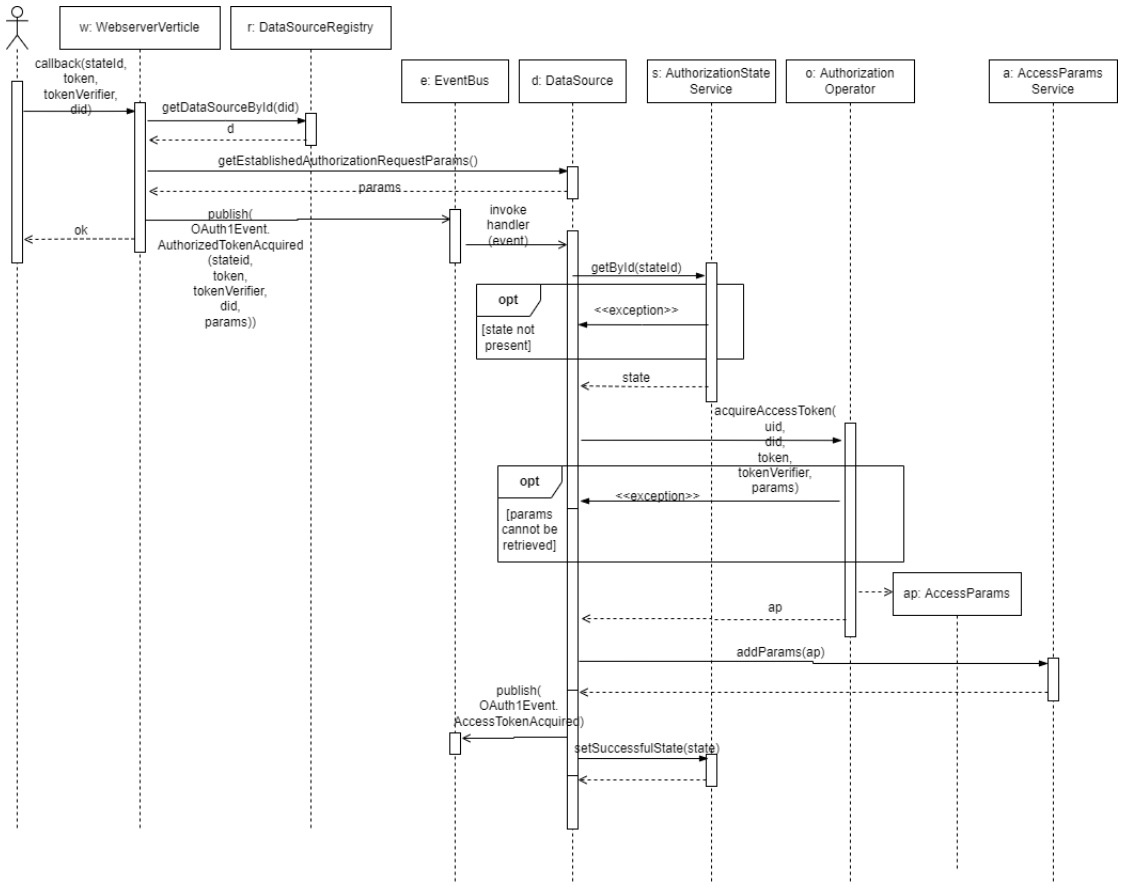

Figure 4.11: Sequence diagram of the OAuth1 Authorization callback process.

This diagrams shows the case when the user authorization was a success and the user gave consent to the application to collect data. The next step is to process the response and exchange the received information to authentication tokens. An important point to highlight here is the use of the *EventBus*. From the perspective of the framework's user, their only responsibility is to publish a new authorization protocol specific event with the required parameters. The inner workings of the framework does not have to be known, since the EventBus acts as a facade here. Once the event is published, the targeted DataSource, after it reads up the state information, tries to exchange the received information to the final authentication parameters. After a successful retrieval, the new tokens are saved and the authorization flow is concluded. The OAuth2 flow of user enrollment is basically the same, but simpler, since it does not require three network calls.

Continuing the the description of the DataSource Class diagram, after the *IDataSource* interface is introduced, the **DataSource** abstract class is next on the line and it provides implementation for some of the interface's functionalities, introduces new ones and declares the three common dependencies for both OAuth1 and OAuth2 Data Sources, the *IAuthorizationStateService*, *IEventBus* and the *IAccessParamsService*. The DataSource class is abstract, because it does not provide every implementation required by the interface. This is the one common super-class before the whole hierarchy splits into the OAuth1 and OAuth2 branches, so this is mainly used to declare template methods[25] to orchestrate function calls. These template methods aims to encapsulate protocol independent logic. For instance, as mentioned before in the enrollment process and illustrated on Figure 4.10, the *initiateUserAuthorization* call creates a new *AuthorizationState* object in the OAuth1 as well as in the OAuth2 flow and then calls other operations before it returns. Nevertheless, moving down one layer on the diagram to its descendants, they are the **OAuth1DataSource** and **OAuth2DataSource** abstract classes. These are completely independent of each other, as their respective protocols are. The main business logic of handling the authorization flow steps are implemented here and it provides implementation for every function declared by their super-classes/interfaces that do not require any device specific knowledge. Even though they are different, their dependency declarations are very similar. They both require their client setting classes, **OAuth1ClientSettings** and **OAuth2ClientSettings**. These encapsulate device specific variables, such as client_id and client_secret in case of OAuth2 and consumerkey and consumer_secret in case of OAuth1. The operator classes are the ones performing the network calls. The event subscription is also initialized here for handling the callbacks from the third-party Web API as explained earlier with the two sequence diagrams. The most-bottom classes on the diagram are completely protocol specific. The **OAuth1RequestToken** encapsulates the OAuth1 Request Token and Token Secret pair. The **OAuth2TokenRefreshParams** provides a way, just like the *AuthorizationRequestParams*, to customize the OAuth2 token refreshment calls with additional fields. There are also two other fields in both cases, the **DataCollectionExecutionEvent** classes and the **DataCollectionPreparationEvent** declared at the very top of the diagram. These are used for initiating the data collection flow. This aspect of the structure is not detailed here, because it is explained in detail in section 4.4 and illustrated on sequence diagrams starting from Figure 4.17.

This structure at first sight might seem a bit overcomplicated, but its merit shines when a concrete wearable device has to be integrated into the framework. The following two example showcases the structure of the existing Fitbit and Garmin implementation. Dexcom and Withings are not detailed here, because they are exactly the same as Fitbit's integration. Firstly, Figure 4.12 displays the class diagram of the Fitbit integration. For Fitbit utilises OAuth2 as authorization protocol, it has to inherit the *OAuth2DataSource*. The **Scopes** enum class is there to model the available scopes from Fitbit. Moreover, the **FitbitDataCollectionType** details the

supported data types. There are only four methods in the class that required to be implemented. The functions returning the unique identifier of the class, the *constructScopeStringsForAuthorizationUrl* method, which returns the requested scopes in the format Fitbit requires (separated by white spaces), the *assembleDataCollectionUri*, which returns the URI where the requested data type can be queried and lastly the *getDataCollectionPreparationEventFromPing*, which extracts from a *DataCollectionPreparationEvent* object from the ping notification received from the third-party Web API. There is one more dependency highlighted on the diagram, which is the **ConfiguredObjectMapper**. This class is the chosen the singleton class of the chosen serialization library, which helps with JSON related operations, such as parsing the notification ping. It's implementation will be detailed in the Implementation section (5).

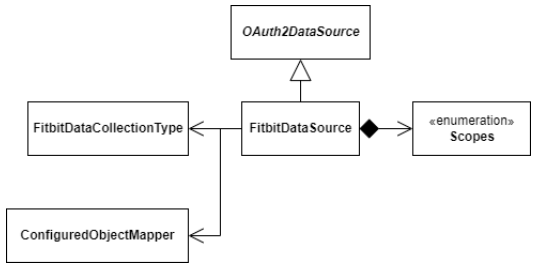

Figure 4.12: Class diagram of the FitbitDataSource.

Secondly, Figure 4.13 showcases the implementation of Garmin. It is exatcly the same as Fitbit's, however, it inherits from the *OAuth1DataSource* since Garmin utilises OAuth1 and there is no *Scopes* class, since OAuth1 does not have the concept of scopes, so this implementation is even more simpler than the previous one. It overwrites the same methods as Fitbit excepts the *constructScopeStringsForAuthorizationUrl* operation.

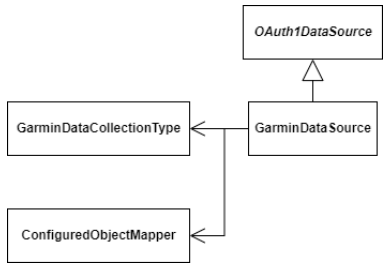

Figure 4.13: Class diagram of the GarminDataSource.

The most significant aspect of these two implementations is the fact that they only required the implementation the bare minimum of operations, which are completely vendor dependent and cannot be handled generically. By implementing the correct super-class for an authorization protocol, the bulk of the logic comes out of the box, the developer do not have to worry about the details, they just has to focus on implementing the vendor specific functions. This highly eases the way of integration more wearable devices on the way, which was one of the goals of the design of the framework.

This wraps up the design of the Authorization module. In conclusion, this design provides a way to handle the requirements of enrolling users and managing the tokens. It also simplifies the integration of new Data Sources as much as possible for future expansions.

4.4 Data collection module

Once the users are enrolled and the application possesses authentication information, the requirements are met to start querying user data from the vendor's Web API. Every API offers different kinds of data types with different structures and different ways to query them, however, there is always a unique identifier and endpoint that is associated with every data type. These are the information the previously introduced *DataCollectionType* interface aims to encapsulate.

The unique identifier of the Data Types is an important piece of information. Vast majority of the modern APIs are equipped with a subscription based service. Whenever the end-user uploads some data to the vendor's web server, a notification is sent to the subscribed application with usually the content of the user's id and the identifier of the updated Data Type. With this information the application can effectively query the Web API to retrieve the updated data in a near real time fashion. However, there are also APIs (e.g. Dexcom), which does not provide a service like this. There is no way to get the data whenever it is updated, the collection has to be queried from time to time to get the updates.

Furthermore, even though the main part of the module is to retrieve the data, there should also be a way to transform the data into a specific format chosen by the user of the framework. As mentioned, every Data Type is represented in an entirely different way. One Heart Rate Data Type of a Web API may completely differ from one other's Type. The framework must offer a possibility to register custom data transformation logic to every registered Data Type in order to get the data into a desired format. In this section the design of the Data Collection Module will be outlined and the decisions taken to meet these requirements.

First of all, Figure 4.14 displays the Component Diagram of the Collection module. On the left side, the **DataCollectionService** is the main component that will

handle the retrieval and transformation of data. The right side is occupied by the **DataTypeTransformerRegistry** module, which will hold the registered *Transformers* for each DataSource. These will be explained later in the section.

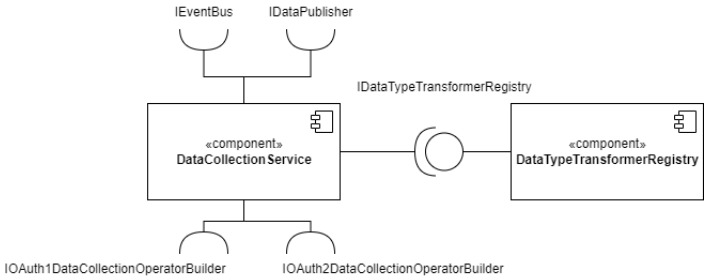

Figure 4.14: Component diagram of the Collection module.

There are five interfaces in total required by the *DataCollectionService* and there is just one that is provided by the module. Starting from the top, the **IEventBus**, just like in case of the Authorization module, is there for messaging. the **IDataPublisher** is the one that publishes the data retrieved from the third-party Web Server in a transformed format. This is the developer's responsibility to implement it to meet their requirements. The Operator Builders at the bottom are builder classes that return configured Data Collection Operators. These Operators are used to execute the actual HTTP requests towards the Web Servers to retrieve the data. Figure 4.15 shows the Class Diagram of the *DataCollectionService* component.

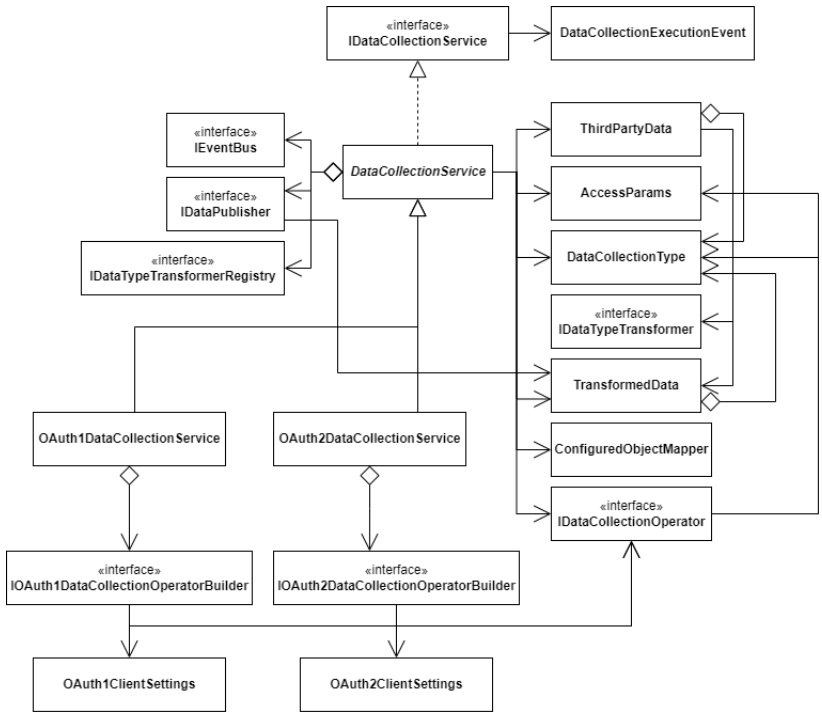

Figure 4.15: Class diagram of the DataCollectionService component.

At the very top, there is the **IDataCollectionService** interface which only function is to handle **DataCollectionExecution** events. The whole data collection flow will be illustrated and explained using Sequence Diagrams after the Class Diagram is discussed. One layer below the **DataCollectionService** abstract class handles the collection and transformation logic. It is abstract, because it has a factory method which returns a configured *DataCollectionOperator* according to the correct authorization type used by the Data Source the data should be collected from. Essentially, the concrete children of the *DataCollectionService*, namely **OAuth1DataCollection Service** and **OAuth2DataCollectionService** just implement that factory method and subscribe to their respective *DataCollectioExecutionEvent* (It also has OAuth1/2 specific version). With the help of the Operators, the *DataCollectionService* is able to retrieve the data from the Web servers. A collected data is represented in the system as a **ThirdPartyData**. It contains the raw response from the API and some metadata to identify it, such as *DataCollectionType* and IDs of the user and Data Source. Once the data is collected, it has to be transformed. The **IDataTypeTransformerRegistry** and **IDateTypeTransformer** interfaces serve this purpose. The hierarchy of these interfaces is showcased on Figure 4.16.

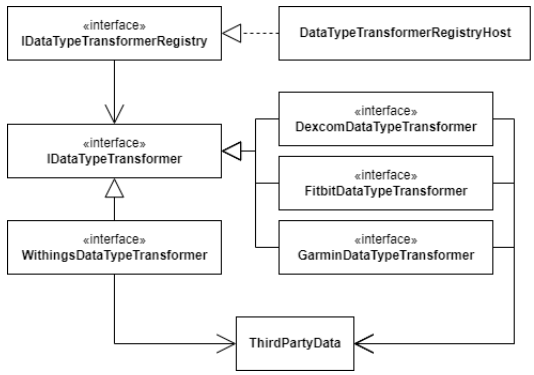

Figure 4.16: Class diagram of the DataTypeTransformerRegistry component.

The *IDataTypeTransformer* represents a class that is used for transformation. Every Data Source is supposed to have it's own transformer interface, the transformers, shown as the descendants on the Diagram. These interfaces lists one method for each supported Data Type listed in the corresponding *DataCollectionType*. They take a *ThirdPartyData* as an argument and transform it according to the logic implemented by the user of the framework. It is the responsibility of the developer to provide the implementation for these classes that fits their needs. The *IDataTypeTransformerRegistry* serves as a collection of these transformers. It allows registration of one transformer per Data Source and these can be retrieved later on with the Data Source's identifier. Once the data is correctly transformed, it becomes a **TransformedData** and that can be published using the **IDataPublisher** interface.

To explain this collection flow in more detail, the following Sequence Diagrams will demonstrate how the collections works in case of an OAuth2 Data Source. As with the earlier sequence diagrams, these are also done from CACHET's implementation perspective. It consists of three phases. The first phase is illustrated in Figure 4.17.

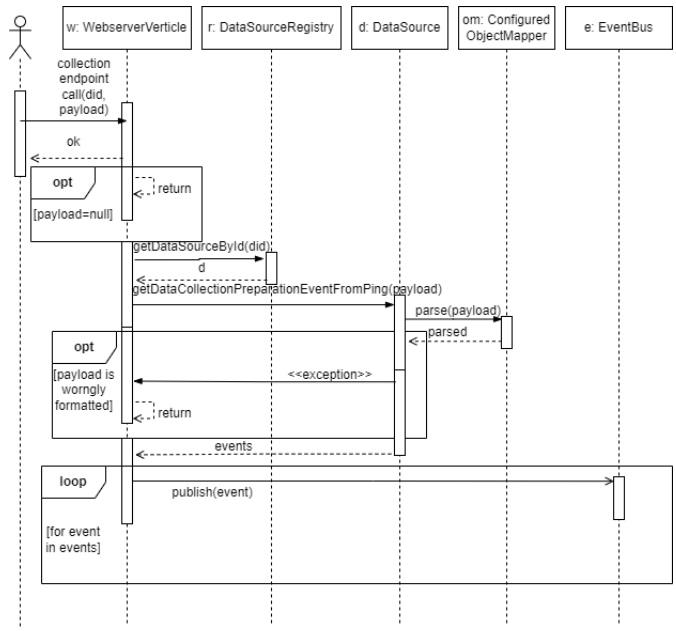

Figure 4.17: Sequence diagram of the OAuth2 data collection endpoint flow 1.

This phase's main focus is on the publishment of **DataCollectionPreparationEvent** objects, which contains the three significant information needed for data collection. What should be collected, (*DataCollectionType*), for whom (ID of the user) and from where (ID of the Data Source). The diagram showcases the scenario when the Web API notifies the application using the subscription service. Every Data Source will have a *getDataCollectionPreparationEventFromPing* helper method, which will establish event object from the raw, Data Source specific ping notification. Once they are created, they can be published using the *EventBus*. However, ping notifications are not always the case, as mentioned earlier. If one Data Source does not possesses a subscription system and it has to be queried periodically. This design is also capable of accomplishing that. These events can be constructed by the user, since they do not require anything specials besides the three properties (what, from where, for whom) and can be published. A cron job or any polling implementation can publish these events and the rest of the flow behaves the same way. After the preparation events are published, phase 2 begins, which is illustrated on Figure 4.18.

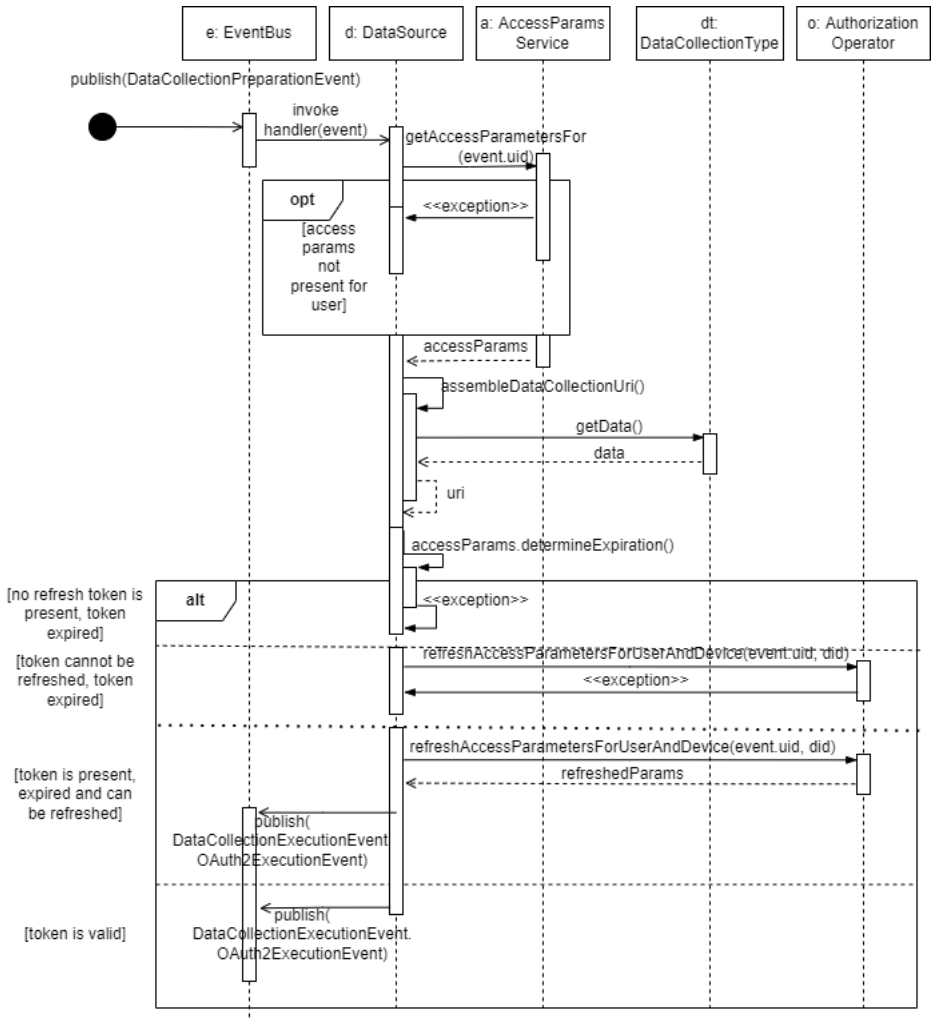

Figure 4.18: Sequence diagram of the OAuth2 data collection endpoint flow 2.

The second phase is fully the Authorization module's responsibility. When it received the event, it will gather the necessary information for data collection. It will query the access parameters for the user and Data Source, the client settings of the Data Source, a *DataCollectionType* and a completely assembled URI where the Data Type can be collected using the access parameters. Once the gathering is completed, the expiration of the *AccessParams* are checked. This step is missing in OAuth1, since its tokens are long-lived and there is no notion of refreshing the token without the need of re-authorizing the user. Once the tokens are successfully refreshed if it was needed, the new **DataCollectionExecutionEvent** is constructed with the pa-

rameters mentioned and published to the *EventBus* and phase three starts off, shown on Figure 4.19.

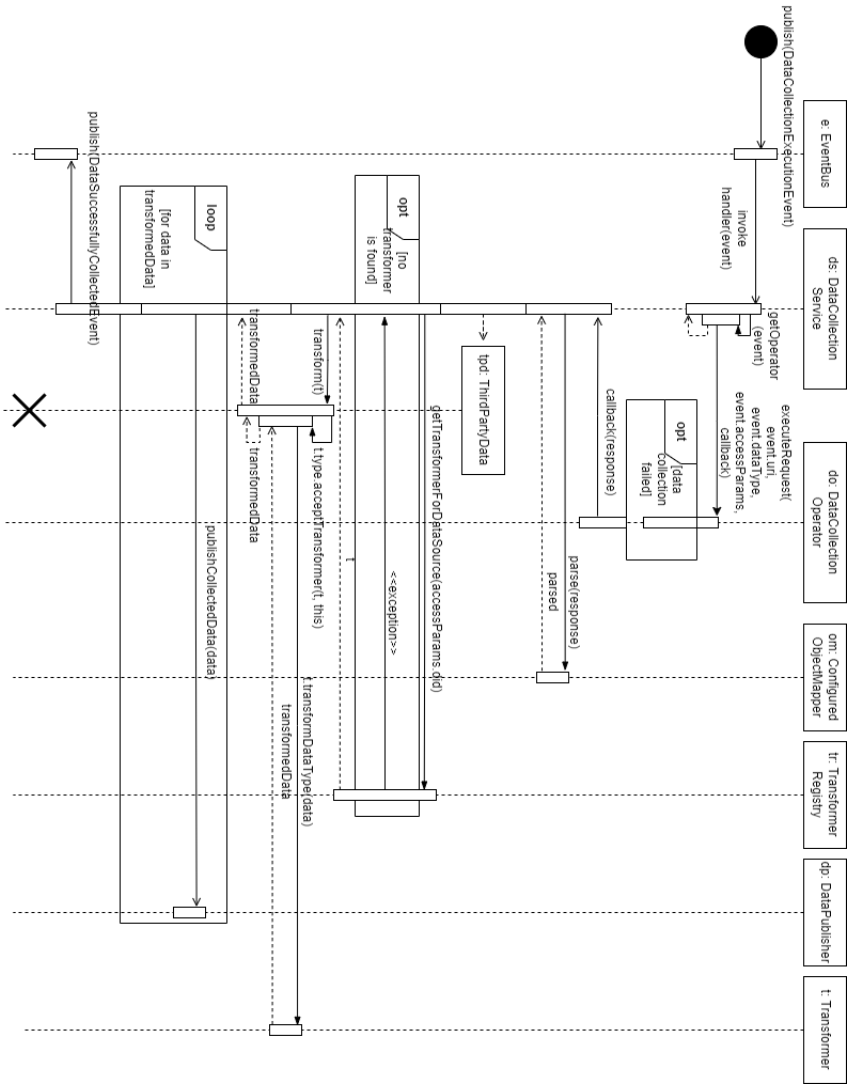

Figure 4.19: Sequence diagram of the OAuth2 data collection endpoint flow 3.

Phase three is the part where the collection module performs its purpose. Upon the retrieval of the execution event, the *DataCollectionService* calls the execution on its configured operator with the parameters found in the event and with one more

parameter, which is a Kotlin higher-order function as a callback function that is supposed to be used once the network communication successfully conducted. If so, the callback is called with the raw response from the third-party API and a new *ThirdPartyData* object is initialized. At that point, the transformation is next, after the proper *IDataTypeTransformer* is retrieved from the registry for the Data Source. The whole procedure utilises a type of the Visitor design pattern[25]. The *ThirdPartyData* knows what type of data it contains through its *DataCollectionType* field, so after it gets the transformer as a parameter, it can choose the appropriate transformation method using the Double Dispatch approach. As a result, new *TransformedData* objects are constructed that is ready to be published through the *IDataPublisher* interface.

In the end, this concludes the design of the collection module. The most significant part of this module is how the transformers are handled. With the use of the *IDataTypeTransformer* and the *IDataTypeTransformerRegistry* interfaces, the logic of the collection module will never have to be modified when extending the framework with new Data Sources or different transformers, honoring the Open/Closed principle. The registered transformers for Data Sources are even changeable at runtime, because the framework only utilises interfaces.

4.5 CANS Implementation architecture

The framework on itself is not a functional software. As it was seen on the Framework's Architecture Component Diagram on Figure 4.1, it has many unsatisfied dependencies that need to be provided. The CANS implementation aims provide these requirements according to their existing infrastructure.

CACHET's main system, CANS, as discussed in the Analysis Section 3.5, is running with Spring in a Docker container on a Linux based Web Server, using PostgreSQL as its main database and RabbitMQ for messaging. With the current design of the framework, the implementation possibilities are numerous. However, to not complicate the main system further it was decided to implement the framework in a different project and run it as a separate microservice. This way the only real limitation regarding the infrastructure for the implementation is the RabbitMQ, since that is being used by the main system, so the framework's implementation also has to connect to that in order to push the required data. Talking about the required data, CANS also requires a transformed data to be aligned with their "data points". The transformation logic using the *DataTypeTransformer* interfaces also have to be implemented.

The web framework and database can be freely chosen independently from the main system. Thus, the implementation will use Vert.x as an asynchronous web framework and MongoDB as the database. The following Component Diagram on

Figure 4.20 details how the implementation satisfies the Core's dependencies.

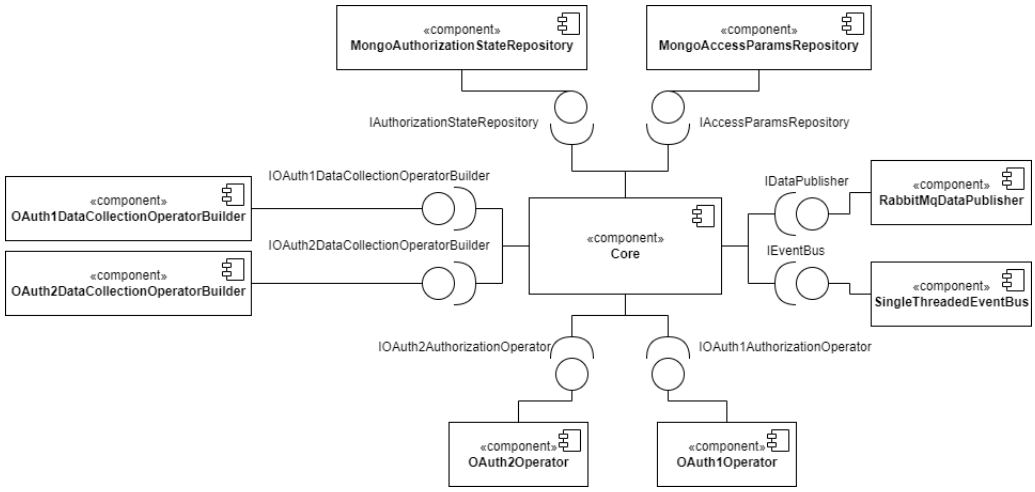

Figure 4.20: Component diagram of the CANS implementation.

Starting the explanation from the top, both the *IAuthorizationRepository* and *IAccessParamsRepository* interfaces will be implemented by classes that connect to a MongoDB instance. The *IDataPublisher* interface is realized by a class that connects to the main RabbitMQ instance on the server. The *IEventBus* is provided by the framework's **SingleThreadedEventBus** class, which simulates the messaging on the same thread the event was fired. This is class later on could be changed to a class using RabbitMQ instead if true parallelism needs to be supported. The Authorization and Data Collection operators are all implemented by the **OAuth1Operator** and **OAuth2Operator**. The builders return instances of these classes. These classes make use of the Scribe Java library[3], which is one of the most used OAuth utility libraries in the Java ecosystem. The *IDataTypeTransformer* implementation are not noted on the Diagram, however, there must be implementation classes for Garmin's, Fitbit's, Withings' and Dexcom's *IDataTypeTransformer* interfaces and these have to registered in the *IDataTypeTransformerRegistry*.

With the dependencies satisfied, the framework can correctly be instantiated and called. However, there also must be a Controller Layer where the HTTP requests can be mapped to the correct handlers and the framework service functions can be called. With the current design, three endpoints can handle everything.
The first endpoint must be the one where the user enrollment will be started. This

[3]https://github.com/scribejava/scribejava

is the endpoint that initiates the authorization flow illustrated on Sequence Diagram 4.10. The endpoint's URI will be the following: "*/wearables/api/authorize/: dataSourceId/:userId*". The ID of the DataSource and the ID of the user about to be rolled in can be extracted from path, thus every required parameter is met.
The second endpoint is the callback method that is called by the third-party Web Servers once the user gave consent. This flow is noted on Sequence Diagram 4.11. The path of the URI should be the following: "*/wearables/api/oauth/:dataSourceId /callback*". With this, the DataSource ID can be found in the path. However, most of the required params, including the "stateId" (which is the ID of the *AuthorizationState* object associated with the user's current authorization session) will be found in the query parameters as the protocols dictate. The rest can be extracted from there.
The last endpoint is the one that initiates the data collection. In this case, this is the endpoint that is called by the Web Servers using the subscription mechanism. The URI in this case is: "*/wearables/api/collection/:dataSourceId*". The ID of the DataSource, as in the previous cases, can be found in the path. The Ping notification itself will be present in the request body.

The last thing that needs to be addressed is the way this all fits in into CACHET's current projects. As a background, CANS is used to run *studies*. These studies are managed by researchers and their aim is to collect data according to a set of rules defined in a *protocol* that is contained by the study. The researchers are also capable of recruitment of participants to the study. Once they are enrolled, they can be *deployed*, which results in a creation of a new *deployment*. This deployment can be comprehended as a manifestation of the study. It belongs to a recruited user and they able to upload data to that specific deployment according to the protocol defined in the study, which also applies to the deployments. These protocols define a device which is used for data collection. These devices could be numerous things, but the notable part is that they could be, as of right now, a Garmin or a Fitbit device. This is where this application comes into the picture, because this allows CANS to authorize the wearable devices of the users and collect data from them, which prior this, was not possible. Once a deployment is created, the user receives an email. If the protocol definition contained a Fitbit or Garmin device as a device description, CANS will insert an authorization link into the email which can be clicked on by the user and that will redirect them to the first endpoint, which will start the authorization flow. By the end of the process, the application should be able to collect data from the participants, transform the data into the "data point" format and publish it to the RabbitMQ. On the other end, CANS will act as a consumer of these messages and save them.

4.6 Deployment process

As the previous section aimed at designing an implementation that fits into the existing CANS ecosystem, this section is dedicated to the design of how the implementa-

tion should be deployed into the mentioned infrastructure. Before the actual deployment configuration can be discussed, the profiling aspect of the application should be taken care of. There are many configuration constants in the framework/implementation that need to be externalized. Vert.x offers a way to place JSON files containing key-value pairs into the classpath, which can be easily read up at the application's start time. This configuration files will contain the profile specific connection strings to RabbitMQ and MongoDB as well as the client configuration settings for the different Data Sources. For this version of the project, there is two main profiles that can be used: *local* and *production*. Local is meant to be used for local development, so the connection string will mainly target localhost as destinations. The production is aimed at CACHET's server where CANS is deployed, so the connection strings will attempt to connect to the existing Docker Network to reach the existing RabbitMQ instance.

As outlined in Section 3.6, the deployment will use Docker as containerization technology. The implementation needs to have its own Dockerfile containing the image build logic. Once it can be run as a container, an orchestrator tool is needed to run multiple docker containers, because MongoDB and the RabbitMQ should also be started, so the application can establish connection to them. Docker-compose does just that, with two profile-specific Docker-compose configuration, the entire architecture can be set-up in a single command. The Deployment Diagram on Figure 4.21 showcases the current *production* profile architecture set-up.

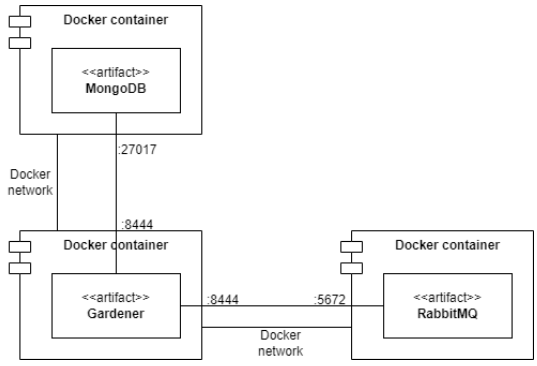

Figure 4.21: Deployment diagram of the Docker Production Configuration.

The application runs in its own docker container with the port *8444* open. That port is the application's server's port. The container is also attached to the same docker network as the MongoDB's container and the RabbitMQ's container. The docker network makes it possible to use container names in the connection strings instead of IP addresses, which is a huge benefit. On the noted ports the application can connect to the other two software. The next Deployment Diagram on Figure 4.22

illustrates how the deployment is going to look like on CACHET's Web Server.

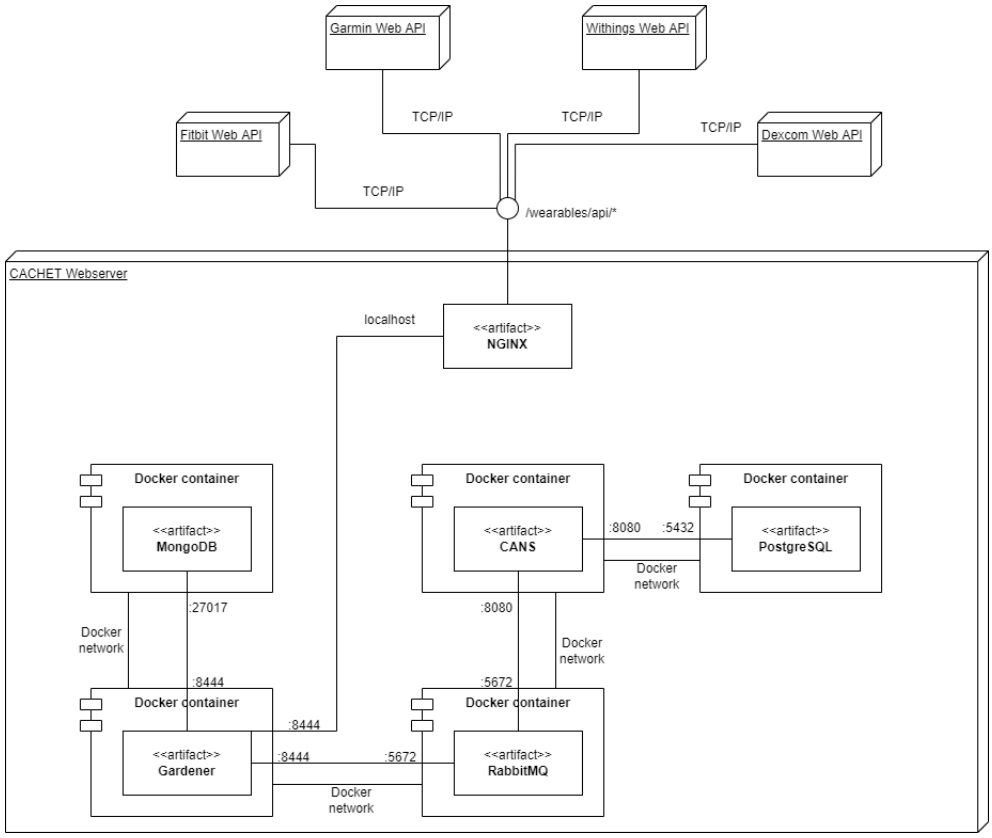

Figure 4.22: Deployment diagram of Gardener on the CANS server.

At the left corner of the figure, the mentioned architecture runs. CANS and its PostgreSQL database runs on different ports, but on the same docker network. RabbitMQ is the only connection between the components, which is beneficial, because the existing infrastructure on the server did not have to be modified in any way. The communication happens through the RabbitMQ message queues. One more important piece of the architecture is the NGINX reverse proxy sitting at the top of the Webserver module. It is not running in Docker, so it cannot reach the Docker network, thus it has to target the application on the *localhost*, which is completely valid, however, the docker container's port mapping also have to be configured to occupy the host's port *8444*. Whenever a request comes in from the outside targeting the "*/wearables/api/**" URI, the reverse proxy redirects the request to the

"*localhost:8444*" location, which is taken by the application, so it can be served. This deployment configuration is tested and the results will be discussed in chapter 6.2 and 6.3.

4.7 Design overview

To effectively summarize the design process and reason for some of the decisions taken, the goals of the project should be recapped, stated in section 1.3. The first objective was to achieve extensibility, the ability for developers to extend the framework with ease. The deep class hierarchy of *DataSource*, illustrated on Figure *data-source-class*, was designed to achieve this property. In the concrete *DataSource* classes only the wearable device specific functionally has to be implemented, every other aspect comes by inheriting from the super classes. In total with this design, the following parts have to be implemented, when integrating a new device: The concrete *DataSource* class, inheriting from one of the *OAuthDataSource* classes, a *DataCollectionType* class detailing the supported Data Types of that Data Source and lastly, a device specific *IDataTypeTransformer*, which will detail the transformation logic to each specific Data Type. None of these could have been part of the generic framework, because all of them are device specific information. Furthermore, the question of customisability comes down to a lot of factors in the system. Its most important characteristic was to not limiting the user to specific technologies. The design does not enforce any technology besides Kotlin and Jackson's serialization library. This freedom was achieved by the utilisation of interfaces and dependency injection techniques. The developers can freely implement their own versions of interfaces with the technologies they desire and those implementations can be injected into the core classes. Besides the technologies, data transformation logic was also a huge concern. The Collection Module current design allows the registration of any *IDataTypeTransformer* implementation for any *DataSource* and the existing code does not have to modified in any way for the new transformation to succeed. Furthermore, the use of the *EventBus* also offers high level of customisability. Developers can register their own handlers to each of the event types and those handlers will be executed. Lastly, CACHET's implementation, as discussed in section 4.6, fulfills their requirements and fits into the existing infrastructure. The next section will explain the implementation of the design and elaborate on these statements written here.

CHAPTER 5
Implementation

This chapter deals with the details of the implementation of the concept presented in the previous section. First, the structure of the project is explained in more detail. Secondly, the framework and its components are discussed. Next, it explains how the implementation uses the framework and meets the requirements set by CACHET, and finally it discusses the operational details. It is important to note at this point that it is highly advisable to look at the source code while reading this chapter to gain a better understanding of the topics discussed, as not only is full documentation provided using KotlinDoc, but also only the most important pieces of code are presented in this report.

5.1 Project structure

In order to distinguish the framework and CACHET's implementation, the project is implemented using Gradle's multi-project set-up. This allows separate dependency declarations, thus the framework remains free of any unnecessary dependency. In addition, the other merit of this detachment is that the core framework can stay fully general purpose. CACHET's requirements do not affect the framework's functionality, thus it can be reused by other users to implement their requirements. The name of the project containing the framework and the implementations as subproject is **gardener.parent**. In the root folder, the *settings.gradle.kts* file lists the other two projects, so Gradle is able to recognise and include them in the build task. The most of the remaining files here are mostly Gardle related configuration files, a *README.md*, used to give an introduction about the project, and a *.gitignore* file, used by Git[1] to remove files/folders from the version control. The subproject of the framework is in the **gardener.core** package and the implementation's is in the **gardener.carp-implementation** package. Both of the subprojects follow the same structure: a *build.gradle.kts* file describing the build tasks and dependencies and a *src* folder containing the source code. In addition, the carp-implementation also harbors a folder named *docker*. It contains the Docker specific files such as the main Dockerfile of the project and profile specific docker-compose files as well as configuration files and set-up scripts for the MongoDB and RabbitMQ instances. As a result of the multi-project configuration, both subprojects can be separately built, ran, and tested. Using the

[1] https://git-scm.com/

parent project, commands can be issued that will be applied for every subproject. In the following sections, the subprojects will be discussed in detail and in Section 5.4 the building and containerization process will be highlighted.

5.2 Framework

First and foremost, the implementation of the framework has to be detailed in order to establish the application core. The package of *gardener.core* contains the *src* package, which contains the source code. It has two subpackages, namely *main* and *test*. The *test* package contains unit and integration tests of the framework's functionality. This will be discussed in detail in Section 6.1. The *main* package is the one containing the actual source code that is the implementation of the Design Section. There are four main packages representing the modular design discussed in Section 4.1. The *authorization* package contains the Authorization Component exclusive classes. So does the *collection* package with the Collection Component specific classes. Moreover, the *common* package harbors the the classes and interfaces that are being used by both components, for instance the *EventBus*. Lastly, the *infrastructure* package contains implementations for the required interfaces, noted on Figure 4.1 mainly for testing purposes. The repositories are using an in-memory storage, so it is not advised to use them in a real environment.

5.2.1 Authorization module

This section is dedicated to the Authorization Component, illustrated on Component Diagram of Figure 4.3. Before anything else, the package structure should be discussed. There are five main packages and each of them correlates to one of the components showcased on the Diagram. However, on the Diagram two more packages can be observed, namely the *AccessParams* and *DataCollectionType*. These are located in the *common* package, since they are being used by the Collection Module as well.

5.2.1.1 AuthorizationRequestParams

Starting off with the *authorizationrequest* package, it contains the **AuthorizationRequestParams** classes, represented on Figure 4.6. Since the frameworks aims to be general and to be implemented in many different ways, it has to offer customisability. The point of these classes is to offer an opportunity to customize the network calls made during the authorization process. The parameters appended to these collections are excepted to be attached to the requests. Every *DataSource* will define its own object of this, because some of them might require specific parameters to be present. For instance, in case of Withings, a header with a constant value needs to be present during the authorization. The *WithingsDataSource* will attach this to its object, shown on Listing 5.1. In the code snippet, there is one more noteworthy part.

The parameter passed to the constructor of the return object is a **RestrictedMap** object. This is a wrapper class created for cases like this one. This wrapper holds a mutable map object inside, but restricts the access to it by only allowing to append key-value pairs when the key is not already present in the collection. If so, the append will not succeed. This precaution was taken to avoid users accidentally modifying parameters that was put there by the framework, just like in Withings' case the "*action*" key. On the other hand, an immutable version of the wrapped collection can be requested by the class the values inside can be freely observed.

```
override fun getEstablishedAuthorizationRequestParams():
    AuthorizationRequestParams {
    val additionalParams: MutableMap<String, String> = mutableMapOf("action"
         to "requesttoken")
    return OAuth2AuthorizationRequestParams(additionalParamsForTokens =
        RestrictedMap(additionalParams))
}
```

Listing 5.1: AuthorizationRequestParams for Withings.

The abstract *AuthorizationRequestParams* class is also annotated with *Jackson* specific annotations to configure the polymorphic de/serializtion. It defines that the serialized form of this class will have an additional field called "*type*" and the OAuth1 child of the class will contain this field with the value of "*oauth1*" and the OAuth2 version is with "*oauth2*". If this were not there, Jackson would not know how to handle the abstract *AuthorizationRequestParams* classes when it comes to de/serializtion.

Additionally, the last field in the class is the *applicationData* field. This is a nullable String field, which is there to contain implementation specific data. Developers implementing the framework can choose what they want to be saved there and the value of this field will be saved with the *AccessParams* as well. This was included to ensure an even higher level of customisability and the benefits will be presented when CACHET's requirements will be discussed in Section 5.3.

5.2.1.2 AuthorizationState

Moving on on the list of components, this sections details the *AuthorizationState* component. Its class diagram is displayed on Figure 4.5. This class' objective is to save information during the authorization process. During OAuth1 and 2, there is a step when the user is redirected to the vendor's website to authorize the application. After the redirection, from the application's perspective, the HTTP connection is closed and the task is completed. However, there is data that should be persisted in every authorization session per user. Information, such the *Request Token* and *Token Secret* when it comes to OAuth1 and the *applicationData* field, discussed in the previous section. This data cannot be sent over the wire, because some of these should be kept secret. Fortunately, both of the protocols offer a way to send arbitrary data with the authorization requests and the vendors implementing their version of the protocol are obligated to send those back with the callback as query parameters. The

AuthorizationState object's id field is the perfect for this case. The id is the only thing that is required to be sent out from the application and during the callback this id can be used to query the entire State object and use the data. Therefore, the *AuthorizationState* abstract class contains field for user and data source identifiers, which can be used to correctly query the state for a given user and for a given data source. This class is also configured to be serializable in a polymorphic way, as discussed in the previous section. It has two descendants, the **OAuth1AuthorizationState** and **OAuth2AuthorizationState** classes. These objects can be persisted using the **IAuthorizationStateRepository**. It defines the needed persistence functionalities which to be implemented by the users of the framework. The **IAuthorizationStateService** and **AuthorizationStateServiceHost** which declare and realise the application services. The Service Host uses the repository to handle the persistence of the *AuthorizationState* objects and provides functionalities to manipulate them. This component's implementation is pretty straight forward. The significant part is their place in the whole picture.

5.2.1.3 AccessParams

Even though this is a common concern, it needs to be discussed here, because the next section will build upon it. The *AccessParams* represents the authentication tokens received after a successful authorization flow in case of OAuth1 and 2. Since the protocols utilise totally different approaches, it also has two children, the concrete classes, namely *OAuth1AccessParams* and *OAuth2AccessParams*. The Class Diagram of this component is illustrate on Figure 4.4. The connection between the repository and service classes and their responsibilities are completely the same, thus this section will rather focus on the implementation of the domain classes. First of all, the nature of the access parameters should be discussed in case of different devices. The protocols only dictate the presence of some fields, they do not restrict the expansion, which leads to the reality that the access parameters of different devices can vastly differ. Some may include some fields that need to be saved the retrieved when accessing data from the API. This was the main reason why the *AccessParams* abstract class stores the entire response from the third-party API as a Jackson *JsonNode* object, which represents JSON file which can be effectively queried and traversed. With this approach, if any implementation needs something special, it can get it by querying this object, because it will contain the raw response, ensuring that every device specific parameter is correctly saved there. Apart from this very significant design decision, it also contains additional fields, such as *internalUserId*, which is the ID of the user they were enrolled during the authorization process into the framework. The *externalUserId*, which is the ID of the user in the third-party system. The *dataSourceId*, which is the ID of the *DataSource*. The *applicationData* field, which contains the application specific data, set by the developer using the framework. The *AccessParams* class is also annotated with the usual polymorphic serialization settings, just as detailed in Section 5.2.1.1. In both OAuth1 and OAuth2 there is one token that can be used to access the protected resources. The *AccessParams* class declares an abstract method

to extract this token from the response. This is the *extractAccessToken* function, which has to be overwritten by the subclasses. Apart from that, there is one more function declared and implemented, which is the *getParamValueFor*, which purpose is to get the value for the key passed as a parameter from the *params* object, which contains the raw authentication information. Moving on to the first concrete class, the **OAuth1AccessParams** declares OAuth1 specific methods to extract protocol specific information. The extraction is represented on the following Listing 5.2.

```
companion object {
    const val OAUTH_TOKEN_KEY = "oauth_token"
    const val OAUTH_TOKEN_SECRET_KEY = "oauth_token_secret"
}
override fun extractAccessToken(): String {
    return params.get(OAUTH_TOKEN_KEY).textValue()
}
```

Listing 5.2: OAuth1AccessParams parameter extraction.

It declares the OAuth1 protocol specific keys and tries to extract them from the raw response. In this case, this is a mandatory field to be present in the authentication response. However, this might not be the same in other cases, especially in OAuth2. Its declarations are the following, on Listing 5.3

```
companion object {
    const val ACCESS_TOKEN_KEY = "access_token"
    const val REFRESH_TOKEN_KEY = "refresh_token"
    const val TOKEN_TYPE_KEY = "token_type"
    const val EXPIRES_IN_KEY = "expires_in"
    const val SCOPES_KEY = "scope"
}
override fun extractAccessToken(): String {
    return params.get(ACCESS_TOKEN_KEY).textValue()
}
fun extractExpiresIn(): Long? {
    return params.get(EXPIRES_IN_KEY)?.asLong()
}
fun determineExpiration(): Boolean {
    val expiresIn = extractExpiresIn() ?: return true
    return updatedAt.plusSeconds((expiresIn - 180)) < Instant.now()
}
```

Listing 5.3: OAuth2AccessParams parameter extraction.

The OAuth2 protocol marks the *expires_in* field as optional, so it might not be present in an implementation of the protocol. One more important function can be noted at the bottom of the snippet, called *determineExpiration*. This function is used, as the name suggests, determine the expiration. The *expires_in* field, according to the specification, contains the seconds the token is valid for counting from the moment of token creation. If the field is not present, the token is instantly marked as an expired token. If it is present, the comparison next to the return keyword will be used. The *updatedAt* field is used, because it contains the last time the *params* field was updated

(which is where the *expires_in* field is extracted from). The deduction of 180 seconds is used to artificially simulate a long running HTTP response. The application might have gotten the authentication information with a delay if the network struggled for some reason and the *expires_in* field is valid from the moment the token was issued on the third-part side, therefore without the artificial delay it might determine a token valid even if it is not.

5.2.1.4 DataSource

Next on the line of discussion is the *DataSource* package. Class diagram is displayed on Figure 4.9. This is the main component of the Authorization module. The discussion will be started from the top of the diagram where the most generalized from the *DataSources* are shown and on the way down the specialization will be detailed.

Said that, the **IDataSource** interface declares functionalities that should be used by the framework's user. The most important ones are the *initiateUserAuthorization*, *getDataCollectionPreparationEventFromPing*, and *getEstablishedAuthorizationRequestParams* functions. The *initiateUserAuthorization* is the one that should be called when a new uses needs to be registered and redirected to the third-party authorization page. This sequence is explained in detail on Figure 4.10. The *getDataCollectionPreparationEventFromPing* extracts multiple *DataCollectionPreparationEvent* object from the raw ping sent by the third-party Web Server. A list is returned, because some of the ping notifications contain more than one notification object, thus to extract all of them a list collection is needed. This function is one of the functions that needs to be implemented in every concrete *DataSource* just for the reason mentioned, every ping notification is different. Furthermore, the *getEstablishedAuthorizationRequestParams* function returns a protocol specific *AuthorizationRequestParams* object containing device specific constants if there is any. This function has a default implementation in the **OAuth1DataSource** and **OAuth2DataSource**, but it can be overwritten, just like how in Withing's case, explained in the previous Section.

Moving one layer down on the Diagram, the **DataSource** abstract class takes place. As already explained in the Design Section 4.3 of the Authorization module, this is the last level before the structure branches out into protocol specific implementations. This is also the place where the protocol independent dependencies are injected, such as *IEventBus*, *IAuthorizationStateService* and *IAccessParamsService*. Firstly, it handles the subscription for the **DataCollectionPreparationEvent**. Both OAuth1 and OAuth2 *DataSources* have to be able to gather the information necessary for their data collection requirements. Secondly, it implements some of the methods declared by the *IDataSource* interface. To examplify, the implementation of the *initiateUserAuthorization* function is displayed in the following code snippet on Listing 5.4.

```
1 override fun initiateUserAuthorization(
2         userId: String,
```

```
        dataSourceId: String,
        params: AuthorizationRequestParams
): AuthorizationRequest {
    if (accessParamsService.isUserAlreadyRegistered(userId, dataSourceId)) {
        throw IllegalArgumentException("User $dataSourceId/$userId is
            already authorized. Request aborted.")
    }
    val state = registerAuthorizationRequestForUser(userId, dataSourceId,
        params.applicationData)
    val uri = getAuthorizationRedirectUri(state, params)
    return AuthorizationRequest(uri, state)
}
```

Listing 5.4: Implementation of the *initiateUserAuthorization* method.

This method's main responsibility is to only allow new users to register, create a new *AuthorizationState* object for the user and construct the full redirection URI with the newly created state's ID. Creating the state object (*OAuth1AuthorizationState* or *OAuth2AuthorizationState*) and constructing the URI (calling the protocol specific operator) is protocol dependent, thus, it has to be implemented in either the OAuth1 or OAuth2 specific *DataSource*. Inspired by the Template Method Design Pattern, the class declares two new abstract functions *registerAuthorizationRequestForUser* and *getAuthorizationRedirectUri*. Moreover, despite other abstract functions, it also declares one more template method, named *prepareDataCollection*. This is the callback function of the *DataCollectionPreparationEvent*. It consists of two steps. Firstly, collecting the access parameters for the given user and Data Source using the *getAccessParametersFor* function, which is fully implemented here, because it is the same behaviour regardless of the protocol. Secondly, it calls the *assembleDataCollectionUri* operation to construct an **Uri** instance, which represents a web endpoint that should be called to collect a Data Type. This operation the second one that must be implemented by the concrete *DataSource* instances, because it is also fully device specific. This concludes the *DataSource* abstract class' part in the hierarchy. From now on, every operation will be protocol specific. With that said, the *OAuth1DataSource* and the *OAuth2DataSource* need to be detailed.

Starting with *OAuth1DataSource*, this class implements the OAuth1 protocol specific details. It declares two more protocol dependent dependencies, such as **OAuth1Client Settings** and **IOAuth1AuthorizationOperator**. The *OAuth1 ClientSettings* class encapsulates constants that are required by the OAuth1 protocol and parameters that are device specific, such as which callback URI should be called by the third-part Web Server or the links where third-party API is located. The **IOAuth1 AuthorizationOperator**, as discussed in the Design, performs network calls that are needed for the OAuth1 authorization flow. Moreover, this class handles the subscription for the **OAuth1Event. AuthorizedTokenAcquired**. Once this event is emitted, the *acquireAccessToken* function is called as a handler, as this sequence was shown on Figure 4.11. The following functions are also implemented here: *registerAuthorizationRequestForUser*, illustrated on Figure 4.10, *getAuthoriza-*

tionRedirectUri as declared by its superclass, it has to call the *IOAuth1Authorization Operator* to construct the required redirection URI, the *getEstablishedAuthorization-RequestParams*, which is the default implementation of the function as discussed previously in Section 5.2.1.1, which returns an empty *OAuth1AuthorizationRequestParams* and lastly the *publishDataCollectionExecutionEvent*, which is called at the end of the before-mentioned *prepareDataCollection* function. In case of OAuth1, this method is just a normal event publication of the *DataCollectionExecutionEvent.OAuth1Execution Event*.

Moving to its counterpart, the **OAuth2DataSource** has to implement the same functionalities according to the OAuth2 requirements. This *DataSource* is highly similar to the OAuth1 version. It also introduces its own version of client settings, which is **OAuth2ClientSettings**. Its purpose is the exact same as the *OAuth1ClientSettings*'s, but adjusted for the OAuth2 terms. Likewise, the second new dependency is the **IOAuth2AuthorizationOperator**. The same network call conductor as the OAuth1 version, but for OAuth2. Furthermore, this class subscribes for the **OAuth2Event. AuthorizationCodeAcquired** event and then it is handled with its own *acquireAccessToken* function. The logic of the function body is the same as in the other case. The rest of this paragraph will focus on the notable differences compared to OAuth1. Starting the list with the *getAuthorizationRedirectUri* call, which in this case receives *OAuth2AuthorizationRequestParams* as a parameter. This class contains a new field, called *scopes*. This field is a list of String containing the requested scopes from the third-party API during the authorization process. However, every API requires the scopes to be specified in a different format. For instance, Fitbit requires them to be separated by white spaces, while Withings accepts them separated by commas. Consequently, the *OAuth2DataSource* declares a new function called *constructScopeStringsForAuthorizationUrl* which has to be implemented by the concrete *OAuth2DataSource* for the reason mentioned above, it is completely device specific. Continuing the differences, the second notable one is the *publishDataCollectionExecutionEvent*. While in OAuth1 it was just a plain event publication, in this case it something distinct. The code fragment on Listing 5.5 showcases the function body.

```
override fun publishDataCollectionExecutionEvent(
    dataType: DataCollectionType,
    dataCollectionUri: Uri,
    accessParams: AccessParams
) {
    val tokensAreExpired = (accessParams as OAuth2AccessParams).
        determineExpiration()
    val refreshedParams = if (tokensAreExpired) {
      refreshAccessParametersForUserAndDevice(
        accessParams.internalUserId,
        accessParams.dataSourceId,
        getEstablishedTokenRefreshParams()
      )
    } else accessParams

```

```
15      eventBus.publish(this::class, DataCollectionExecutionEvent.
            OAuth2ExecutionEvent(
16        accessParams = refreshedParams,
17        clientSettings = clientSettings,
18        uri = dataCollectionUri,
19        dataIdentifier = dataType
20      ))
21  }
```

Listing 5.5: Implementation of the *publishDataCollectionExecutionEvent* method in *OAuth2DataSource*.

As it can be seen, it checks whether the *OAuth2AccessParams* are expired or not. The idea is that the *DataCollectionExecutionEvent* will only contain valid tokens, so the data collection will not fail due to expired credentials. The expiration check is detailed in Section 5.2.1.3. In case the tokens are expired, a refresh operation is attempted. This is done by calling the *refreshAccessParametersForUserAndDevice* function, which is also a new, exclusive function declared by the *OAuth2DataSource*. It calls the *IOAuth2AuthorizationOperator* to refresh the expired tokens and upon successful refreshment it saves the new tokens to the existing *OAuth2AccessParams* object and updates it in the database using the *IAccessParamsService*. After ensuring the validity of the tokens, it omits the *DataCollectionExecutionEvent.OAuth2ExecutionEvent*. The last difference is the newly declared *getEstablishedTokenRefreshParams*. This is also a customisation ensuring operation, returning an instance of **OAuth2TokenRefreshParams**. Some third-party APIs might require the presence of different information during the token refreshment. As an example, Withings also requires the same "*action*" header to be present for token refreshment, just as in case of requesting normal tokens. This function is also declared as open, so the concrete classes can overwrite if need be.

5.2.1.5 DataCollectionType

The second common concern that is utilised by the *DataSources* is the **DataCollectionType**. Its class diagram can be observed on Figure 4.7. This interface represents a third-party data type. Its functions are mostly aimed at enforcing the presence of different information pieces, such as *getIdentifier* method, which should return the ID of the Data Type from the third-party API's perspective, the *getNamespace* method, which is supposed to uniquely identify the data from the developer's perspective. To exemplify the difference, *getIdentifier* could be "*activities*" in Fitbit's case and *getNamespace* would be "*com.fitbit.activities.*". Activities is a pretty common data type among wearables, so the *getIdentifier* on itself could not be used as as ID in the framework. Furthermore, *getEndpoint* method returns the relative URI where the data can be collected on the vendor's website and *getCustomName*, which returns a humanly readable custom name for the data. Using the previous example, it would return "*Fitbit Activity Summary*". Additionally, there is one more method called *acceptTransformer*. This will be detailed later down in the section. Apart from the

interface itself, its descendants are mainly used in this module. As it can be seen on the Class Diagram, there is a concrete *DataCollectionType* for every concrete *DataSource*. Continuing using the Fitbit as an example, Listing 5.6 details a fragment of its implementation.

```
enum class FitbitDataCollectionType(
    private val id: String,
    private val ep: String,
    private val cn: String,
    private val ns: String,
    val version: String = "1",
) : DataCollectionType {
    ACTIVITIES(
        "activities",
        "activities",
        "Fitbit Activity Summary",
        "com.fitbit.activities"
    ) {
        override fun acceptTransformer(transformer: IDataTypeTransformer,
            data: ThirdPartyData): List<Any> {
            transformer as FitbitDataTypeTransformer
            return transformer.transformActivities(data)
        }
    }
    override fun getIdentifier() = id
    override fun getEndpoint() = ep
    override fun getCustomName() = cn
    override fun getNamespace(): String = ns
}
```

Listing 5.6: Implementation of the *DataCollectionType* interface for Fitbit.

FitbitDataCollectionType lists more data types, but only this one is taken as an example. It is an enumeration, because enums in Kotlin and many other languages can implement an interface and their nature perfectly fits the data type requirements. Every concrete *DataCollectionType* takes a minimum of four arguments in the constructor to satisfy the interface's definition and it it needs be, more arguments can be specified, just like in case of Fitbit, where another *version* argument is added, because some of their data endpoints have different versions than the others. Most of the interface method are realized in a global way applicable for every enum entry, except for one, which is the *acceptTransformer*. This is the method that calls the correct transformer on the Data Type for data transformation. This is presented on Figure 4.19. It takes a transformer and the data to be transformed as an argument, it casts the transformer to its expected type and calls the correct method. Essentially, this implements the Double Dispatching mechanism. The significance of this mechanism reveals itself when used in the Data Collection Module. It returns a list of unknown types, because the framework could not know the correct class of the transformed type, because it is implemented by the developer using the framework. In addition, the *ThirdPartyData* taken as an argument contains the whole raw data retrieved from the third-party API and it might contain multiple data en-

tries, depending of the data representation of the vendor. The structure of the other concrete *DataCollectionTypes*, such as *GarminDataCollectionType*, *DexcomDataCollectionType* and *WithingsDataCollectionType* are implemented in the exact same way, thus there is nothing important there to mention.

5.2.1.6 Devices

Having the *DataSources* and *DataCollectionTypes* explained, the focus should shift towards how they can be utilised. The concrete device implementations make use of these classes. The *devices* package contains the Dexcom, Fitbit, Garmin and Withigns implementation. All of them incorporates the same structure. An **AccessTokenResponseExtractor** utility class, which provides one static method that is supposed to transform the raw response from third-party Web Server into the protocol specific *AccessParams* class. This was mainly made to be used by the developers of the framework while implementing the *OAuthAuthorizationOperators*, because almost every vendor declares the token response in a different way. Furthermore, there is one device specific *DataTypeTransformer* interface, which lists the transformer functions for every data type declared in the device specific *DataCollectionType* classes. Last but not least, there is the main device specific *DataSource* class, which inherits from either the *OAuth1DataSource* or the *OAuth2DataSource*. To demonstrate the OAuth2 implementation, the *FitbitDataSource* will be taken as an example. Its Class Diagram can be seen on Figure 4.12. After inheriting from the *OAuth2DataSource*, four classes were left to be implemented. Starting off with the *constructScopeStringsForAuthorizationUrl* function, its implementation can be examined on Listing 5.7.

```
override fun constructScopeStringsForAuthorizationUrl(scopes: List<String>?)
    : String {
    if (scopes != null) {
      scopes.forEach {
        if (!Scopes.isValid(it)) {
          throw IllegalArgumentException("The requested Fitbit scope $it is
              not valid!")
        }
      }
      return scopes.joinToString(" ")
    }
    return Scopes.values().map { it.key }.toList().joinToString(" ")
}
```

Listing 5.7: Implementation of the *constructScopeStringsForAuthorizationUrl* function for Fitbit.

Fitbit requires the scopes to be separated by white-spaces. The function takes the scopes requested at user enrollment time as an argument, checks whether each of them are valid using the Scopes nested enumeration, which lists all the available scopes Fitbit has and and if so, it joins them together as it is required. If no scopes were passed to the function, it returns back every scope available, because, as the protocol dictates, scopes have to be present when requesting authorization. Proceeding to the

second necessary method implementation, it is the *assembleDataCollectionUri*. Its function body is shown on Listing 5.8.

```
override fun assembleDataCollectionUri(
    accessParams: AccessParams,
    dataType: DataCollectionType,
    rawPing: JsonNode
  ): Uri {
      dataType as FitbitDataCollectionType
      val uriString: String = when (dataType) {
        FitbitDataCollectionType.ACTIVITIES -> "${clientSettings.dataUrl}/${
            dataType.version}/user/-/${dataType.getEndpoint()}/date/${
            rawPing.get(AP_DATE_KEY).textValue()}.json"
        FitbitDataCollectionType.HEART_RATE -> "${clientSettings.dataUrl}/${
            dataType.version}/user/-/${dataType.getEndpoint()}/date/today/1d
            .json"
        FitbitDataCollectionType.BODY -> "${clientSettings.dataUrl}/${
            dataType.version}/user/-/${dataType.getEndpoint()}/date/${
            rawPing.get(AP_DATE_KEY).textValue()}.json"
        FitbitDataCollectionType.SLEEP -> "${clientSettings.dataUrl}/${
            dataType.version}/user/-/${dataType.getEndpoint()}/date/${
            rawPing.get(AP_DATE_KEY).textValue()}.json"
        FitbitDataCollectionType.FOOD -> "${clientSettings.dataUrl}/${
            dataType.version}/user/-/${dataType.getEndpoint()}/date/${
            rawPing.get(AP_DATE_KEY).textValue()}.json"
      }
      return Uri(HttpMethod.GET, uriString)
  }
```

Listing 5.8: Implementation of the *assembleDataCollectionUri* function for Fitbit.

As it can be seen, this is where the **FitbitDataCollectionType** is used, because, as discussed in Section 5.2.1.5, it has information about the actual data type. For instance, the endpoint where it is available. Using these, the full collection URIs can be constructed as the vendor requires them. In the argument list one more important parameter can be noted, the *rawPing*. This is the entire ping notification that the third-part Server sends when it notifies the application about the update of one data collection. It is practical, because some of the notification may contain some key information that might be needed for URI construction. In this specific case, it can be seen being used in the URIs: *rawPing.get(AP_DATE_KEY).textValue()*. *AP_DATE_KEY* is a constant in the class with the value of "*date*". This function call return the value for that field from the ping notification, because Fitbit pings contain the date that should be queried. Moreover, the third significant function is the *getDataCollectionPreparationEventFromPing*. Its implementation is demonstrated on Listing 5.9.

```
override fun getDataCollectionPreparationEventFromPing(notification: String)
    : List<DataCollectionPreparationEvent> {
    val eventList: MutableList<DataCollectionPreparationEvent> =
        mutableListOf()
    try {
```

```
4       val notificationNode = ConfiguredObjectMapper.instance.readTree(
            notification)
5       notificationNode.forEach { ping ->
6         val dataType = FitbitDataCollectionType.from(ping.get(
              AP_DATA_TYPE_KEY).textValue()) ?:
7           throw IllegalArgumentException("The requested Fitbit Data Type is
                not valid!")
8         eventList.add(
9           DataCollectionPreparationEvent(
10            dataSourceId = DATA_SOURCE_ID,
11            userId = ping.get(AP_USER_KEY).textValue(),
12            dataType = dataType,
13            rawPing =  ping
14          )
15        )
16      }
17    } catch (ex: Exception) {
18      throw IllegalArgumentException("Notification extraction failed for
            Fitbit: ${ex.message}. \nSent notification: $notification")
19    }
20    return eventList
21  }
```

Listing 5.9: Implementation of the *getDataCollectionPreparationEventFromPing* function for Fitbit.

Its main responsibility is to construct *DataCollectionPreparationEvent* instances from the notification pings. Object instances, because most of the vendors send multiple notifications in one ping. Therefore, using Jackson's configured *ObjectMapper*, the notification is transformed into a JSON structure and that structure is traversed, while in each iteration a new *DataCollectionPreparationEvent* is created according to the Fitbit notification. Lastly, the events are returned. In the end, the very last method that has to be implemented is the *getId* method. It just returns a String that uniquely identifies the actual *DataSource*. In this case, this is "*fitbit*". Furthermore, there is one more notable aspect, which comes out in case of Withings. As discussed before in Section 5.2.1.1, Withings requires some headers to be present at authorization and token refreshment. These constants are configured in the *getEstablishedAuthorizationRequestParams* and *getEstablishedTokenRefreshParams* methods. Fitbit was not required to implement these, because it does not require any constant to be present at those times and these functions have default implementation in the superclass. Moreover, this was the case of *OAuth2DataSource* classes, but Garmin utilises OAuth1, therefore it inherits from the *OAuth1DataSource*. That implementation is even simpler, because the scopes do not have to be specified there, thus only three methods had to be implemented: *getId*, *assembleDataCollectionUri* and *getDataCollectionPreparationEventFromPing*. The whole OAuth1 protocol logic came from the superclass.

5.2.1.7 DataSourceRegistry

The last component in the Authorization Module is the **DataSourceRegistry** component, contained by the package *datasourceregistry*. Its Class Diagram is shown in Figure 4.8. Its implementation is bare-bones compared to the previously introduced packages, however, it is still an important part that will be significant during the implementation. The **IDataSourceRegistry** interface declares two methods: *getDataSourceById* is for retrieving *DataSource* instances by their ID and *activateDataSource*, which saves the *DataSource* instances. The **DataSourceRegistryHost** realizes this interface and stores the *DataSources* in the Map collection, where the key is the *DataSource* ID and the value is the object itself.

5.2.2 Collection Module

The next module on the framework's Component Diagram (Figure 4.1) is the Collection Module. Its own component diagram is illustrated on Figure 4.14 and it is implemented in the *collection* package and has the following structure: package *data* declares classes for the data retrieved from the third-party APIs. Package *oauth1* harbors the OAuth1 specific services and so does package *oauth2* with the OAuth2 services. The package *publisher* contains the *IDataPublisher* interface. The rest of the classes are located in the root package.

5.2.2.1 DataCollectionSerivce

Starting off the explanation with the most important part of the module, the **DataCollectionSerivce**'s main responsibility is to collect data from the third-party services according to the received *DataCollectionExecutionEvent*. Its interface, **IDataCollectionService** declares one function which does just that, the *executeDataCollectionRequest*. The *DataCollectionService* is the main class that contains the logic for data retrieval and transformation. The whole procedure can be observed in the Sequence Diagrams describing the OAuth2 Data Collection flow, starting from Figure 4.17. Before the implementation of this class is explained, this time it makes more sense to start from the specialized classes. The **OAuth1DataCollectionService** and the **OAuth2DataCollectionService** are the specialization of the *DataCollectionService* and the only responsibilities they have are to subscribe to the protocol specific *DataCollectionExecutionEvent* and return the protocol specific **DataCollectionOperatorBuilder** using the *getOperator* class, which is the only function they have to implement. The correct **DataCollectionOperatorBuilder** is injected here in the constructor. The builder classes' responsibility is to construct a configured instance of a protocol specific **IDataCollectionOperator**, which declares a method to retrieve data from the third-party servers. Continuing with the *DataCollectionService*, once one type of (OAuth1 or OAuth2) *DataCollectionExecutionEvent* is received, the *executeDataCollectionRequest* function is called as a callback. The function body retrieves the correct operator using the *getOperator* function and calls the

executeRequest function on it with the specified arguments. One argument is a Kotlin higher-order function, which is supposed to act as a callback to a successful HTTP call. This was done to give the user the possibility to handle the HTTP calls asynchronously and in their callbacks only this higher-order function has to be invoked with the required parameters, which is just the raw response from the third-party API as a String. This higher-order function can be observed on Listing 5.10.

```
callback = { apiResponse: String -> processFetchedData(event.accessParams,
    event.dataIdentifier, apiResponse) }
```

Listing 5.10: Kotlin higher-order function in the *DataCollectionService*.

The main aspect to point out here is the actual function that should handle the response, the *processFetchedData*. A most significant fragment of its function body can be observed in Listing 5.11.

```
private fun processFetchedData(accessParams: AccessParams, dataType:
    DataCollectionType, result: String) {
    val resultNode = ConfiguredObjectMapper.instance.readTree(result)
    val fetchedData = ThirdPartyData(
        userId = accessParams.internalUserId,
        dataSourceId = accessParams.dataSourceId,
        dataIdentifier = dataType,
        rawResponse = resultNode,
        applicationData = accessParams.applicationData
    )
    val transformer = transformerRegistry.getTransformerForDataSource(
        fetchedData.dataSourceId)
    val transformedData = fetchedData.transform(transformer)
    transformedData.forEach { data -> publisher.publishCollectedData(data) }
```

Listing 5.11: Implementation of *processFetchedData* function.

The *result* parameter contains the entire raw response. Firstly, it is mapped into a JSON representation. A **ThirdPartyData** wrapper is instantiated to encapsulate the response and its meta-data. Then, the transformation begins. The *transformerRegistry* is an instance of the *IDataTypeTransformerRegistry* registry, which contains registered *IDataTypeTransformer* instances for each *DataSource*. Its inner workings will be detailed later in the section. The actual *transform* function is part of the *ThirdPartyData* class, which can be noted on the following Listing 5.12.

```
fun transform(transformer: IDataTypeTransformer): List<TransformedData> {
    val transformedDataTypes = this.dataIdentifier.acceptTransformer(
        transformer, this)
    return transformedDataTypes.map { transformedDataType ->
        TransformedData(
            dataSourceId = dataSourceId,
            userId = userId,
            dataType = dataIdentifier,
            collectedAt = collectedAt,
            value = transformedDataType
        )
```

```
    }
}
```

Listing 5.12: Implementation of *transform* function.

Since the *ThirdPartData* instance knows what type of data it holds through its *dataIdentifier* field, which is a *DataCollectionType*, it can the *acceptTransformer* method to initiate the Double Dispatching mechanism, detailed in Section 5.2.1.5. After the transformation is completed, it constructs a **TransformedData** instance, which is also a wrapper class for the transformed data and its meta-data. This concludes the transformation. The only part left is to publish the retrieved and transformed data using the *IDataPublisher* interface. This is supposed to be implemented by the developer using the framework.

5.2.2.2 DataTypeTransformerRegistry

The last component of the Diagram 4.14 is the *DataTypeTransformerRegistry*. As previously mentioned, it is used to keep track of the *IDataTypeTransformer* implementations per *DataSource*. The **IDataTypeTransformerRegistry** interface declares function to register these transformers and to retrieve them for a given *DataSource*. The **DataTypeTransformerRegistryHost** realizes this interface and holds the *IDataTypeTransformer* objects in a Map collection, where the key is the *DataSource* ID and the value is the object itself. In the previous section, as this *DataTypeTransformerRegistryHost* was injected into the *DataCollectionService*, it was able to retrieve the configured transformer for the given *DataSource* and use it to transform the data.

5.2.3 EventBus module

The Authorization module and the Collection module as well heavily rely on the **EventBus** to subscribe for messages and to publish their own state. In order for the whole framework to function properly, the *EventBus* implementation is critical. This module's Class Diagram is displayed on Figure 4.2 and the implementation is located in the *eventbus* package, contained by the *common* package. In the Design section, precisely in Section 4.2, the structure of the event hierarchy and the purpose of the **IntegrationEvent** and **DataSourceEvent** has already been discussed. This paragraph is dedicated for the implementation of the actual **EventBus** class and to the details of the concrete event classes. The interface **IEventBus** is used throughout the entire framework. It declares three methods. The *publish* method enables "clients" to publish an *IntegrationEvent*. The two remaining functions are both aimed at subscribing for different kinds of events. They both called *subscribe*, this the function is overloaded and the correct function call is decided by the parameter list. The two function declarations can be observed in Listing 5.13.

```
interface IEventBus {
```

```
    fun subscribe(subscriber: KClass<*>, eventType: KClass<out
        IntegrationEvent>, handler: (IntegrationEvent) -> Unit)
    fun subscribe(subscriber: KClass<*>, eventType: KClass<out
        DataSourceEvent>, dataSourceId: String, handler: (DataSourceEvent)
        -> Unit)
    fun publish(eventSource: KClass<*>, event: IntegrationEvent)
}
```

Listing 5.13: Function declarations of *IEventBus*.

The first *subscribe* function offers a way to to be subscribed to any *IntegrationEvent*. The second one is more restricted and only allows it to any *DataSourceEvent* and also requires to specify the actual ID of the *DataSource*. *DataSourceEvents* can be emitted by any *DataSource*, therefore this allows the clients to only subscribe to events emitted by a certain *DataSource* of their choosing. For better understanding, the Listing 5.14 showcases how the *OAuth2DataSource* subscribes to an *AuthorizationCodeAcquired* event, which is supposed to be *DataSource* specific, for these are only relevant for the certain *DataSources*.

```
eventBus.subscribe(
    subscriber = this::class,
    eventType = OAuth2Event.AuthorizationCodeAcquired::class,
    dataSourceId = this.getId(),
    handler = { event: OAuth2Event.AuthorizationCodeAcquired ->
        acquireAccessToken(event) } as (DataSourceEvent) -> Unit
)
```

Listing 5.14: Event subscription example.

In case of Fitbit, the *this.getId()* call results in ”fitbit”, thus this handler should only be executed if an *OAuth2Event.AuthorizationCodeAcquired* event is emitted and it contains the *DataSource* ID value as ”fitbit”. In order to achieve this behaviour The **EventBus** abstract class realizes the interface and provides implementation for the two *subscribe* methods. Its internal representation of subscription is made using the **Handler** protected inner class. It is showed on the following Listing 5.15.

```
protected data class Handler(
    val eventSource: KClass<*>,
    val handler: (IntegrationEvent) -> Unit,
    val dataSourceId: String? = null
)

protected val subscribers: MutableMap<KClass<out IntegrationEvent>,
    MutableList<Handler>> = mutableMapOf()

private fun registerHandler(eventType: KClass<out IntegrationEvent>, handler
    : Handler) {
    if (subscribers.containsKey(eventType)) {
      val handlerList = subscribers[eventType]!!
      handlerList.add(handler)
    } else {
      subscribers[eventType] = mutableListOf(handler)
```

```
15      }
16  }
```

Listing 5.15: *Handler* class implementation for the *EventBus*.

Both *IntegrationEvent* and *DataSourceEvent* subscriptions are stored in the *subscribers* Map Collection using the *Handler* classes as values. In the *Handler* class the field *dataSourceId* is nullable and only has value when it is a *DataSourceEvent*, that is how they can be differentiated. The *subscribers* Map contains a List for values, because for event there could be more than subscription. The last function in the code fragment is the *registerHandler* private method. It manages the *subscribers* by attaching the *handler* if there is already a collection registered for the specific *eventType*. Otherwise, it creates a new collection and attaches the *handler* to the new collection. There is no production ready *EventBus* implementation in the framework, because it is supposed to be implemented by the developer using the framework according to the technology stack they use. However, there is one test implementation, which is called *SingleThreadedEventBus*. This class is located in the *infrastructure* package, which will be discussed in the next Section.

5.2.4 Infrastructure

As the framework has numerous unsatisfied dependencies, the *infrastructure* package aims to provide implementations for them that can be used for testing purposes. In this chapter, the the classes that realize the required interfaces will be introduced, which can be seen on the framework's Component Diagram in Figure 4.1. Firstly, the before-mentioned **SingleThreadedEventBus**. It inherits from the *EventBus* abstract class and provides an implementation for the *publish* method. As the name suggests, it stays on the same thread and invokes the subscriptions one by one. As next, regarding the two repositories, the **InMemoryAccessParamsRepository** and **InMemoryAuthorizationStateRepository** provide in-memory implementations for their interfaces. Both of the implementations utilise a Map Collection to save the required domain classes. Lastly, there are **EmptyDataTypeTransformer** classes for each of the four concrete *DataSources*. These realize the device specific *DataTypeTransformer* interfaces and provide a simple implementation of the required transformer methods by simply returning the *ThirdPartyData* object from the argument list.

5.3 CACHET's Implementation

Now that the framework is implemented, it should be customized to meet and fulfil the requirements formulated by CACHET and connect an fitting implementation of the framework into the CANS system. An outline of the objective was already given during the design process in Section 4.5 and 4.6 and the proposed Component Diagram of the implementation was illustrated on Figure 4.20. This section is focused

on explaining the showcased components and how the CANS specific requirements were added. First of all, the package **gardener.carp-implementation** contains the Vert.x implementation. In the *src* package, the *kotlin* folder contains the source code and the *resources* folder contains configuration files, such as profile specific JSON files detailing the constants used throughout the application and logger configurations in the *logback.xml* file. The logging right now is configured to be appended to the console with the format specified in the configuration file. Moving back to the source code, the application start point is defined in the **AuthenticationModuleApplication.kt** file. It deploys, as Vert.x requires it, the **MainVerticle** class, which contains the initiation of the framework and the declaration of the service endpoints. The class' *start* function is called from Vert.x to start up the Verticle. It declares the HTTP server instance that will serve the incoming requests and that runs on the port specified in the configuration files. To be able to use the constants defined in those files, the application uses the **PropertiesConfig** class, which reads up the profile specific configuration file from the classpath according the the "*profile*" environment variable, which has to be set before the software is run. The value of the variable could be *local*, *testing* and *production*, just as the prefixes on the configuration files. However, before the HTTP servlet is started, there is one more function is called, which is the *setUp*. The whole framework is initiated here along with the declaration of the routes. The rest of the chapter is focused on how the instantiation, customisation and the interconnection of the framework modules are done.

5.3.1 Required interface implementations

As the Component Diagram states, there are eight required interfaces that need to be implemented in order for the framework to function properly. During the explanation, the CANS infrastructure, especially the running RabbitMQ instance will be references. The Deployment Diagram of the Server can be noted on Figure 4.22. First of all, the *EventBus*, the implementation utilises the **SingleThreadedEventBus** for simplicity. Later on this could be changed to use the RabbitMQ instance instead to support true messaging. For the *IDataPublisher* implementation, the **RabbitMqDataPublisher** class is used. This connects to a RabbitMQ instance by establishing a connection at class instantiation time and implements the required interface method by sending the *TransformedData* object to the RabbitMQ queue where CANS can consume it. Furthermore, there are two repositories that need to persist data. both **MongoAccessParamsRepository** and **MongoAuthorizationStateRepository** utilises MongoDB for this purpose. The *MainVerticle* instantiates a Vert.x specific *MongoClient*, which can be called to persist and query entities. This client is passed to both classes and they implement the required interface methods using the client using the NoSQL dialect. Lastly, there are the *Operators*. From the framework's perspective, four *Operators* have to be implemented. The Authorization Module requires the *IOAuth1AuthorizationOperator* and the *IOAuth2AuthorizationOperator*. The Collection Module needs an *IDataCollectionOperator*, which has to have an OAuth1 ver-

sion, which is returned by the *IOAuth1DataCollectionOperatorBuilder* and an OAuth2 version, which is made by the *IOAuth2DataCollectionOperatorBuilder*. Both versions of the Authorization and Data Collection Operators are implemented in the **OAuth1Operator** and **OAuth2Operator** classes. In the Kotlin ecosystem there any many open-source utility libraries that provide well-tested solution for OAuth operations, thus implementation by hand is not necessary and more importantly not advised especially in the cause of OAuth1 due to the presence of cryptography. Implementing the steps of OAuth protocols might lead to unintended bugs. Therefore to avoid this scenario, the library *ScribeKotlin* is used. It has support for every functionality required for the framework. However, it only supports the plain OAuth protocols. In case of Withings, as mentioned in Section 5.2.1.1, it requires certain parameters to be present during requesting OAuth tokens, thus the requesting Access Token and Refresh Token part of the library was extended to incorporate adding additional arbitrary headers. Other than this minor complication, the implementation using *ScribeKotlin* was smooth and straight-forward. Moreover, there is one more aspect to these operators that has not been mentioned yet. The question of subscription will be explained in the following Section.

5.3.2 Subscription handling

In order for the framework to collect data, something has to initiate the collection flow. The initiation is done by publishing *DataCollectionPreparationEvent* objects. This can be done manually, by a scheduled job or by the subscription services provided by the third-party APIs. Every API requires the subscription to be made in a different way. In addition, it is mainly making some network calls, so it was decided to make the subscription handling purely an implementation concern, the framework does not support hem yet. The decision will be discussed in Section 7.4. The end of a successful authorization is noted by retrieving the authentication information for a user. At this point, the software is excepted to query the third-party APIs on the user's behalf for data, so just after receiving the authentication information is a good point to make subscriptions for the user. Starting with the *OAuth2DataSources* first, in the *retrieveAccessParams* method just after the tokens are received, the function tries to make subscriptions for the given user and the given *DataSource*. Out of the three devices using OAuth2, only Fitbit and Withings possesses subscription capabilities, Dexcom does not have one.

On one hand, in case of Fitbit, it is mandatory to have a *Subscription* set up and registered on the Fitbit developer portal. Individual subscriptions for users can only be registered to one of these main *Subscriptions*. This configuration is required to be made by hand by the developer. Once the *Subscription* group is set up, the ID of the group and a unique verification code is established and as the next step, it has to be verified using the code. The verification happens in a way that Fitbit's Web API calls the application using the registered callback with

the verification code. Two calls happen, one with the correct code and one with a purposefully wrong one and the application has to respond with correct HTTP Response Code according to the excepted behaviour (204 for the right and 404 for the wrong one). If the verification was successful, the personal subscriptions can be submitted. From this, the task is pretty straight-forward. To make a subscription, a call has to be made to the following address: "*https://api.fitbit.com/1/user/-/apiSubscriptions/$userId.json?subscriberId=1*". Where the *userId* path parameter is the unique name of the subscriptions. It was chosen to made the internal identifier of the user, because that is unique. The ID of the Subscription Group is sent in the query parameter list. After the call, the subscription is created, which subscribes the user for notification of every available group.

On the other hand, Withings' subscription handling requires an entirely another approach. There is no step that requires manual configuration, but instead it requires two network calls in addition with some cryptographic operations. The first call is aimed at retrieving the cryptographic nonce (a random string of characters) from the API and then the second call makes the actual subscription with the new nonce. These will not be detailed here, because call that have to be made are already detailed in the official documentation[2]. The calls require the presence of several client specific credentials and the cryptographic operation is a SHA-256 hash of the required parameters, as a Message Authentication Code (MAC) to avoid the modification of the parameters, because they are transferred as query parameters. After the calls succeeded, the subscription is created on Withings' side. As a personal opinion, even though Withings' procedure is cumbersome to implement, it was a great to study their approach to use nonces to prevent replay attacks and MACs to avoid parameter modifications.

Furthermore, the last *DataSource* that is implemented is Garmin. It does not require any subscription submission, the ping notification endpoints have to be configured on their developer portal by hand and they are applicable for every user, therefore, there is no OAuth2 subscription making in the implementation. If there is even going to be a need for one, however, the same way could be applied here that was applied in case of OAuth2.

5.3.3 Framework initialisation

After every necessary interface is implemented, the framework service classes can be instantiated. The instantiation can also be divided into two batches. One are the services for the Authorization Module and the others for the Collection Module, apart from the *EventBus*, *OAuth1Operator* and the *OAuth2Operator*, which are needed for both. Starting with the Authorization module, with the repositories implemented, the core services can be instantiated, which are: *MongoAuthorization-*

[2]https://developer.withings.com/api-reference

StateRepository, *MongoAccessParamsRepository*, *AuthorizationStateServiceHost*, *AccessParamsServiceHost* and the *DataSourceRegistryHost*. The *DataSources* should also be created and registered here. Listing 5.16 how the *FitbitDataSource* is registered in the Registry.

```
val fitbitClientSettings = OAuth2ClientSettings(
    clientId = properties.getProperty("fitbit.client.id"),
    clientSecret = properties.getProperty("fitbit.client.secret"),
    authorizationUri = properties.getProperty("fitbit.oauth2.authorization.
        uri"),
    accessUri = properties.getProperty("fitbit.oauth2.access.uri"),
    dataUrl = properties.getProperty("fitbit.data.uri"),
    callbackUri = properties.getProperty("fitbit.client.callback.uri"))
val fitbitDataSource = FitbitDataSource(
    eventBus,
    authorizationStateService,
    accessParamService,
    fitbitClientSettings,
    OAuth2Operator(fitbitClientSettings, vertx, properties))
dataSourceRegistry.activateDataSource(fitbitDataSource)
```

Listing 5.16: Service registrations for the Authorization module.

Using the *PropertiesConfig* object *properties* the client settings can be extracted from the configuration file. With those set and the services registered, the *DataSource* can be instantiated and with the *activateDataSource* method registered. The other *DataSource* registrations are handled the same way. These are all the steps that have to be taken to get the Authorization Module operational. Moving forward with the Collection Module, its required objects are the *DataTypeTransformerRegistryHost*, *RabbitMqDataPublisher*, *OAuth2DataCollectionService* and *OAuth1DataCollection Service*. In this module, the transformers should also be registered, which is shown on Listing 5.17.

```
transformerRegistry = DataTypeTransformerRegistryHost().apply {
     registerTransformer(FitbitDataSource.DATA_SOURCE_ID,
         FitbitCarpTransformerI())
     registerTransformer(GarminDataSource.DATA_SOURCE_ID,
         GarminCarpTransformerI())
     registerTransformer(WithingsDataSource.DATA_SOURCE_ID,
         WithingsCarpTransformerI())
     registerTransformer(DexcomDataSource.DATA_SOURCE_ID,
         DexcomEmptyTransformerI())
}
```

Listing 5.17: Transformer registration for the Collection module.

The implementation also contains *DataTypeTransformer* implementation for every device, that transforms the *ThirdPartyData* into CACHET's required "data points" format. With the mentioned classes created and with the necessary connections configured, the framework is ready to serve requests. In order to do that, however, the service endpoints have to be configured.

5.3.4 Endpoint declarations

The application endpoints declares the HTTP routes on which particular services are available. The implementation, not counting one endpoint that was specifically made to handle the Fitbit subscription creation detailed in Section 5.3.2, lists three endpoints in total for handling the user enrollment and data collection procedures. One common characteristic is that they are all device and user independent. The path parameters "*dataSourceId*" and "*userId*" will be present in every case to identify the *DataSource* and *User*.

The first one is the following: "*/wearables/api/authorize/:dataSourceId/:userId*". This is the first endpoint thatinitiates the user authorization process and redirects the user to the third-party website. The *userId* here can be anything, this will be the internal identifier of the person. This endpoint also excepts query parameters. The OAuth2 *scopes* will be set here and transferred in the *OAuth2AuthorizationRequest Params* object for the *dataSource.initiateUserAuthorization* function call. CACHET's exclusive requirement was to also save the "*deploymentId*", which is also sent in the query parameters. The concept of deployments and CACHET's user-case is detailed in Section 4.5. This is completely independent from the framework and from the OAuth2 protocols, this is completely a user specific requirement, therefore it will be saved in the previously introduced *applicationData* field in the *RequestParams*. This field was designed to accommodate requirements such as this one. This way, the *deploymentId* will be saved along with the Access Parameters at the end of the authorization flow, thus it can be accessed any time. This is important during the transformation, because their format requires this data to be present in certain fields.

After the user is redirected and gave consent, the third-part Web API will call the application's callback, which will be the following endpoint: "*/wearables/api/oauth/:dataSourceId/callback*". According to the OAuth standards, the parameters, like OAuth2's Authorization Code and OAuth1's oauth_token will be transferred in the query parameters, so these need to be extracted and either a *OAuth2Event. AuthorizationCodeAcquired* or a *OAuth1Event.AuthorizedTokenAcquired* needs to be published with the *EventBus*. Other than the event publication, nothing else needs to be called.

The last endpoint is the "*/wearables/api/collection/:dataSourceId*", which initiates the data collection flow. Here, the HTTP request body will contain, most likely, the information. By retrieving the right *DataSource* from the *DataSourceRegistry* using the *dataSourceId* from the path, the method *getDataCollectionPreparationEvent-FromPing* will get the raw payload and transform it into protocol specific *DataCollectionPreparationEvent*. These can be published. Again, the user of the framework does not have to touch any part of the system besides event publication and calling one function.

It is important to mention there the design, how the *EventBus* hides the inner

workings of the system, thus the only task the developer has is to publish the correct event. The vast majority of the required work in each case is handled by the framework, the only tasks the user has are to handle the HTTP payload/query parameters/path variables. When it comes to device specific payloads in case of the collection endpoint, the transformation of the raw ping into the specific events comes also out of the box.

5.4 Deployment and Operation

In order to enable the system to collect data about the users and correctly send it to CACHET's CANS system, the implementation has to be configured and deployed into their existing infrastructure.

5.4.1 Dockerization

Before the whole server architecture of CACHET is discussed, the connection between the MongoDB, RabbitMQ and the application should be explained. Instances of the mentioned technologies are necessary to be present during runtime, otherwise the implementation cannot save nor publish data. The Deployment Diagram of the required infrastructure is displayed on Diagram 4.21. Using Docker Compose, however, launching instances of them can be easily achieved using their official Docker images available on Dockerhub. The **docker-compose.local.yml** file in the *gardener.carp-implementation/docker* package describes the set-up of the instances for local development using the *local* application profile. The MongoDB and RabbitMQ instances can be addressed using localhost and their specified ports. In addition, the docker-compose file also attaches the required configuration files and set-up scripts for each instance using *volumes* property. These files are located in the *config* folder. For RabbitMQ, the required configurations are the credentials (*rabbitmq.conf*), enables plugins (*enabled_plugins*) and the queue definitions (*definitions.json*), which will set up the queues that CANS uses, so the application can address them and send data. Moreover, for MongoDB, scripts can be found which will set up a new database in the MongoDB instance for the application (*init-mongo.sh* for production and *init-mongo-local.js* for local). Moving onto the production profile, which is actually displayed on the diagram mentioned previously, the **docker-compose.yml** file will also set up a RabbitMQ and MongoDB instance with the same configuration as before, however, this time it also starts the container of the application. In order to do so, a **Dockerfile** has to be present which describes how to build the image that will contain the software. The content of the file is showcased on Listing 5.18.

```
1 FROM gradle:7.2-jdk11 AS builder
2 COPY ./ /src
3 USER root
4 WORKDIR /src
5 RUN gradle clean shadowJar -x test
```

```
6
7  FROM openjdk:11
8  WORKDIR /app
9  COPY --from=builder /src/gardener.carp-implementation/build/libs/*.jar app.
       jar
10 RUN chown -R root:root /app
11 USER root
12 CMD echo "Starting CARP wearables service..." && exec Kotlin -jar app.jar
```

Listing 5.18: Dockerfile of the implementation.

This Dockerfile utilises a two-stage build to minimize the size of the final image by not including the source code and leftover build artifacts by building the app in the first stage, shown on the first half of the code fragment. The *gradle clean shadowJar -x test* will create a Fat JAR of the application, without running the tests because the integration tests which rely on the presence of the database and message queue cannot succeed due to the technologies not running. The second stage copies the JAR from the first stage and runs it. The production Docker Compose takes this Dockerfile and starts the container of the application using the "*production*" profile set in the *environment* variable. The Compose File also declares a common docker network called "*cans*" and, as displayed in the Deployment Diagram" each instance is configured to run on this network, so they can communicate with each other.

5.4.2 Server integration

With the production profile, the application is ready to be deployed into CACHET's Web Server. The Deployment Diagram can be found on Figure 4.22. After cloning the repository on the server and starting the *docker-compose.yml* file using the command "*docker-compose up*", the application should start as excepted. However, the application runs on the server on port 8444, but it is not available from the public network due to the default security policy being deny all. The CANS server has an NGINX[3] reverse proxy upfront, which is the main entry point, thus everything that is calling the "*https://cans.cachet.dk*" domain goes through it first and depending on the path it directs the message to the correct application. The current project needs the "*/wearables/api*" path the be opened and connected to localhost:8444, so editing the configuration file of the NGINX proxy with the code fragment shown on Listing 5.19 will enable the project to serve requests.

```
1 location ~ ^/wearables/api/(.*) {
2     proxy_pass https://$host:8444$uri$is_args$args;
3 }
```

Listing 5.19: NGINX configuration fragment.

This is the important part of the configuration that needed to be attached. With this configuration added, every request calling the "*https://cans.cachet.dk/wearables /api/...*" address will be correctly redirected to the application.

[3]https://www.nginx.com/

5.4.3 Wearables configuration

The software also has to be registered on the third-party Web Servers. Most of the companies have a developer portal where there are options to create new *applications*, after which the unique OAuth credentials are given to the user. This had to be done manually, so right now, Fitbit, Garmin, Withings and Dexcom has an application registered and their credentials are stored in the profile specific configuration files. Listing 5.16 already showcased how the properties are used to configure one *DataSource* and Listing 5.20 displays the actual properties stored for one *DataSource*, specifically for Fitbit.

```
"fitbit.client.id": "XXXX",
"fitbit.client.secret": "YYYYYYYYYYYYYYYYY",
"fitbit.oauth2.authorization.uri": "https://www.fitbit.com/oauth2/authorize",
"fitbit.oauth2.access.uri": "https://api.fitbit.com/oauth2/token",
"fitbit.data.uri": "https://api.fitbit.com",
"fitbit.client.callback.uri": "https://cans.cachet.dk/wearables/api/oauth/fitbit/callback",
"fitbit.client.subscription.code": "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ",
```

Listing 5.20: Client credentials for Fitbit.

For the production profile, the *client.callback.uri* property targets the deployed application. For local development, this is server's domain is substituted with localhost. The other properties are static, depending on weather the same application on the third-party sites is used for local development and production. The application is determined by the *client.id* and *client.secret*, because those are unique for every application.

5.5 Implementation overview

This Chapter explained how the components detailed in the Design Chapter were implemented using Kotlin and Vert.x. The implementation was able to achieve entirely what the design dictated. The current build process completes with zero warnings and errors. CACHET's specialised version of the framework is also a success and meets their requirements. The application is containerised, therefore it is ready to be deployed to a web server. The following Evaluation Chapter will discuss how the project was tested.

CHAPTER 6

Evaluation

In this chapter, we describe the different types of approaches that were used to test whether the application met the desired requirements. We briefly start with the integration tests and the requirements for applying the unit tests and finally provide the results of the application operating on the CANS server with the Fitbit, Garmin and Withings devices.

6.1 Unit/Integration Tests

First of all, the main evaluation techniques, unit and integration tests, must be discussed. At this stage, more than a hundred test cases have been implemented to test the main parts of the system. Specifically, 91 cases for the framework and 9 dedicated cases for the implementation can be found in the code base. In each of the projects, the test classes are located in the package *test*.

In the case of the framework, tests are implemented using the official Kotlin testing library[1] to process test cases in a collective manner, and additionally using the Kotlin version of Mockito[2], which is one of the top ten most used Java libraries[3] for mocking and testing, for more fine-grained statements|assertions.

The test architecture follows a certain structure. The parent class of each concrete test class is the abstract class **CoreTest**. It initializes the common concerns of the framework and declares a cleanup method that is automatically executed after each test case, called *clean*, whose task is to clean up the repositories to avoid interference between test case runs. The common concerns mostly use the test implementation of interfaces from the *infrastructure* package discussed in section 5.2.4, and some of them are mocked up with the Mockito method *spy*, e.g. *EventBus* and *DataPublisher*. This mocking is presented in listing A.1.
The OAuth specific operators are fully mocked with their own implementation in the test package to avoid network calls. Moreover, the abstract classes **OAuth1Test** and **OAuth2Test** are the ones that instantiate the protocol specific *DataSources*. One

[1] https://kotlinlang.org/api/latest/kotlin.test/
[2] https://site.mockito.org/
[3] https://www.overops.com/blog/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/

significant aspect to mention here is the *resources* package containing test data in the form of JSON files for each concrete *DataSource*, such as client setting, valid and invalid authentication information, mock ping notification and wearable data. These are read and configured in the before-mentioned test classes for each concrete *DataSource* and used throughout the test cases to simulate a real-like behaviour with real data. Furthermore, each bigger part of the system is tested, such as *AccessParams*, *DataSourceRegistry*, but most importantly, the concrete *DataSources*. Every concrete one (Fitbit, Garmin, Dexcom, Withings) has their own test class with test cases that test either an important function or an entire task flow. As an example, whether the *AccessParamsRepository* contains the authentication information at the end of an authorization flow or there were the correct events fired with the correct parameters on the *EventBus* during data collection. One of this test cases can be found on Listing A.1. These were possible due to Mockito's ability to capture function arguments during execution, so these arguments could get validated with their excepted values.

On the other hand, the implementation part contains much less test cases, since the main functionality is already tested in the framework and therefore only the own interface implementations, such as *MongoAccessParamRepository*, *MongoAuthorizationStateRepository* and *RabbitPublisher*, are evaluated. Although the same structure is found here, i.e. a main initialization superclass is declared along with a cleaning method, it differs significantly. First, the JUnit[4] integration of Vert.x is used, and the initialization consists of instantiating the class *MainVerticle* and deploying it in Vert.x. Therefore, the entire application is launched just like in a production scenario. Since the cleanup method uses a real MongoDB instance, it runs NoSQL queries to clear the specified collections before each test case, so each test case has a clean working state from the start. Speaking of MongoDB, these tests also require the database and RabbitMQ to run in Docker containers in order to be used. To avoid deleting the framework user's local database, the tests were run using the *test* profile, which connects to another test database in the MongoDB instance, so that the local data remained untouched while the tests were running. Real network calls to real web servers were not made, as a rate limiting mechanism is used in each case, which denies further requests if too many requests are made from one source and a certain limit is reached. Therefore, concrete authorization and data collection were not tested in the implementation.

Finally, a pattern can be observed in both cases regarding the structure of the test cases. Each consists of three phases. In the first phase, the state required for the scenario is established by storing or declaring test data. In the second phase, the commands to be tested are executed. The last phase is the evaluation, where the results of the execution are determined.

[4] https://vertx.io/docs/vertx-junit5/java/

6.2 Technical study

The first "*live*" test that was conducted was the "Technical Trial" that ran from October 26 to November 8 on CACHET's main web server. The purpose of this test run was to determine if the initial version of the implementation and deployment configuration was capable of performing its intended tasks. These tasks consist of the implementation being able to enroll users, receive notifications, and perform data collection for the Fitbit, Garmin, and Withings wearables. Dexcom was screened out because its services were not available to non-U.S. residents. In terms of deployment, it is necessary to verify that the Docker setup is valid and that the application can be deployed on the web server and connected to the existing infrastructure. It should be noted that CANS integration was not performed at this stage, which was explained in section 4.5.

With that said, this study only included me as the only participant handling three watches: Garmin Venu[5], Fitbit Versa 2[6] and a Withings Steel HR[7]. I had each companion application installed on my Android Smartphone and I have worn the watches interchangeably during the day.

On the 26th the application was successfully deployed to the Server. Withings integration was not yet part of the deployment due to it not being implemented yet, so Fitbit and Garmin were the only ones available. The enrollment process succeeded by calling the following two URIs: "*https://cans.cachet.dk/wearables/api/authorize/garmin/ricsi*" and "*https://cans.cachet.dk/wearables/api/authorize/fitbit/ricsi*". The internal user ID was "*ricsi*" in both cases. The logs of the successful authorization of Fitbit can be seen on Figure A.1 and the first ping notifications from Fitbit and Garmin are shown on Figures A.2 and A.3. These snippets are exported from the docker container of the application. As of the first day, the enrollment and data collection flow seemed to be working, however, other complications were discovered, such as the CANS "data point" data transformation was incorrect. This bug was found after checking the logs in CANS to see whether the data pushed to the RabbitMQ instance and consumed by CANS gets persisted or not. The day after, the second major misbehaviour presented itself. The OAuth2 access token, in case of Fitbit, is valid for eight hours. After their expiration, the tokens were correctly refreshed and saved, however, the expiration calculation, which determined whether a token was expired or not, was faulty. After applying fixes to these issues and redeploying the application, the bugs seemed to be no longer present. After issuing a command on the main PostgreSQL database of CANS, which returned the collected data points per day for the user *ricsi*, it could have been seen that on the first day 19 and on the second day 44 data points were collected from both watches. Furthermore, on 28th of November, the Withings integration was done and deployed to the Server. I have enrolled myself using the fol-

[5] https://www.garmin.com/da-DK/p/643260
[6] https://www.fitbit.com/global/us/products/smartwatches/versa
[7] https://www.withings.com/us/en/steel-hr

lowing uri: "*https://cans.cachet.dk/wearables/api/authorize/withings/ricsi3*", so the internal user ID for the Withings test user was *ricsi3*. The authorization concluded successfully and in the logs it could have been clearly seen that Withings also sends data. Important to mention that the token refreshment and data transformation fixes still held up, so no further problems were detected. With all three of the watches authorized the study ran seamlessly until the 8th of November. The logs were monitored constantly during this period and no error were found. At the end, after querying the CANS database with the query displayed on Listing 6.1 for data points for users *ricsi* and *ricsi3*, the result displayed on Figure A.4 was seen. A total of 734 data points were collected throughout the 14 days long period. In the end I started wearing the watches less and this tendency can clearly be seen on the last days.

```
select count(*), created_at::date from data_points where deployment_id = '
    ricsi' or deployment_id = 'ricsi3' group by created_at::date;
```

Listing 6.1: PostgreSQL query to get the data points per day for user *ricsi* and *ricsi3*.

In the end, the Technical Study was a success, because it demonstrated user enrollment and data collection can be achieved and the whole project can be deployed into the CANS ecosystem.

6.3 Fitbit study with CANS

Near the end of the implementation of the project and after a successful first study, the next and last evaluation milestone was the *Fitbit study* that was run from the 11th of November until the 16th of December. It was given the name "Fitbit" for the reason that only Fitbit devices were involved. CACHET was able to provide six additional Fitbit devices that were handed out to different participants. These random people were given the devices and were told to sue them as their daily drivers.

In addition to the increased number of users, the main goal of the study was see whether the fully aligned version of the project with CANS is able to authorize, collect and publish the data to the main system in the final format they require. In order to achieve this, the CANS system also had to be modified to incorporate the new functionalities. The desired flow was detailed in Section 4.5, as referenced before, however, the modifications made will quickly be summarized here as well. A new *protocol* was made especially for Fitbit that described that the main master device that will collect the data about the user is going to be a Fitbit device. CANS' invitation subsystem was expanded with this scenario, that whenever a new user is invited to a study that has a Fitbit specific protocol, the subsystem will attach an authorization link to the email that will redirect the user to the "*https://cans.cachet.dk/wearables/api/authorize/fitbit/:userId*" page, which will initiate the authorization flow in the application. In my case, the exact authoriza-

tion URI looked the following snippet: "*https://cans.cachet.dk/wearables/ api/authorize/fitbit/ aea5fe7c-30aa-4852-b49f-bd0ebabfabfb? scopes=activity,heartrate,weight, sleep,nutrition,profile,settings & deploymentId=072a7c19-4684-4093-9731-77a78b750148*". The *userId* in the previous URI is substituted with the account ID of the user in CANS and the query parameters, next to the requested OAuth2 scopes, also includes the deployment ID associated with the user. Inserting this link based on the presence of device specific protocols was the only modification that was needed on the main system's part. Garmin and Withings are also implemented in CANS the same way, however, one protocol can only handle one kind of device as of right now, thus they were not included in the study.

After the alignment, on Fitbit's developer portal an entirely new *Application* was created especially for this study. The application's details can be seen on Figure A.5. These details are included in the *production* profile of the deployed software. Furthermore, on the dashboard, a new study was created. The dashboard itself is showcased on Figure A.6. Through the dashboard, the user were enrolled one by one by email addresses. I also invited myself to participate with the Fitbit Versa 2 watch. After a successful authorization process, my watch instantly uploaded some activities data from the day after I opened the Android application, which was presented on the dashboard. Figure A.7 displays that eight activities data point that was uploaded to my deployment on 11th of November. This means that the following statements are true:

- The authorization was a success through the email, because the application was able to retrieve data with the saved authentication information.
- Fitbit notification were correctly created upon authorization, because Fitbit correctly notified the application through the notification service.
- The data was collected from the vendor and correctly pushed to the main system, because the dashboard displays it.
- The data was transformed into the right format, because the dashboard can display it categorized and it is displayed for my deployment.

As a first impression, the application seems to achieve what it sets out to do. In the following days, six more participants were rolled in in the same way. In each case, the authorization was a success and the data upload began. The application logs were constantly monitored, but no errors were found this time.

6.4 Evaluation overview

This chapter explained how the application was tested. The Unit/Integration test cases aimed at programmatically test out the main functions of the application-core.

The two tests conducted on the web server had the main purpose to test out the implementation and operation setup for CACHET. As every test case succeed and the two deployed tests were a success, it can be stated the application is capable of performing the tasks required.

CHAPTER 7

Discussion

This section provides discussion about how the initial goals were met by the final implementation, what kind of unique values are provided that elevates this project from the rest, what limitations the current version possesses and what the future plans are.

7.1 Goals and results

Section 1.4 described four goals that needed to be satisfied the end of the project. This section aims to provide a reasoning how those requirements were met.

7.1.1 How can the software be designed that capable of collecting data from various wearables?

First of all, Section 3.2 presented a handful of different wearable device provider companies and their unique requirements that should be paid attention to while designing the solution. Furthermore, Section 3.3 surveyed the field of Web APIs to assert the common characteristics found in the different wearable providers, such as the utilisation of OAuth authorization protocols and overwhelming use the JSON data format. With the possible unique requirements and market standards in mind, Section 4 provided a design that supports OAuth1 and OAuth2 based Web APIs and capable of collecting, transforming and publishing data represented in different formats. This claim is proved by the Technical test, described in Section 6.2, where three different kinds of wearable devices were used to collect data. Even in this trial, the data was sent to CACHET's main system, CANS, however, the proper integration of the two projects was not done yet. After the full integration was designed and implemented, the Fitbit study, detailed in Section 6.3, showcased how multiple participants, enrolled in an actual *Study* that was made on the Web Portal of CANS, were able to upload device data in the excepted "*data points*" format to the main system. With the two studies conducted, it is clear that the application is perfectly capable of collecting wearable device data from different providers as long as it is required and permitted.

7.1.2 How can the design offer extensibility?

On one hand, extensibility meant that more wearable devices can be integrated into the framework over time. On the other hand, even though the framework might be extensible, but the extension process must take the least amount of effort possible. To explain how this property was achieved, first the Class Diagram of the *DataSource* component on Figure 4.9 has to be mentioned. This exhaustive hierarchy of classes provides bases for wearable integrations relying on OAuth1 and OAuth2 protocols. *OAuth1DataSource* and *OAuth2DataSource* classes define everything needed to authorize an user and handle user specific authentication information, such as protocol specific tokens. By inheriting from one of these classes, the entirety of the protocol specific logic is given for the user. However, as Section 3.2 detailed, some of the wearable providers might dictate the presence of arbitrary parameters during authorization phases. To exemplify this, the integration of Fitbit and Withings have to be highlighted. Withings required the an one header, namely the *action* header, which must have had the value *requesttoken* during getting the access tokens and refreshing them. On the contrary, Fitbit does not specify anything that is not default OAuth2 requirement. This problem was solved by introducing the *AuthorizationReqyestParams* class, detailed in Section 5.2.1.1, as they are used to provide a way for the user to specify parameters like this. The highlighted Listing 5.1 in the Section displays how it is implemented in the *WithingsDataSource*. Apart from the custom parameters requirement, the actual integration of the devices has to be emphasized. There are currently four devices implemented, Fitbit (*FitbitDataSource*), Garmin (*GarminDataSource*), Withings (*WithingsDataSource*) and Dexcom (*DexcomDataSource*). Each of the before-mentioned classes inherits from the protocol specific *DataSource* and implements the remaining abstract functions. However, neither of them contains any implementation that is not entirely provider dependant. Therefore, when new devices need to be integrated into the system, only the most necessary, provider specific details have to be implemented, such as the representation of OAuth2 scopes, supported *DataTypes* and the provider specific notification ping extraction. The integration of four completely independent wearable device provider serves as proof that the design succeeds at achieving the extensibility property in the most convenient way possible.

7.1.3 How can the design offer customisability?

The second most important property of the system is the customisability. The initial description of the goal was to keep the project technology free. To achieve this requirement, the design relies on the design decision of the framework based approach and following the hexagonal architecture, detailed in Section 3.4. Firstly, the Component Diagram of the framework on Figure 4.1 needs to be recalled. The Diagram displays numerous required interfaces that need to be implemented when using the framework. Every dependency defines some functionality where an actual technology is required. For instance, the repositories except to be provided with an database implementation, such as MongoDB. The operators require network communication implementations.

Lastly, the publisher can be provided with anything the user desires. The most significant aspect of having these required dependencies is that specific technologies are not hard-wired into the solution. The only actual dependency the framework has is the serialization library, Jackson, which was essential for being able to handle JSON data. The dependencies are dependency injected in the necessary places, therefore the framework is able to work with the functions declared by the interfaces. Furthermore, the requirements mentioned the need for customisable data transformation techniques. As detailed at the end of the Section 4.4, the solution took this into consideration and provided a way to register new transformers without modifying the core logic. To prove that the framework stayed successfully technology independent and it can be implemented to meet arbitrary requirements, CACHET's implementation has to be highlighted. Beforehand, it has to mentioned that the framework was not influences by the specific requirements formulated by CACHET. With that said, their conditions were that the data had to be transformed into the "*datapoints*" format and the transformed data has to be pushed to the established RabbitMQ instance. By providing custom transformers and an implementation for the RabbitMQ integration and injecting these classes into the framework, their requirements were met. The way these implementations were provided, just as the MondoDB repositories, any other technology could have been used and injected in their places. This proves that the framework achieves customisability and can be implemented in a way to meet one's requirements.

7.1.4 How can the solution solve CACHET's unique requirements?

The last goal was whether the project meets CACHET's requirements and can provide a solution for their problem. The previous sections already mentioned how most of their requirements are satisfied. However, the main proof of the success is the Fitbit study detailed in Section 6.3. In order to achieve this, their main system also have to modified with device specific *Protocols* and new email templates that contain an authorization link that redirects the user to the authorization endpoint of the project running on the web server. The detailed design of the integration is discussed in Section 4.5. As the result of the Fitbit study shows, the requirements were met by the project with the minimal modification on the existing system's side possible.

In the end the goals the project set out to achieve were met. A general purpose software framework was created that can be used for data collection from different wearable technology providers. The framework can also be freely shaped during the implementation process to satisfy special requirements, just as it has been done in CACHET's case. The next section will talk about how this solutions can provide unique value against its competition.

7.2 Implications

In the summary of the related work Section 2.5, it was stated that the competition mostly succeed in extensibility, they seriously lack in customisability. This project, while supporting the same if not a higher level of extensibility than its competitors, offers customisability in a way that is not present in any other before-mentioned projects. Every technological dependency can be freely chosen by the user as well as the data transformation techniques, which is not present at all in case of the rest. In addition, the framework can also be implemented as part of an existing project as well as implemented on its own, such as in CACHET's case. By focusing solely on solving the problem of collecting data from wearable vendors and not forcing an entire platform on the user, just like how AWARE and RADAR-base do, it stays out from the crowd as a lighter-weight solution. Lastly, SHIMMER is still a great solution, which it also focuses on solving the collection problem, it is a complete package with hard-wired dependencies. It can be fired up out of the box and will complete the task at hand and collect the data in an Open mHealth format. If there is no need for heavy customisation and different data formats, SHIMMER can get the job done. Nevertheless, if great customisation capabilities and custom data transformation are requirements, this project is preferable. In the end, the project proposes unique values that makes it a great competitor in its field.

7.3 Limitations

This section describes some of the limitations currently identified in the project. The limitations are split into two categories, one for the framework and one for the implementation.

7.3.1 Limitations of the framework

First of all, the most obvious limitation of the framework is the only dependency it has, **the serialization library**, Jackson. It is one of the most popular serialization libraries in the Java ecosystem, right next to Gson[1] and the official default serializer of the Spring ecosystem. The problem is not about the choice of the library, but about that a specific version is configured in the project. A scenario can present itself that an existing project is using a different version of Jackson and the developer decides to implement the framework as part of their existing one. Thus, it will lead to library version mismatch. The existing project ought to use the same or higher version of Jackson to be able to integrate the framework. It might force the developer to bump the version of their library. On the other hand, if a newer version is used, even though it is highly unlikely, but the implementation of the library's methods

[1] https://github.com/google/gson

that are used by the framework might change and might contain some undesired side effects that could cause unintended bugs. However, the test-base is always there to filter out cases, such as this one.

Secondly, the **subscription handling** is not part of the framework. This is an important aspect of the Web APIs and most of the time the use of such a service is preferable over a polling mechanism. However, personally I am still not convinced weather it should be part of the framework, because it is vastly different in every case. Among the wearable provides implemented, not even one is similar to the others in any way. Garmin does not require anything to be handled in the code-base. On the contrary, Fitbit dictates separate subscription calls for each different user. Additionally, Withings also requires network calls, multiple ones. The requirements of each case vastly differ from one another. There are also cases, such as Dexcom, which does not posses a subscription mechanism. The framework was designed in a way that the data collection can be initiated by anything, only the right event has to be published to the Event Bus. This event can be omitted by the subscription service or by a scheduled job, totally depending on the choice of the developer using the framework. This is why the subscription handling is mainly implementation concern, however, the framework could provide some helper functionality.

Lastly, there is no functionality to **remove users** from the system. This is a serious missing feature, because users must be able to be removed. This idea will be discussed further in the next, *Future Work* section.

7.3.2 Limitations of the implementation

The implementation introduces some specific limitations on the top of the ones introduces in the previous section. The first limitation that needs to be discussed is a continuation of a before-mentioned problem, the **subscription handling**. The subscriptions are created once the user is successfully is authorized. This functions as it is supposed to do so, however, no logs are kept of the subscriptions per user. Right now, the only user specific subscription creation is in Fitbit's case and every subscription uses the user's internal identifier as unique identifier, therefore it is manageable. In addition, once the subscriptions are created, the implementation does not delete them, because there is no functionality to remove user from the system yet. This will also be discussed further in the *Future Work* Section.

Moreover, the personal **tokens of each user are stored in the database in plain-text**. Even though the database engine protects the documents, if an attacker gains access to the database it means that the tokens are compromised and can be used to request data from the third-party Web APIs. Luckily, the data on itself cannot be linked to a specific person without further information, because the identifier of the user is a *Deployment ID* in the CANS system. However, if more security measures

must be taken, the data can always be encrypted. This will take its toll on the performance due to the additional cryptographic operations, but at least the data will not be stored in plain-text anymore. As a sidenote, Garmin in the notification ping sends out the long-lived OAuth1 access token of the user in the payload. Even though the communication happens over HTTPS, it seems insecure. For one, the configuration of the ping endpoints are done on the developer portal. If that gets compromised and the callback URIs are modified, the attacker gets the tokens straight from Garmin. Therefore, the security does not only depend on the application, but on the vendor as well.

When it comes to security, it is also important to mention that the endpoints of the application are not protected in any way. Lack of authentication means that they can be freely invoked by someone who is aware of the endpoint paths. If an attacker gains intel on one of the users enrolled in the system, they could simulate valid ping notifications from one of the vendors, for instance Fitbit. As a consequence, the application would try to collect the data from Fitbit, since from its perspective, the notification is valid. If the attacker replays this attack multiple times, this could result in blocking the application from collecting data by the vendor's security mechanism. To avoid this problem, it is advised to modify the server's firewall settings to only allow ping notifications to be only received from the official address of the third-party vendor. The authorization endpoint in CACHET's case has to be left open to the public due to the email registration links. They cannot be pinpointed to a static location.

Lastly, even though it is not an actual limitation now, but **idempotent event handling** has to be mentioned. If a message broker is used to distribute the events, duplicate message delivery could happen due to network or other errors. If duplicate messages are delivered and the implementation does not filter them out, it can lead to bugs. For instance, assuming that the current implementation of the MongoDB repository is used and a real message broker, if an OAuth2 *AuthorizationCodeAcquired* event is delivered twice, the framework will attempt to get the tokens twice as well. If both of them succeed, that means two entries will be saved for the given user in the database, thus it will lead to conflict. The current implementation uses the *SingleThreadedEventBus*, therefore networks errors will not happen. However, it had to be mentioned that the developer implementing the framework must pay attention to these problems when implementing the *publish* function of the Event Bus. The events are already equipped with unique identifiers and creation timestamps to support filtering.

7.4 Future Work

This section is dedicated to discuss the plans for the future. One project cannot be truly over, there is always some aspects left to refactor or improve. Apart from these,

the following bigger tasks are definitely worth considering.

Firstly, the biggest open question, the state of subscription handling. If it will be decided that it should be part of the framework, first of all, it has to be designed. Since most of the time it requires several network calls, it is most likely should be handled just how the OAuth networks calls, with operators. Every wearable device must have its own operator, since every one of them requires the subscriptions to be managed in a different way. After the subscriptions are created, the framework should keep track of them in a separate repository. If it is decided that it stays out of the application core, the implementation can handle it in the most convenient way. In the current implementation, the cancellation of the created subscriptions are missing. This feature is one of the most important to-do tasks, along with removing users from the system.

The removal of users is needed in the framework as well as in CACHET's implementation. In the *studies* they ran there is a functionality to stop *deployments*. Once they are stopped, the data collection in this project must also be stopped. As of right now, this feature is missing. One could argue that when the user stops using the smartwatch they were given or the consent is removed from their account which enables the project to collect data about them, the notifications will no longer be sent to the project, therefore no new data will be collected. However, this is not a proper solution. Once the deployment is stopped in CANS, it has to send out a new message through the RabbitMQ to the project and the implementation has to act upon it by deleting the subscription and removing the authentication information stored about the user. The message sending is completely CACHET specific requirement, however, the reaction to the message, the removal of subscription and user information is a common concern.

Apart from the missing features, integration of more devices should also be done, starting with the devices already analyzed in Section 3.2. SHIMMER still offers a higher variety of devices, therefore it is essential to integrate more to stay competitive. In addition, default transformers could also be provided in framework, such as the Open mHealth format transformers.

Integrating more devices and transformers are rather nice-to-have options than essential. The most significant tasks for the future are the subscription handling and the user removal.

CHAPTER 8

Conclusion

This thesis project involved the design, implementation and operation of a service that is capable of collecting data from various third-party Web APIs. First of all, a survey of the existing solutions was conducted which resulted in the identification of properties that are missing from the analysed systems, such as support for customisation and extensibility. The project focused on providing a solution that could improve upon its competitors and offer a viable alternative.

After the goals were identified, the Analysis Chapter provided a comparison among the possible technological choices the project could utilise for designing and implementing the software. In addition, a comprehensive analysis of the Web APIs of existing wearable device providers were given in order to identify common and unique concerns, which contributed to the generic design.

Arguably the most important pieces are the Design and Implementation Chapters. The Design showcased the architecture of the framework, implementation and deployment of the project through several UML diagrams and reasoned about the choices made. The explanation started from the highest level of abstraction with Component Diagrams and ended with detailed Activity Diagrams that focused on the most important functionalities of the system. Furthermore, the Implementation Chapter showed how the components and classes were implemented with the chosen technologies, such as Kotlin, Vert.x and Docker.

After the implementation of the designed software was completed, the Evaluation chapter detailed how the solution was tested. The exhaustive set of unit and integration tests were introduced as well as the two trials conducted on CACHET's web server. Both the trials can be considered a success and every test case passes, therefore it can be stated that the software works as intended.

Lastly, a discussion was given about the current state of the project, how it successfully met the requirements stated by CACHET and how it provides improvements over the current existing solutions. Additionally, future work recommendations were also detailed. Overall I believe the project achieved what it had to and I had a blast doing it. I think it was a great project where I could learn a lot and I could proudly present it as the final conclusion of my studies here at DTU.

APPENDIX A

Figures and Listings

```
\\ declaration of spying EventBus
protected val cleanEventBus: IEventBus = SingleThreadedEventBus()
protected var spyingEventBus: IEventBus = spy(cleanEventBus)

@Test
fun successfulDataCollectionEventIsFiredUponSuccessfulCollection() {
    val userId = TestProperties.FITBIT_TEST_USER_EXTERNAL_ID
    registerFitbitUserFor(userId)

    // Fire preparation events
    val notificationText = fitbitActivitiesPing.toString()
    val events = fitbitDataSource.getDataCollectionPreparationEventFromPing(
        notificationText)
    assertEquals(1, events.size)
    events.forEach { event -> spyingEventBus.publish(this::class, event) }

    // Capture the events that were published
    val argumentCaptor = com.nhaarman.mockitokotlin2.argumentCaptor<
        IntegrationEvent>()
    verify(spyingEventBus, atLeast(3)).publish(any(), argumentCaptor.capture
        ())

    // Check if a data collected event is fired
    val executedEvent = argumentCaptor.allValues.filterIsInstance<
        DataSuccessfullyCollectedEvent>().firstOrNull()
    assertNotNull(executedEvent)
    assertEquals(userId, executedEvent.userId)
    assertEquals(FitbitDataSource.DATA_SOURCE_ID, executedEvent.dataSourceId
        )
    assertEquals(FitbitDataCollectionType.ACTIVITIES, executedEvent.dataType
         as FitbitDataCollectionType)
}
```

Listing A.1: Example test case for Fitbit.

```
Authorization request initiated for fitbit/ricsi
Access params not found for fitbit/ricsi.
Redirecting user to: https://www.fitbit.com/oauth2/authorize?response_type=co
pe=activity%20heartrate%20weight%20sleep%20nutrition%20profile%20settings&sta
Authorization state saved for fitbit/ricsi.
Authorization Callback called for fitbit
Authorization state found for id b48fc0e8-9643-4361-8edf-dbaea6b460a0.
OAuth2 access params are retrieved from operator for fitbit/ricsi.
Authorization state saved for fitbit/ricsi.
Access params saved for fitbit/ricsi.
(HTTP 200) Fitbit notification created for user ricsi
```

Figure A.1: Successful Fitbit Authorization log for user *ricsi* in the Technical Study.

```
Data collection callback called for fitbit
Extracting collection event for Fitbit from [ {
  "collectionType" : "activities",
  "date" : "2021-10-26",
  "ownerId" : "9LD6WW",
  "ownerType" : "user",
  "subscriptionId" : "ricsi"
} ]
Access params found for fitbit/9LD6WW.
OAuth1 Data collection event published for https://api.fitbit.com/1/user/-/activities/date/2021-10-26.json
OAuth2 data successfully collected from operator for fitbit/ricsi.
New datapoint successfully published to third-party-queue-development.
```

Figure A.2: Fitbit ping notification in the Technical Study.

```
Data collection callback called for garmin
Extracting collection event for Garmin from {
  "dailies" : [ {
    "userId" : "fa70fafa-170a-4693-8efd-d79379181d92",
    "userAccessToken" : "5c26f276-c1f5-4916-81ec-237b9f5cc51a",
    "uploadStartTimeInSeconds" : 1635264850,
    "uploadEndTimeInSeconds" : 1635264850,
    "callbackURL" : "https://apis.garmin.com/wellness-api/rest/dailies?uploadStartTimeInSeconds=1635264850&uploadEndTimeInSeconds=1635264850"
  } ]
}
Access params found for garmin/5c26f276-c1f5-4916-81ec-237b9f5cc51a.
OAuth1 Data collection event published for https://apis.garmin.com/wellness-api/rest/dailies?uploadStartTimeInSeconds=1635264850&uploadEndTimeInSeconds=1635264850
OAuth1 data successfully collected from operator for garmin/ricsi.
New datapoint successfully published to third-party-queue-development.
```

Figure A.3: Garmin ping notification in the Technical Study.

	count	created_at
1	19	2021-10-26
2	129	2021-10-27
3	89	2021-10-28
4	86	2021-10-29
5	58	2021-10-30
6	57	2021-10-31
7	93	2021-11-01
8	45	2021-11-02
9	56	2021-11-03
10	47	2021-11-04
11	36	2021-11-05
12	6	2021-11-06
13	13	2021-11-08

Figure A.4: Results of the Technical Study.

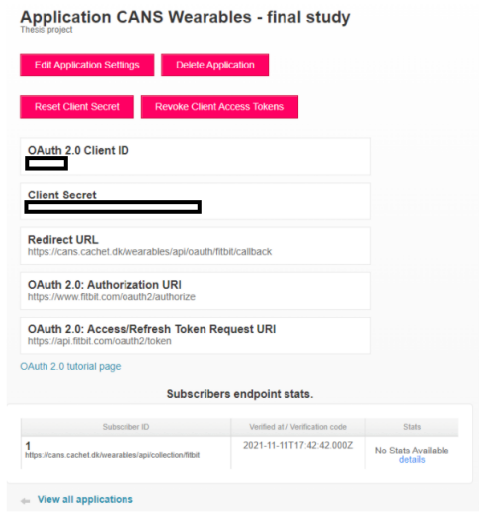

Figure A.5: Application details for Fitbit in the Fitbit Study.

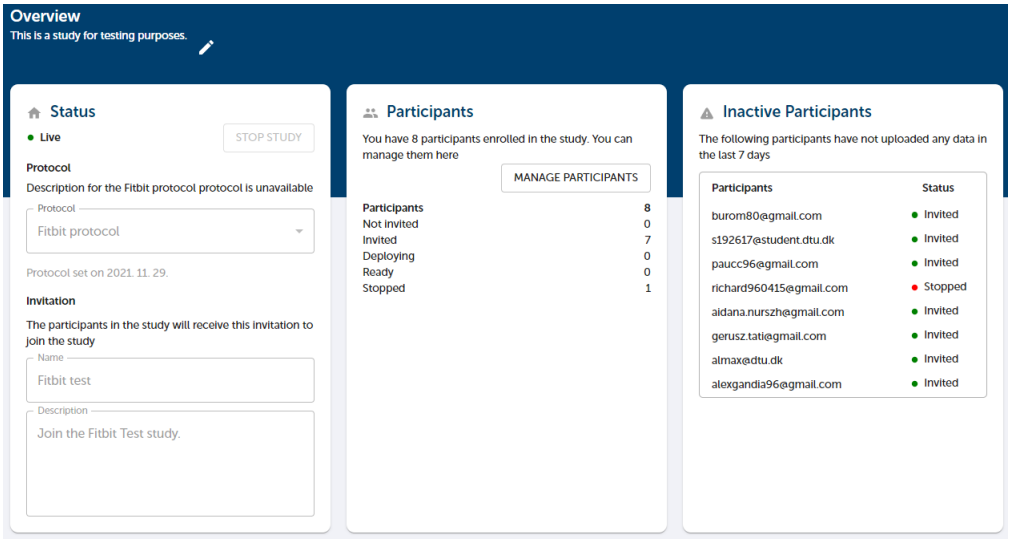

Figure A.6: Study dashboard of the Fitbit Study.

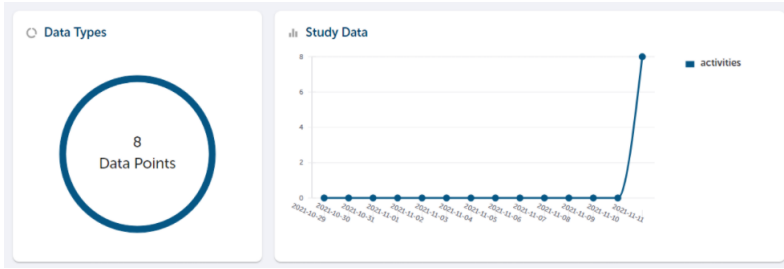

Figure A.7: My participantion's dahsboard on the first day of the Fitbit Study.

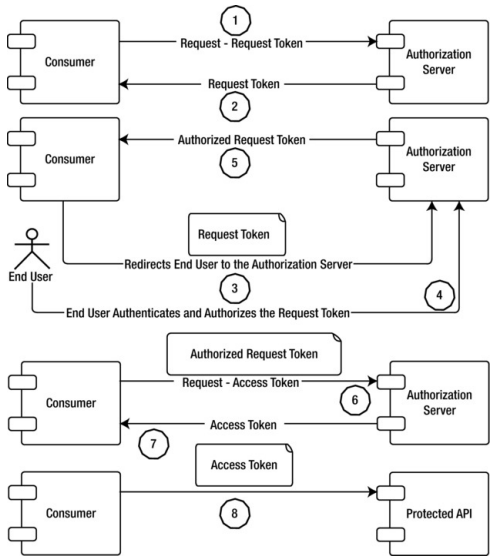

Figure A.8: OAuth1 token exchange.

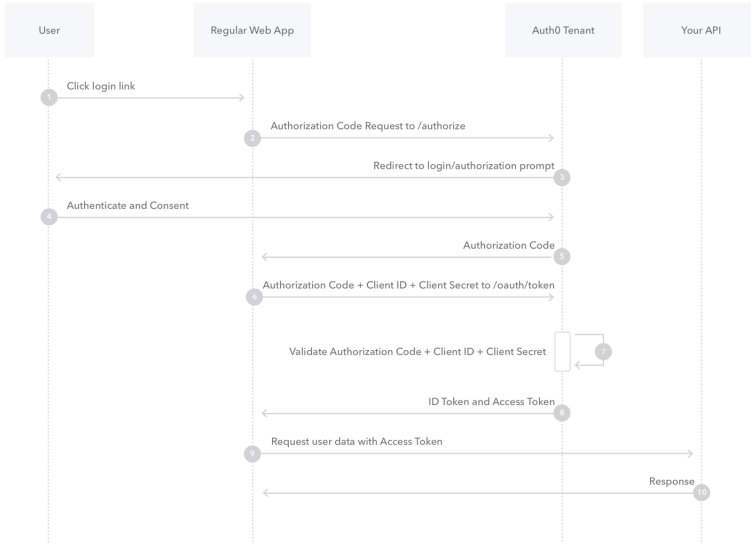

Figure A.9: OAuth2 Authorization Code grant.

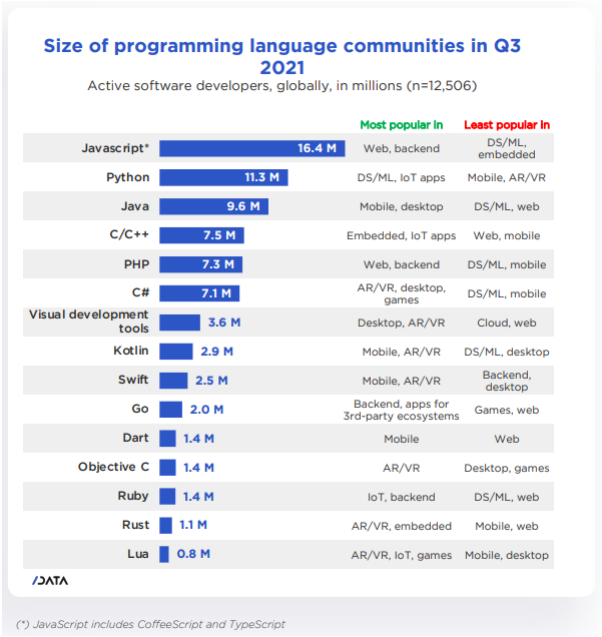

Figure A.10: Most used programming languages in 2021.

APPENDIX B

Authorization

Before the OAuth protocols are explained, the definition of the word *Authorization* needs to be clarified: Official permission for something to happen, or the act of giving someone official permission to do something[1]. This definition perfectly summarizes up the purpose of an authorization mechanism. In the project's case, it needs to get permission from the user to collect their data. Now that it is clear, OAuth is also an authorization protocol for establishing identity management standards across services[26]. The protocol proposes a generic framework, which can be used to grant permission to clients to access protected resources on behalf of the resource owner without revealing the owner's secrets.[27, 28]. Both versions are token-based protocols. Tokens are a string of characters that are established by authorization systems. These units are most-likely shared between multiple modules, so every token has to uniquely identifiable and must be not repeated. The token's purpose is to hold information about the user or entity they represent. For instance, typical information would be the identifier of the token, what it can be used for and for how long it is valid[29]. Tokens are issued at the end of the authorization session and those can be used to access the protected resource.

B.1 OAuth1

OAuth 1.0 was the first step to have a standard identity delegation system[30], which can solve the problem of delegated access to a protected resource. Version 1.0 of the protocol was first published in October of 2007[31]. However, an attack was found in this version, which is detailed in the official report[2]. After fixing the mistake, a revision was made public in June of 2009 and was called Revision A. After that, OAuth 1.0A was the default version of the protocol that is used throughout the industry. Figure A.8[30] showcases how the authorization flow is handled in the first version. The *Three-legged flow*[3] is illustrated clearly on the picture. The exact parameters required in each step is well-documented in the official RFC[31], they will not be detailed here[4].

[1] https://dictionary.cambridge.org/dictionary/english/authorization
[2] https://oauth.net/advisories/2009-1/
[3] https://oauth1.wp-api.org/docs/basics/Auth-Flow.html
[4] https://datatracker.ietf.org/doc/html/rfc5849#section-2

The first phase is the **Temporary Credentials Acquisition** (Step 1, 2), where The client retrieves temporary credentials from the OAuth1 Authorization server.
Secondly, the **Authorization** phase follows (Step 3, 4, 5). The resource owner authorizes the temporary credentials acquired in the previous phase through the Authorization server and the client gets back the authorized credentials.
Lastly, the **Token Exchange** phase happens (Step 6, 7). The client exchanges the authorized credentials to a set of token credentials. This concludes the authorization flow, because the token credentials can be used to access the protected resources on the server.

In case of OAuth1, the token on itself is not sufficient to access the resources. Many other parameters ought to be provided and most importantly, a *signature*[5] of those parameters as well, which is a hash of the concatenated parameters. Therefore, it is highly advised to use a well-tested library instead of implementing the steps manually when it comes to implementation. Furthermore, the tokens themselves are long-lived tokens. In the specification, there is no notion of token refreshment, yet they can expire if configured to do so. Most of times, however, OAuth1 tokens posses an infinite life spam.

One more important aspect to mention in the end is the state handling of the protocol. The whole process consists of three steps, which also includes asynchronous communication (phase 2, when the user is redirected), therefore it is essential to store state information. The protocol definition states that when the user is redirected and a callback URL is provided (as the definition dictates), it might contain client provided query parameters. These parameters must be preserved and returned to the client as they are. Thus, state information can be stored among the query parameters.

B.2 OAuth2

OAuth 2.0 improves upon its predecessor in many ways and renders it obsolete. Version 2 of the specification builds on the OAuth 1.0 deployment experience, as well as additional use cases and extensibility requirements gathered from the wider IETF community. The OAuth 2.0 protocol is not backward compatible with OAuth 1.0[32]. The new version introduces entirely new terminology and concepts, such as distinct authorization grant flows and the Access/Refresh token pair. An authorization grant is a credential representing the resource owner's authorization (to access its protected resources) used by the client to obtain an access token[6]. The four new types of this grant are: *Authorization Code*, *Implicit*, *Resource Owner Password Credentials* and *Client Credentials*. The description of the mentioned grants can be found in the official RFC. One of the most used grant is the Authorization code. This could have been

[5] https://datatracker.ietf.org/doc/html/rfc5849#section-3.4
[6] https://datatracker.ietf.org/doc/html/rfc6749#section-1.3

noted in the previous section, when every wearable Web API implemented it. Figure A.9 illustrates how the authorization is conducted. In this scenario, obtaining a token includes only two steps. Firstly, the user is redirected to the Authorization Server where they can give consent. After they granted permission, the client is contacted by the Authorization Server trough a callback and given an *authorization code*. In the next step, this code can be exchanged to get tokens. OAuth2, as mentioned before, introduced the notion of Access and Refresh tokens. Access tokens are short-lived tokens that serve as credentials to access protected resources. On the other hand, Refresh tokens are long-lived ones, who's purpose is to obtain new access tokens when they are expired. It is important to mention that the presence of refresh token is optional, according to the protocol.

OAuth2 also brings in the concept of *scopes*. The scopes are used to limit the "power" of tokens. They can be specified during the authorization process and at the end of the flow the tokens will have the requested scopes associated with them. Protected resources can have scopes defined for them and only those access token can retrieve the resources which has the specific scopes required by the resource.

Lastly, the state management is also discussed here. In this case, the specification explicitly declares a *state* field[7], which can be used to store state information and it is ensured that this parameter will be returned to the client upon redirection.

B.3 Comparison

This section will provide a brief overview of the two protocols and their relationships[8]. As mentioned previously, version 2 is not backwards-compatible and their implementation is vastly different, even though they set out to solve the same problem. The second version seems to improve upon the first in almost every aspect. First and foremost, it eliminates the need of complex cryptographic requirements formulated by OAuth1 during authorization and accessing resources. OAuth2's access token can simply be included in the request without anything else and the resource can be accessed. Mentioning the cryptographic operations, executing them consumes computing power, especially at scale. In addition, since client credentials are utilised in every request to make signatures, the resource servers also have to know them in order to validate the requests. On the contrary,OAuth2 only uses client credentials during the authorization phase. Therefore, scalability favors the second version. Furthermore, the first version only offered a single way to obtain authorization. In comparison, OAuth2 lifts this restriction by having four different *authorization grants* and the developers can choose whichever fits their needs. Lastly, the tokens are different in each version. While OAuth1 utilises extremely long lived tokens. Once its

[7] https://datatracker.ietf.org/doc/html/rfc6749#section-4.1.1
[8] https://www.oauth.com/oauth2-servers/differences-between-oauth-1-2/

expired, the user has to be authorized again. OAuth2 solves this issue by introducing a short-lived and a long-lived token pair. The short-lived access token is used to access resources and once it's expired, the long-lived refresh token can be used to get a new pair. Thus, the user does not have to be authorized again, only if the long-lived token also expired due to not using the issued tokens for a longer period of time.

Even tough from this summary it seems like the second version is superior, the first also has its merits. For instance, assuming that the authentication tokens are stolen during a communication over the public internet, an attacker cannot do anything with an OAuth1 token without the secret and other client information. However, the OAuthh2 token can freely be used to access the protected resource. OAuth1 is still used in the industry despite its drawbacks. The tech-giant Twitter still offers ways to authorize with OAuth1 1.0A[9]. In the end, the use of OAuth2 is advised, because the first version was marked obsolete. Additionally, it is being constantly developed and upgraded and with a correct implementation, it can offer a high-level of security[33].

B.4 OAuth libraries

Implementing OAuth authorization is a common requirement when it comes to development and there are numerous libraries for every platform. Implementing OAuth functionalities by hand is error-prone, because the flows are complex and rudimentary implementation without proper testing might lead to unintended bugs. Thus, well-tested library is necessary. For the Java ecosystem only, there is a wide variety of libraries available[10]. Out of the bunch, there are many which are framework specific or has too many dependencies. One project in particular seems to stand out from the crowd of libraries that tries to stay framework independent, ScribeJava[11]. The library has by far the most amount of stars on GitHub and seems to be the most popular choice by searching them on popular online communities, such as StackOverflow[12]. The popularity means that it is widely used and has the most support online, which is a huge benefit when it comes to finding errors. Furthermore, it is a library that is able to handle OAuth 1.0A as well, which will be useful . Therefore, for its popularity, lightweight nature and feature set, the library ScribeJava is chosen to be used during implementation.

[9] https://developer.twitter.com/en/docs/authentication/oauth-1-0a
[10] https://oauth.net/code/java/
[11] https://github.com/scribejava/scribejava
[12] https://stackoverflow.com/

APPENDIX C

API documentation

This section includes the API documentation for CACHET's implementation. The company uses Postman[1] as their main API documentation tool, therefore it was done using their platform. A new collection was created for containing the endpoint descriptions, called *Wearables*. The Listing C.1 below displays the collection exported as JSON in the v2.1 format[2]. This JSON can be imported into a Postman Client and the collection will be constructed.

The "*item*" key contains an array of the endpoints. There are two types in total: Data Collection and Authorization endpoints. The "*name*" field indicates which is which. Every entry details the exact URI that has to be called with an example payload if needed and a description of the request.

```
{
  "info": {
    "_postman_id": "e685f23c-bc91-4882-a306-c1672f3411ac",
    "name": "Wearables",
    "schema": "https://schema.getpostman.com/json/collection/v2.1.0/
        collection.json"
  },
  "item": [
    {
      "name": "Collect Fitbit data",
      "request": {
        "method": "POST",
        "header": [],
        "body": {
          "mode": "raw",
          "raw": "[\n   {\n      \"collectionType\":\"activities\",\n
              \"date\":\"2021-10-26\",\n      \"ownerId\":\"9LD6WW\",\n
                  \"ownerType\":\"user\",\n      \"subscriptionId\":\"ricsi
              \"\n   }\n]",
          "options": {
            "raw": {
              "language": "json"
            }
          }
        },
        "url": {
```

[1] https://www.postman.com/
[2] https://blog.postman.com/travelogue-of-postman-collection-format-v2/

```
          "raw": "https://localhost:8444/wearables/api/collection/fitbit",
          "protocol": "https",
          "host": [
            "localhost"
          ],
          "port": "8444",
          "path": [
            "wearables",
            "api",
            "collection",
            "fitbit"
          ]
        },
        "description": "This endpoint endpoint initiates the data collection
             process for Fitbit. The body is the official structure of a
            ping notification sent by Fitbit. This structure must be
            followed if the collection needs to be initiated by hand.\n\n
            --------\n\nSuccessful response: \n\n- `204`: No Content.\n\
            nUnsuccessful response:\n\n- `500`: Internal Server error, when
            the Data Source is not found."
      },
      "response": []
    },
    {
      "name": "Collect Garmin data",
      "request": {
        "method": "POST",
        "header": [],
        "body": {
          "mode": "raw",
          "raw": "{\n  \"bodyComps\" : [ {\n    \"userId\" : \"fa70fafa-170a
              -4693-8efd-d79379181d92\",\n    \"userAccessToken\" : \"5
              c26f276-c1f5-4916-81ec-237b9f5cc51a\",\n    \"
              uploadStartTimeInSeconds\" : 1635176900,\n    \"
              uploadEndTimeInSeconds\" : 1635176900,\n    \"callbackURL\" :
              \"https://apis.garmin.com/wellness-api/rest/stressDetails?
              uploadStartTimeInSeconds=1634220000&uploadEndTimeInSeconds
              =1634221200\"\n  } ]\n}",
          "options": {
            "raw": {
              "language": "json"
            }
          }
        },
        "url": {
          "raw": "https://localhost:8444/wearables/api/collection/garmin",
          "protocol": "https",
          "host": [
            "localhost"
          ],
          "port": "8444",
          "path": [
            "wearables",
            "api",
            "collection",
```

```
            "garmin"
          ]
        },
        "description": "This endpoint endpoint initiates the data collection
             process for Garmin. The body is the official structure of a
            ping notification sent by Garmin. This structure must be
            followed if the collection needs to be initiated by hand.\n\n
            --------\n\nSuccessful response: \n\n- `200`: No Content.\n\
            nUnsuccessful response:\n\n- `500`: Internal Server error, when
            the Data Source is not found."
      },
      "response": []
    },
    {
      "name": "Collect Withings data - specific date",
      "request": {
        "method": "POST",
        "header": [],
        "body": {
          "mode": "raw",
          "raw": "userid=27444008&appli=16&date=2021-10-28",
          "options": {
            "raw": {
              "language": "text"
            }
          }
        },
        "url": {
          "raw": "https://localhost:8444/wearables/api/collection/withings",
          "protocol": "https",
          "host": [
            "localhost"
          ],
          "port": "8444",
          "path": [
            "wearables",
            "api",
            "collection",
            "withings"
          ]
        },
        "description": "This endpoint endpoint initiates the data collection
             process for Withings. The body is the official structure of a
            ping notification sent by Withings. This structure must be
            followed if the collection needs to be initiated by hand.\n\n
            --------\n\nSuccessful response: \n\n- `200`: No Content.\n\
            nUnsuccessful response:\n\n- `500`: Internal Server error, when
            the Data Source is not found."
      },
      "response": []
    },
    {
      "name": "Collect Withings data - period",
      "request": {
        "method": "POST",
```

```
        "header": [],
        "body": {
          "mode": "raw",
          "raw": "{\n    \"userid\": \"27444008\",\n    \"appli\": \"54\",\n
                  \"startdate\": \"1634824501\",\n    \"enddate\":
              \"1634770861\"\n}",
          "options": {
            "raw": {
              "language": "json"
            }
          }
        },
        "url": {
          "raw": "https://localhost:8444/wearables/api/collection/dexcom",
          "protocol": "https",
          "host": [
            "localhost"
          ],
          "port": "8444",
          "path": [
            "wearables",
            "api",
            "collection",
            "dexcom"
          ]
        },
        "description": "This endpoint endpoint initiates the data collection
             process for Garmin. The body is the official structure of a
            ping notification sent by Garmin. This structure must be
            followed if the collection needs to be initiated by hand.\n\n
            --------\n\nSuccessful response: \n\n- `200`: No Content.\n\
            nUnsuccessful response:\n\n- `500`: Internal Server error, when
            the Data Source is not found."
      },
      "response": []
    },
    {
      "name": "Collect Dexcom data",
      "request": {
        "method": "POST",
        "header": [],
        "body": {
          "mode": "raw",
          "raw": "{\n    \"user_id\": \"userid\",\n    \"data_type\": \"egvs
              \",\n    \"date\": \"2021-10-25\"\n}",
          "options": {
            "raw": {
              "language": "json"
            }
          }
        },
        "url": {
          "raw": "https://localhost:8444/wearables/api/collection/dexcom",
          "protocol": "https",
          "host": [
```

```
            "localhost"
          ],
          "port": "8444",
          "path": [
            "wearables",
            "api",
            "collection",
            "dexcom"
          ]
        },
        "description": "This endpoint endpoint initiates the data collection
             process for Dexcom. This structure must be followed if the
            collection needs to be initiated by hand.\n\n--------\n\
            nSuccessful response: \n\n- `200`: No Content.\n\nUnsuccessful
            response:\n\n- `500`: Internal Server error, when the Data
            Source is not found."
      },
      "response": []
    },
    {
      "name": "Authorize new user",
      "request": {
        "method": "GET",
        "header": [],
        "url": {
          "raw": "https://localhost:8444/wearables/api/authorize/{{
              DATASOURCE_ID}}/{{USER_ID}}",
          "protocol": "https",
          "host": [
            "localhost"
          ],
          "port": "8444",
          "path": [
            "wearables",
            "api",
            "authorize",
            "{{DATASOURCE_ID}}",
            "{{USER_ID}}"
          ]
        },
        "description": "This endpoint is used to start the user
            authorization process. After a successful authorization request,
             the user will be redirected to the third-party website to give
            consent.\n\nParameters:\n\n`DATASOURCE_ID`: The unique
            identifier of the Data Source the user should connect to. Values
             can be: `fitbit`, `garmin`, `withings`, `dexcom`.\n\n`USER_ID`:
             Arbitrary String value that will be the unique identifier of
            the user in the system.\n\n--------\n\nSuccessful response: \n\n
            - `302`: Redirection\n\nUnsuccessful response:\n\n- `500`:
            Internal Server error, when the user with the given id is
            already authorized for the given Data Source."
      },
      "response": []
    }
  ]
```

```
193 }
```

Listing C.1: Postman API documentation.

APPENDIX D

Version Control System

Version controlling helps developers tracking and managing changes during development of a software. Version Control System (VCS)s are software tools that helps developers achieve version control over their projects[1]. There are many services available on the market and Git[2] is one of the biggest and most widely adopted[3] open-source VCS.

The project was developed using GitHub[4], which offers repository hosting using Git for free. Even though I was the only developer on the system, version controlling was still useful whenever I made a mistake and I needed to roll back to a stable version. During the development, sensitive information (OAuth secrets) was committed to my personal repository, thus it cannot be made public, because those could leak from the commit logs. After removing them, the software was copied to CACHET's main GitHub repository without any previous commits and is available on the following link:

`https://github.com/cph-cachet/carp.gardener`

[1] https://www.atlassian.com/git/tutorials/what-is-version-control
[2] https://git-scm.com/
[3] https://rhodecode.com/insights/version-control-systems-2016
[4] https://github.com/

APPENDIX E

Deployment Guide

E.1 Deployment

This section describes how the current implementation for CACHET should be deployed into the Web Server. Section 5.4.2 detailed how the *dokcer-compose.yml* file can be used to fire up an instance of the application along with MongoDB and RabbitMQ instance. This guide shows the actual steps to deploy it.

Prerequisites:

The following requirements have to be met before the application can be deployed.

- The server ought to have Docker Engine installed on it. The installation of Docker is explained in their documentation[1].
- Client-id/Client-secret for OAuth2 and Consumer-key/Consumer-secret for OAuth1 devices must be configured in the profile specific configuration files. The following Section E.2 discusses this in detail for each wearable provider.
- The Web Server should open up the "*/wearables/api/**" route for public access. If there is a reverse proxy installed, it should redirect the requests to the port *8444* where the application will be running.

Deployment process:

- Clone the repository[2] to the Web Server if there is Git installed. If not, copy the code-base to the server.
- Navigate to the *docker* folder in the *carp-implementation* package. Execute the command: **docker-compose up**. This command starts the "*docker-compose.yml*" and the application, along with the MongoDB and RabbitMQ instances, will start up using the *production* profile.

If everything is set up correctly, the docker compose command should be able to start the application as intended.

[1] https://docs.docker.com/engine/
[2] https://github.com/cph-cachet/carp.gardener

E.2 Wearable devices setup

This section shows how to set up application on the developer portals for different wearable devices. Dexcom will not be detailed in this section, because only US-based developers can register developers account and support real data collection[3], therefore I was not able to create my own account. Fortunately, they provide "*sandbox data*"[4] to try out their functionalities and imitate "real" collection and authorization.

E.2.1 Fitbit

Developer portal: `https://dev.fitbit.com/apps`
To access this website, one should possess a Fitbit Account.

In the "*Register an app*" tab, a new application can be created. Figure E.1 illustrates the values used in the Fitbit Study.

The significant parts are the *Redirect URL* and subscriber *Endpoint URL* fields. The redirect URL is set to call the callback endpoint of the application, the subscriber URL is set to call the collection endpoint. Figure E.2 illustrates the final properties. The client-id and client-secret values need to be copied to the properties file. In case of Fitbit, there is one more additional value that should be configured, the verification code. This code will be displayed at the bottom of the picture in the *Verification code* section once the application is created. This code must be included in the properties file along with the others, so the application can handle the Fitbit subscription verification flow[5].

E.2.2 Garmin

Garmin Web APIs are not accessible for the public, it is strictly for business use. To access the portals, one should have a verified business account registered.

Developer portal: `https://developerportal.garmin.com/`

On the developer portal a new application can be created. Once it is done, the consumer-key and consumer-secret will be provided for the user. These two properties need to be copied to the configuration file. For subscription handling, an other portal needs to be used.

Application specific portal: `https://apis.garmin.com/tools/login`

[3] https://developer.dexcom.com/content/frequently-asked-questions
[4] https://developer.dexcom.com/sandbox-data
[5] https://dev.fitbit.com/build/reference/web-api/developer-guide/using-subscriptions

The consumer-key and consumer-secret can be used to access the application specific configurations. After logging-in, each data type can be individually customized. For the Technical Study, the configuration of some endpoints can be examined on Figure E.3 and E.4.

The same way as in Fitbit's case, the collection endpoint of the application is configured for each supported data type with the ping system enabled. This way, Garmin will send ping notifications to the specified URIs.

E.2.3 Withings

Developer portal: `https://developer.withings.com/`
To access this portal, one should have a Withings account registered.

After login, the application can be created. Figure E.5 shows how my application was set up for the Technical Study. The important piece here is the *Callback URL*. The application's authorization callback endpoint is configured here.

After the application is set up, the OAuth2 client-id and client-secret is given. The client-secret is called *consumer secret* on their portal. These two properties have to be copied into the configuration file. Figure E.6 displays the final settings.

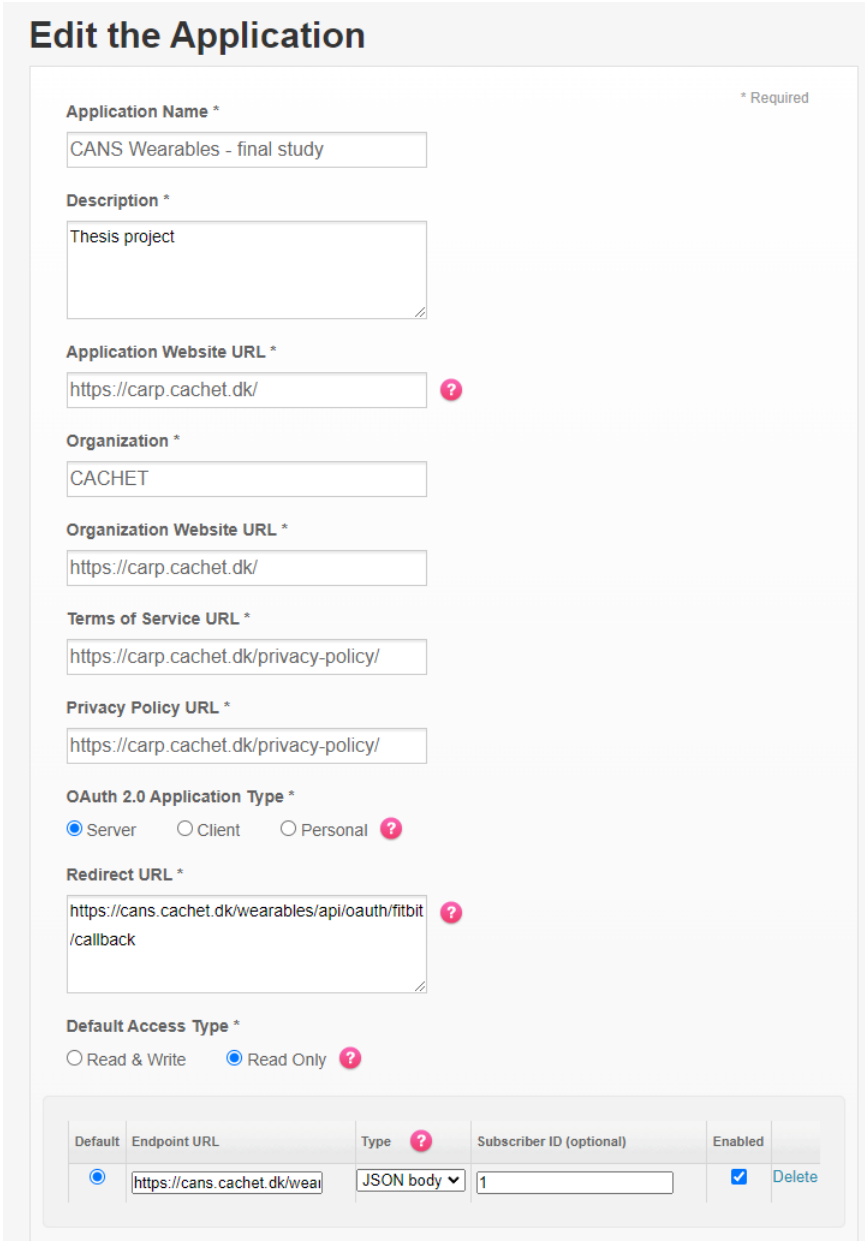

Figure E.1: Fitbit application setup.

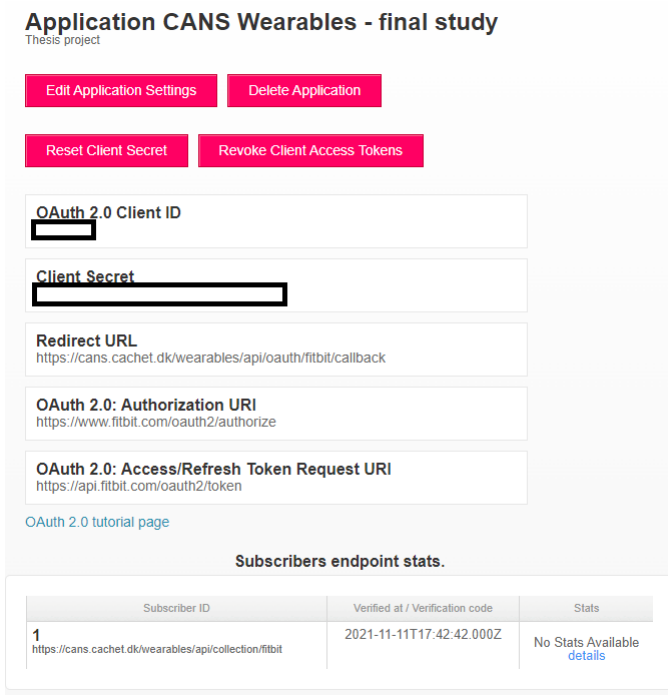

Figure E.2: Fitbit application settings.

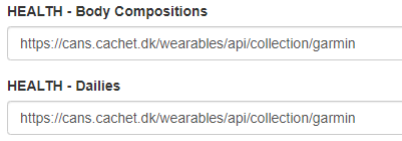

Figure E.3: Garmin callback configuration for data types.

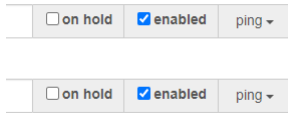

Figure E.4: Garmin ping notification configuration.

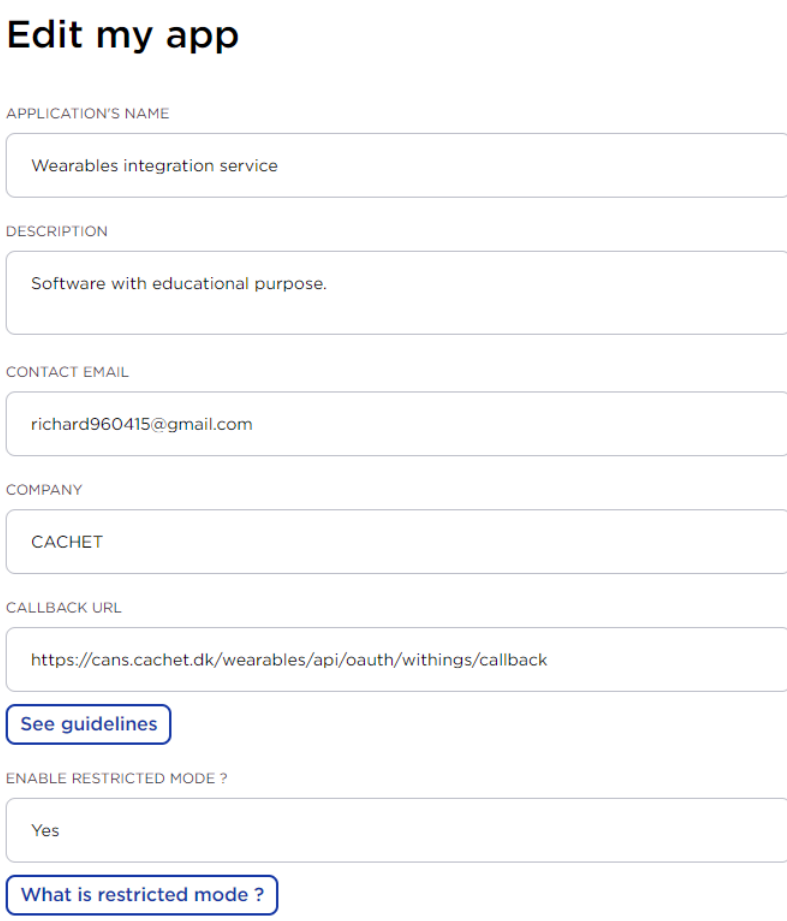

Figure E.5: Withings application configuration.

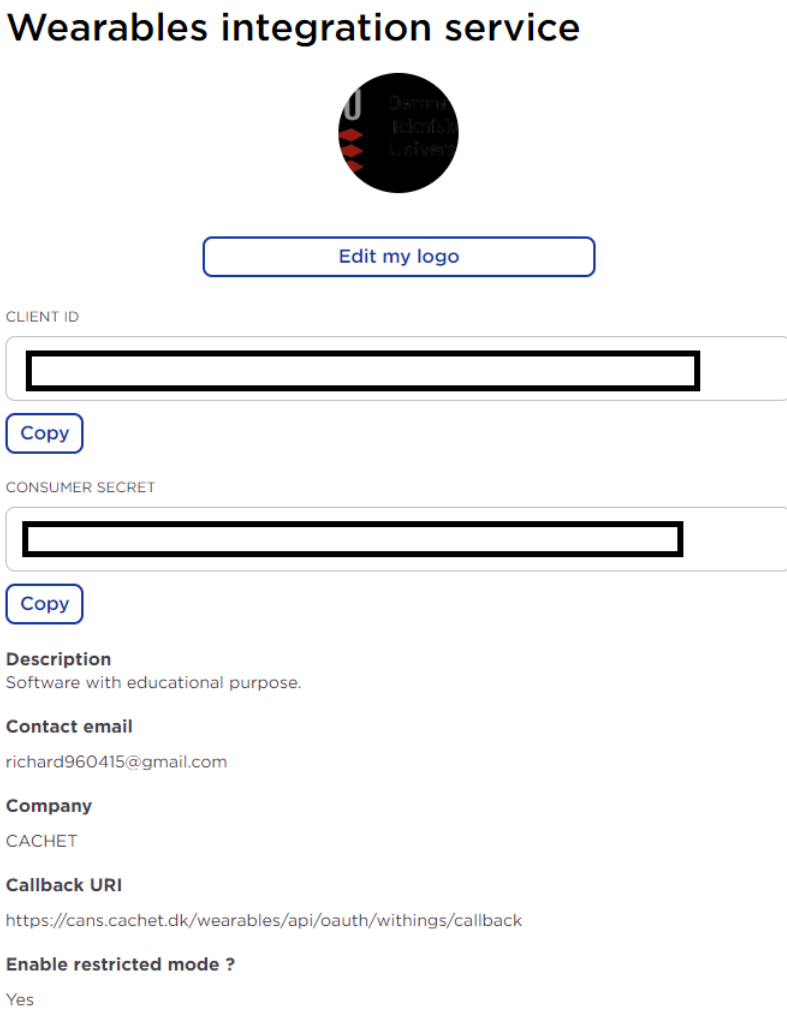

Figure E.6: Withings application properties.

Acronyms

AMQP Advanced Message Queuing Protocol. 25

API Application Programming Interface. i, 3, 4, 6–13, 15, 17–24, 29, 32, 35, 36, 43–45, 47, 49, 52, 62, 65–68, 72, 73, 78, 79, 81, 91, 95, 99, 109, 111, 120

BASE Basically Available, Soft state, Eventual consistency. 27

CACHET Copenhagen Center for Health Technology. i, v, 2–6, 15–18, 24, 26–28, 30, 31, 39, 48, 52, 54–57, 59, 61, 76, 80–84, 87, 88, 90, 91, 93, 94, 96, 97, 99, 111, 117, 119

CANS Carp Nervous System. 5, 52, 54–56, 76, 77, 82, 85, 87–89, 91, 95, 97

CARP Copenhagen Center for Health Technology Research Platform. 16, 30

DTU Danmarks Tekniske Universitet. 99

HTTP HyperText Transfer Protocol. 7, 18–20, 22, 26, 46, 53, 61, 64, 73, 77, 79, 81, 82

HTTPS Hypertext Transfer Protocol Secure. 96

IETF Internet Engineering Task Force. 108

JSON JavaScript Object Notation. 17–23, 30, 44, 55, 62, 71, 73, 77, 86, 91, 93, 111

JVM Java Virtual Machine. 26

MAC Message Authentication Code. 79

mHealth Mobile Health. 2

NoSQL Not Only Structured Query Language. 27, 28, 77, 86

OAuth Open Authorization. 7, 11, 12, 17–23, 30, 32–34, 36, 38, 39, 42–44, 47, 48, 50, 61–66, 71, 72, 77–79, 81, 84, 85, 87, 89, 91, 92, 96, 97, 107–110, 117, 121

OS Operating System. 29

RDBMS Relational Database Management System. 27

REST REpresentational State Transfer. 9, 10, 22

SHA Secure Hash Algorithm. 19, 79

SQL Structured Query Language. 27

UML Unified Modeling Language. 6, 31, 99

URI Uniform Resource Identifier. 17, 18, 22, 35, 37, 41, 44, 50, 54, 65–67, 70, 87, 89, 96, 111, 121

URL Uniform Resource Locator. 17–19, 108, 120, 121

VCS Version Control System. 117

VM Virtual Machine. 29

XML Extensible Markup Language. 20, 22, 28

Bibliography

[1] Tsung-Chien Lu. *Using a smartwatch with real-time feedback improves the delivery of high-quality cardiopulmonary resuscitation by healthcare professionals.* 2019, pages 16–22.

[2] Marco Cipriano. *Recent Advancements on Smartwatches and Smartbands in Healthcare.* 2021, pages 117–127.

[3] Bertalan Mesko. "Health IT and digital health: The future of health technology is diverse." In: *Journal of clinical and translational research* 3.Suppl 3 (2018), page 431.

[4] Bertalan Meskó et al. "Digital health is a cultural transformation of traditional healthcare." In: *Mhealth* 3 (2017).

[5] John Torous et al. "New tools for new research in psychiatry: a scalable and customizable platform to empower data driven smartphone research." In: *JMIR mental health* 3.2 (2016), e16.

[6] Darius A Rohani et al. "Correlations between objective behavioral features collected from mobile and wearable devices and depressive mood symptoms in patients with affective disorders: systematic review." In: *JMIR mHealth and uHealth* 6.8 (2018), e165.

[7] Pegah Hafiz et al. "Wearable Computing Technology for Assessment of Cognitive Functioning of Bipolar Patients and Healthy Controls." In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4.4 (2020), pages 1–22.

[8] Kieran Woodward et al. "Harnessing digital phenotyping to deliver real-time interventional bio-feedback." In: *Adjunct Proceedings of the 2019 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2019 ACM International Symposium on Wearable Computers.* 2019, pages 1206–1209.

[9] Kit Huckvale, Svetha Venkatesh, and Helen Christensen. "Toward clinical digital phenotyping: a timely opportunity to consider purpose, quality, and safety." In: *NPJ digital medicine* 2.1 (2019), pages 1–11.

[10] Salvatore1 Naddeo et al. "A real-time m-health monitoring system: An integrated solution combining the use of several wearable sensors and mobile devices." In: *Healthinf 2017* (2017), pages 545–552.

[11] Frederik Bülthoff and Maria Maleshkova. "RESTful or RESTless - Current State of Today's Top Web APIs." In: *CoRR* abs/1902.10514 (2019). arXiv: 1902.10514. URL: http://arxiv.org/abs/1902.10514.

[12] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. *Investigating Web APIs on the World Wide Web*.

[13] Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. 2000.

[14] Mohamad Kassab et al. "Software architectural patterns in practice: an empirical study." In: *Innovations in Systems and Software Engineering* 14 (December 2018). DOI: 10.1007/s11334-018-0319-4.

[15] Brett Jones et al. "RabbitMQ performance and scalability analysis." In: *project on CS* 4284 (2011).

[16] Daniela Gotseva, Yavor Tomov, and Petko Danov. "Comparative study Java vs Kotlin." In: *2019 27th National Conference with International Participation (TELECOM)*. 2019, pages 86–89. DOI: 10.1109/TELECOM48729.2019.8994896.

[17] Gjorgji Rankovski and Ivan Chorbev. "Improving Scalability of Web Applications by Utilizing Asynchronous I/O." In: *ICT Innovations 2016*. Edited by Georgi Stojanov and Andrea Kulakov. Cham: Springer International Publishing, 2018, pages 211–218. ISBN: 978-3-319-68855-8.

[18] ANA-GABRIELA BABUCEA. *SQL OR NoSQL DATABASES? CRITICAL DIFFERENCES*. 2021.

[19] Shakuntala Gupta Edward and Navin Sabharwal. "Practical Mongodb." In: 2015, pages 13–23. ISBN: 1484206479, 1484206487, 9781484206478, 9781484206485. DOI: 10.1007/978-1-4842-0647-8_2.

[20] Yishan Li and Sathiamoorthy Manoharan. "A performance comparison of SQL and NoSQL databases." In: August 2013, pages 15–19. DOI: 10.1109/PACRIM.2013.6625441.

[21] Matúš Sulír and Jaroslav Porubän. "A Quantitative Study of Java Software Buildability." In: *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*. PLATEAU 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pages 17–25. ISBN: 9781450346382. DOI: 10.1145/3001878.3001882. URL: https://doi.org/10.1145/3001878.3001882.

[22] Christof Ebert et al. "DevOps." In: *Ieee Software* 33.3 (2016), pages 94–100.

[23] Mathijs Jeroen Scheepers. "Virtualization and containerization of application infrastructure: A comparison." In: *21st twente student conference on IT*. Volume 21. 2014.

[24] Marek Moravcik et al. "Comparison of LXC and Docker Technologies." In: *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. IEEE. 2020, pages 481–486.

[25] Erich Gamma. *Design Patterns*. 1994. ISBN: 0201633612.

[26] Barry Leiba. "OAuth web authorization protocol." In: (2012), page 6123701. ISSN: 19410131, 10897801.

[27] E. Hammer-Lahav. "The OAuth 1.0 protocol." In: (2010).

[28] D. Hardt. "The OAuth 2.0 authorization framework." In: (2012).

[29] Christian Huber et al. "A secure token-based communication for authentication and authorization servers." eng. In: *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10018 (2016), pages 237–250. ISSN: 16113349, 03029743. DOI: 10.1007/978-3-319-48057-2_17.

[30] Prabath Siriwardena. "OAuth 1.0." In: *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*. Berkeley, CA: Apress, 2014, pages 75–90. ISBN: 978-1-4302-6817-8. DOI: 10.1007/978-1-4302-6817-8_6. URL: https://doi.org/10.1007/978-1-4302-6817-8_6.

[31] Eran Hammer-Lahav. *The OAuth 1.0 Protocol*. RFC 5849. 2010. DOI: 10.17487/RFC5849. URL: https://rfc-editor.org/rfc/rfc5849.txt.

[32] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. October 2012. DOI: 10.17487/RFC6749. URL: https://rfc-editor.org/rfc/rfc6749.txt.

[33] Eugene Ferry, John Raw, and Kevin Curran. "Security evaluation of the OAuth 2.0 framework." In: *Healthinf 2017* (2015), pages 73–101. ISSN: 2056497x, 20564961.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ