

UiO : **Department of Informatics**
University of Oslo

Collecting activity data using the Open mHealth platform

An exploratory study on integrating objective data with
sport monitoring systems

Daniel Gynnild-Johnsen , Lars-Erik Holte
Master's Thesis Spring 2017



[**simula** . research laboratory]

Collecting activity data using the Open mHealth platform

Daniel Gynnild-Johnsen Lars-Erik Holte

May 2, 2017

Abstract

Football players work together as a unit to perform on an elite competitive level, and the most minor abnormalities can determine the outcome of a match. Success can often be the result of healthy, uninjured and rejuvenated players working together as a collective. Even though it is impossible to control all outcomes and scenarios, the risk of failure might be minimized by monitoring players closely on an individual level. If we monitor players over a longer period of time we might discover patterns or abnormalities in their training. This information can be used to avoid multiple scenarios related to fatigue, injuries and overtraining.

In this thesis we present a proof of concept for expanding an existing self-reporting monitoring system called pmSys, and look at how football teams and players can utilize modern technology like phones and wearable devices to capture objective data. This system will collect and store the data, which can be processed into useful visualised feedback, and help a team to evaluate their players. This way the coaches can make mitigating measures to improve certain aspects that might be lacking on a player or team level. By eliminating the use of pen and paper, pmSys introduces a simpler way of reporting the players' health status. By expanding the system with objective data, the team performance can effortlessly be evaluated and adjustments made if needed. Compared to the subjective data, which are a player's assessment of themselves and their health, objective data is not as simple to tamper with and rarely lies, and can more accurately determine a player's performance state. We believe this monitoring system can help giving a team the advantage to succeed.

Contents

1	Introduction	1
1.1	Background	1
1.2	Problem definition	2
1.3	Limitations	2
1.4	Research methods	3
1.5	Main contributions	3
1.6	Outline	4
2	Related Work	6
2.1	GPS	6
2.2	Wearables	7
2.2.1	Accelerometer	8
2.2.2	Gyroscope	8
2.2.3	Altimeter	8
2.3	Mobile health	8
2.4	Ohmage	9
2.5	Open mHealth	10
2.6	Docker	11
2.6.1	Composing a docker container	12
2.7	MongoDB	13
2.8	OAuth 2	14
2.9	Third-party data sources	16
2.10	Athlete Monitoring	18
2.11	Summary	19
3	Shimmer	20
3.1	Shimmer components	20
3.1.1	Shims	21
3.1.2	Resource server	21
3.1.3	Console	21
3.2	Schema	23
3.2.1	Design principles	24
3.3	Grant Type: Authorization Code	26
3.4	Installation	28
3.5	GPS in Shimmer	28
3.6	Summary	30

4	Runkeeper and Health Graph	32
4.1	Runkeeper	32
4.1.1	Runkeeper application	32
4.1.2	Wearable hardware	33
4.1.3	GPS accuracy	33
4.2	Health Graph	34
4.2.1	API	34
4.2.2	Health Graph console	35
4.2.3	Limitations	36
4.3	Summary	36
5	PmSys	38
5.1	Current pmSys	38
5.1.1	PmSys Mobile Application	38
5.1.2	PmSys-trainer	39
5.2	PmSys with objective data	40
5.2.1	Motivation	40
5.2.2	Injecting shimmer into the backend	40
5.3	Summary	41
6	Testing	42
6.1	Using Shimmer	42
6.2	Using bash script	43
6.3	Summary	44
7	Proof of concept	46
7.1	System requirements	46
7.1.1	Functional requirements	46
7.1.2	User stories	47
7.1.3	Non-functional requirements	47
7.2	Working with Shimmer	49
7.2.1	GPS schema	49
7.2.2	GPS data point mapper	52
7.3	Connect pmSys user to Runkeeper	53
7.4	Fetch data from Runkeeper	58
7.4.1	Scheduling requests	64
7.5	Database storage	66
7.5.1	Replication	66
7.5.2	Concurrency in MongoDB	68
7.5.3	Big data	71
7.5.4	Database optimization	74
7.6	Summary	75
8	Conclusion	77
8.1	Summary	77
8.2	Main Contributions	77
8.3	Future work	78
8.3.1	PmSys frontend	79

8.3.2	PmSys backend	79
8.3.3	Aggregation of data	81
Appendix A	Accessing the source code	85

List of Figures

2.1	Wearable forecast	7
2.2	Ohmage system architecture	9
2.3	Open mHealth architecture	11
2.4	Difference between Docker and VM	12
2.5	MongoDB data collection	14
2.6	JSON team sample	14
2.7	OAuth 2.0 Protocol flow	15
2.8	Worldwide smartphone OS market share	18
3.1	Shim workflow architecture	20
3.2	Adding client id and secret in the console	22
3.3	List of added APIs in the console	23
3.4	JSON schema sample	26
3.5	OAuth access token example	27
3.6	Grant type: Authorization code flow	28
3.7	JSON raw data	29
3.8	JSON shimmed data	30
4.1	Actions during activity recording	33
4.2	JSON team sample	35
4.3	Open Health developer console	36
5.1	RPE scale	38
5.2	PmSys-app flow	39
5.3	PmSys backend	39
5.4	Isolated pmSys backend with shimmer	41
7.1	Calories burned sample data	50
7.2	Shimmed GPS data from Runkeeper	52
7.3	Application registration in Runkeeper	54
7.4	Logical model of the authorization table	54
7.5	Service data collection	55
7.6	Service authorization data collection	55
7.7	Sequence diagram: Connect to Runkeeper	56
7.8	Connect mockup	57
7.9	Logical model of the activity table	58
7.10	Activity data collection	59
7.11	Sequence diagram: Fetch daily data	64
7.12	Modified JSON response	67

7.13 Database deadlock	69
7.14 Concurrency flowchart	70
7.15 Horizontal database sharding	71
7.16 Big data	72
7.17 Distributed database	73

List of Tables

2.1	Currently supported shims and requestable measures	16
2.2	Supported APIs and requirements	17
3.1	Examples of Unified Code for Units of Measure	25
6.1	Shimmer non-detailed test results	43
6.2	Shimmer detailed test results	43
6.3	Bash script non-detailed test results	44
6.4	Bash script detailed test results	44
7.1	User stories	47
7.2	Crontab entry parameters	65
7.3	Crontab keywords	66

Acknowledgements

We would like to thank Pål Halvorsen for your great help, and keeping the motivation up. You have provided us with great insights, ideas and much constructive feedback throughout our thesis. Your help has truly been an invaluable resource and encouragement for us.

Furthermore we would thank Håvard Johansen for your technical knowledge regarding PmSys and the underlying structure. Your technical courses and information has contributed much to our understanding of how the system as a whole correlates. And we would like to thank you both for letting us contribute to this project.

I, Daniel Gynnild-Johnsen, would like to thank Lars-Erik Holte for his commitment and contribution to this thesis. You have helped to keep my motivation up, and been a great support when brainstorming theory and strategic choices for the direction of our collaboration.

And I, Lars-Erik Holte, want to thank my fellow author Daniel Gynnild-Johnsen for a rewarding collaboration. I must express my profound gratitude for his commitment and being the best sparring partner one could ask for.

Oslo, May 2nd 2017

Daniel Gynnild-Johnsen, Lars-Erik Holte

Chapter 1

Introduction

1.1 Background

Football is a team sport with many individuals. A general senior squad consists of between 20-30 players. This makes it close to impossible for a coaching team to closely observe and supervise each and every player throughout an entire training session. A football match includes a total number of 22 players, with some players operating in the same general area. This creates the possibility of one player working harder than the other. Football has been criticised the past few years for not being a top-class sport with top athletes. A cross country skiing athlete, Finn Hågen Krogh, went as far as saying that with a couple of years of hard work, he could easily play in the norwegian top division, Eliteserien (formerly known as Tippeligaen) [1].

The Player Monitoring System(pmSys) is a tool used by several teams in Eliteserien, the Danish superliga and the norwegian national teams to monitor the players well-being and work load, as well as reporting injuries. The data collected by pmSys is based on subjective data collected through questionnaires made in collaboration with a Norwegian School of Sports Science (NiH) PhD study [2]. The players fills out these questionnaires using a mobile application, making the data subjective and the process semi-manual. Prior to pmSys, and still the reality for many football teams, the players had to go through a tedious and time consuming process of filling out forms using pen and paper. PmSys introduced a digitized version of this daily routine, making it easier for the players to report, and the staff to analyze and aggregate on this subjective data to create improved reports.

The general idea is to be able to maximize the workload without it leading to injuries and overtraining. With knowledge of the players general well-being after a training session, the staff can make decisions and appropriate adjustments based on the data collected. How much work a player has put in during a training session is individual and will vary, and by implementing a way to collect objective data of a player's workload during each training, the staff will get a bigger picture of each individual player's

fitness and fatigue, and the team as a unit. It will create the possibilities of analyzing how far and where a player has run during training, acceleration, deceleration and other useful information.

1.2 Problem definition

When pmSys was created, the system introduced a digitized solution for capturing football players perception towards their physical well-being, work load and injury status. In this thesis we will look to further improve pmSys, with focus on *how to integrate objective activity data from third-party providers, hardware and physical activity tracking wearables*. The main goal is to supplement pmSys with human readable data, which will give the staff a more detailed view and deeper understanding of a player's general fitness and potential changes in fitness over time. This will culminate in the staff being able to adjust their training regime based on the squads general fitness, and make specific changes in training on an individual level. Furthermore, we investigate the most efficient ways of gathering, storing and processing this data, to maintain and enhance the application's usability.

This new functionality should influence the current use of the pmSys as little as possible, making the transition require minimal effort from the end users. The data collecting process will require the players to utilize tracking hardware and wearables, which will ultimately provide valuable individual data from training sessions. The introduced functionality should also shield the system administrators from performing unnecessary assignments as much as possible.

1.3 Limitations

This thesis is a research based on an existing system with a relatively large number of users. To supplement this system with extended functionality, we had to take the current system architecture, flowchart and components into consideration. During this thesis we have gathered data and researched third-party APIs, and we discovered that they all had their limitations and restrictions.

Our collaborator in pmSys is located in Tromsø, which lead to fewer physical meetings and more frequent mail correspondence. This has not made the research process problematic, but might indirectly slowed down the actual developing progress.

1.4 Research methods

The research conducted in this thesis can be divided into these following phases:

- Theoretical study of Open mHealth and their Shimmer application
- Theoretical study of pmSys' components and infrastructure
- Analysis of the APIs supported by Shimmer
- Development of the Shimmer expansion with Open mHealth schema and shims
- Performance analysis of Shimmer compared to other alternatives
- Development of a proof of concept implementing Shimmer in pmSys

To be able to gain an understanding of the topics introduced in this thesis, a theoretical study and secondary research of relevant material was needed. The majority of the information was gathered from technical specifications and documentation from resources online produced by the developers of the different systems. In addition to this, knowledge about security principles and mechanisms, as well as different database technologies was obtained from textbooks and other learning resources.

The analysis conducted on the supported APIs consisted of reading and understanding the documentation and technical specifications from the data providers. This analysis was crucial to be able to identify which APIs were relevant with regards to fulfilling the requirements. It is worth mentioning that since the start of this thesis, Shimmer has worked on implementing support for additional APIs, with four integrations in the pipeline.

The developing phase of expanding the Shimmer application was conducted with a Scrum software development methodology. The sprints were rather short, with a manageable sprint backlog. To be able to measure the Shimmer application's performance, we decided to alternative solutions, which produce the same results as Shimmer. The different solutions were tested in different environments to discover potential deviations.

In the final phase of the thesis, after expanding Shimmer enough to serve its purpose, we went on to producing a proof of concept. The development of this proof of concept went through several iterations, and the concept was improved and changed when new, seemingly superior, possibilities were discovered during the process.

1.5 Main contributions

In this thesis our main contribution is preparatory work through developing a proof of concept. We argue how supplementing objective data to

pmSys through Shimmer, an application made by Open mHealth, can contribute to capture valuable and constructive data for assessing with precision a person's physical health status. By utilizing data fetched through third-party providers, one can draw conclusions and predict future outcomes from exercise, and even adapt an exercise regime to match the current potential a person has. This data is returned to users through the Health Graph API, and through the API the ability for creating collections of detailed data and make a comparison to other user's is presented. The thesis also cover some approaches to how combinations of the API and some logic can store data and be used in context to be presented to the end user.

The second contribution is to give an overview of how the different systems, which constructs the whole architecture, can be fully utilized for optimal performance and availability. We will describe all the different components, and how they can work together to create an environment where objective data from third-party providers can be fetched, processed and stored to complement the existing captured data set. We will present how Shimmer can be used to normalize the data and give it the same contextual meaning in the same format.

Our third contribution is providing test results for solutions which can be used to supply pmSys with the relevant objective user data. This is for outlining the pros and cons with totally different solutions that ultimately achieves the same, and provides the same end results. This includes speed and complexity of implementation.

1.6 Outline

Chapter 2 presents the most important related work and research on topics that helped us gain an understanding of pmSys, and has had an impact on further development of this thesis as a whole.

We then analyse and evaluate the main components of Shimmer and how they work together, and how we can utilize the application and its features in chapter 3. We also look at how OMH is using Shimmer to convey their purpose regarding shared common data formats.

Chapter 4 covers the chosen data provider Runkeeper, how the application works and how wearable technology can be used together with Runkeeper to capture objective data points. We also look at and exemplify the usage of its Health Graph API. The limitations concerning the system and API is also discussed here.

In chapter 5 we introduce the different parts of the current pmSys, the mobile application, the trainer web portal and the backend infrastructure. Here we also discuss how the objective data can help improve the system,

and how it will affect the infrastructure.

Then, in chapter 6 we introduce the testing that has been done with regards to the fetching of data from Runkeeper, comparing Shimmer to a provisional solution, looking at different possibilities of gathering the data and measure the performance of the two.

Based upon the former chapters, chapter 7 presents the proof of concept and ties it up to our problem definition, on how the implementation of objective GPS data in pmSys can be solved.

Lastly, in chapter 8 we conclude this thesis by giving a summary of our discoveries, presenting the limitations of our research and giving recommendations for further work and research.

Chapter 2

Related Work

This chapter will present related works concerning Ohmage and its architecture and components, the concepts of eHealth and mHealth, the Open mHealth organization's purpose and how restructuring and normalizing medical data can help increase its usefulness. We will also look at the global positioning system (GPS), discuss the relevant third-party APIs for gathering GPS data, and other useful technologies related to this research thesis.

2.1 GPS

The global positioning system (GPS) is a navigation system based on satellite. The satellites were originally put into orbit for military purposes. Yet today anyone can use this system 24 hours a day, anywhere in the world in any weather conditions. In short, the user has a GPS receiver which measures the distance to each satellite by the duration it takes before getting a signal from the satellites, and can then compute your coordinates [3].

Coordinates in the coordinate system consists of longitude, latitude and altitude to calculate a three-dimensional location. Longitude specifies the the east-west position, latitude represents the north-south position, with altitude showing the height above sea-level. The longitude and latitude are essential, and without knowing the altitude we can still calculate a two-dimensional location of a specific point on the planet. A football pitch is flat, so the movement we look to capture and analyze is essentially the horizontal movement of the players. Despite of this, the altitude can provide useful analytical data, due to the different locations and distances between football stadiums in the world. In Latin America, there are many high altitude nations, especially those in the Andes. As an illustration, Estadio Hernando Siles is a sports stadium in La Paz, Bolivia. It is located in the Miraflores borough of la Paz, with an altitude of 3 637 meters above sea level. This makes it one of the professional stadiums in the world with the highest altitude. The International Federation of Association Football (FIFA) issued in 2007 an altitude limit, raising the limit from 2 500 meters to a maximum of 3 000 meters. This ban was in 2008 suspended after

some controversy. The bolivian national team, having Estadio Hernando Siles as their home field, is known for their strong performances at home. They have beat former world champions Brazil and Argentina numerous of times, and they won the Copa America in 1963 as the host [4]. It is well documented that at high altitudes, the air pressure lowers and fewer oxygen molecules are present in the air, raising both heart rate and the number of breaths per minute (respiratory rate) to help pump oxygen through the body [5]. Raised respiratory rate, low humidity and dry air are all factors at high altitude that can lead to dehydration. A research article done by Brosnan et al. in 2000 [6] examined repetitive cycling sprints at an altitude difference of approximately 1 500 meters, getting results that shows the altitude reduces sprint performance by between 5 and 10%, and that shorter rest amplifies this effect. Football players, playing a very interval oriented sport, will notice this effect on the body, making the altitude data interesting and important for training purposes.

2.2 Wearables

"In fitness trackers, brands such as Fitbit and Xiaomi are leading the charge. CCS Insight expects that 53 million fitness trackers will be sold in 2016, with volumes reaching 165 million in 2020, at a total value of \$5 billion." [7]

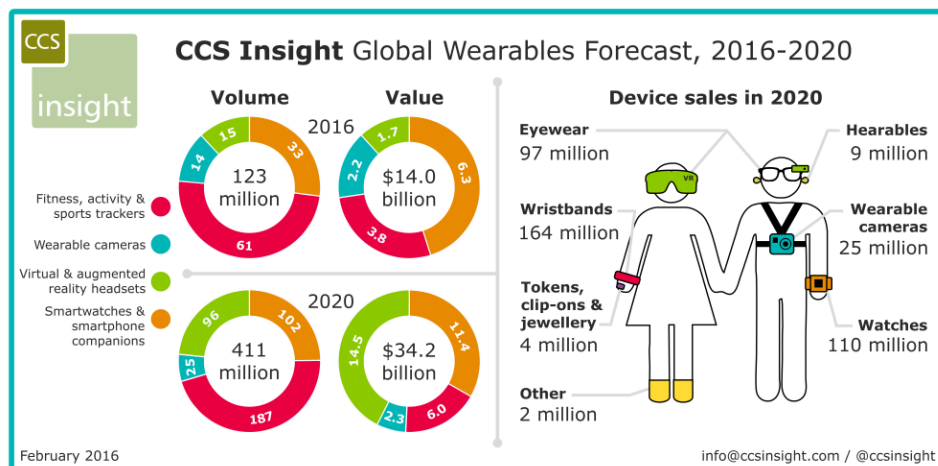


Figure 2.1: Wearables forecast for unit sales and value 2016-2020. Figure src: [8]

Wearables are hardware that a person can use for multiple things. In this context we are referring to wearables used with the purpose of tracking spatiotemporal activities. This hardware often comes in the form of a bracelet or a watch, and quietly monitors different aspect of your health and motion. These wearables are often packed with different technologies which differentiate themselves from each other. Some has eliminated the need for bringing along a smartphone by introducing offline storage, and is synced automatically when the smartphone is back in range. We will

cover the most basic and important technologies which must or should be included in the wearable for the sake of analysing useful data in pmSys.

2.2.1 Accelerometer

In smart devices an accelerometer is a piece of electromechanical technology used so the device can capture data based on orientation and velocity. Common accelerometers utilize the piezoelectric effect or capacitance sensors to deduce these datas. The piezoelectric effect uses tiny crystals, which when applied force becomes stressed. This stress releases small amounts of voltage which the accelerometer interprets and uses to calculate velocity and orientation. Another way of calculating this data is using capacitance sensors, which reads changes in capacitance. capacitance is stored voltage, and the sensors will detect changes in the capacitance when force is applied and translate it into data. Smart devices commonly has 3-axis accelerometers to be able to detect orientation and velocity on a xyz axis (3D).

2.2.2 Gyroscope

Compared to an accelerometer a gyroscope helps determine orientation through gravity. A freely rotating disk, commonly called a rotor, is mounted on a spinning axis to indicate which way is down on a platform to determine gravitational pull. Unlike the accelerometer, the gyroscope can detect changes in orientation on all axes without being applied any force to.

2.2.3 Altimeter

An altimeter is technology devised to detect altitude. The detecting is usually performed through reading changes in atmospheric pressure. It can often be more accurate than GPS to detect altitude as the GPS signal can become unavailable in obscure places. Take into consideration the sports stadium in La Paz mentioned in the GPS chapter 2.1, we could measure performance at high and low altitude with data provided by an altimeter.

2.3 Mobile health

eHealth is a term referring to health care using electronic processing and communication. Mobile health (mHealth) is one of the components of eHealth. The Global Observatory for eHealth has defined mHealth as public medical healthcare supported by mobile phones and other wireless devices. The idea of mHealth is to utilize the core utilities and complex functionality of these devices like 3G and 4G, Bluetooth and GPS among others [9].

Furthermore, mHealth makes it possible to share and receive health data at any moment in real-time, despite geographical distances. This will make it possible to discover and treat illnesses in a shorter timespan, which is

beneficial.

The mHealth concept has a challenge with regards to the amount of closed applications in separate silos with its own separate data format and management and analysis tools. These types of systems are known as stovepipe systems (see figure 2.3). This type of architecture lacks coordination and planning across different systems, prohibiting the mHealth concept of realizing its full potential and value.

2.4 Ohmage

Ohmage is an open end-to-end participatory sensing (PS) platform. PS is a way of approaching distributed data collection and analysis that takes advantage of smartphones. The Ohmage platform is used for gathering data in two separate ways. Either through self-reporting mobile apps by letting participants answer surveys and gathering the survey responses, or by using passive data collection apps and letting the application collect continuous data streams automatically (see figure 2.2).

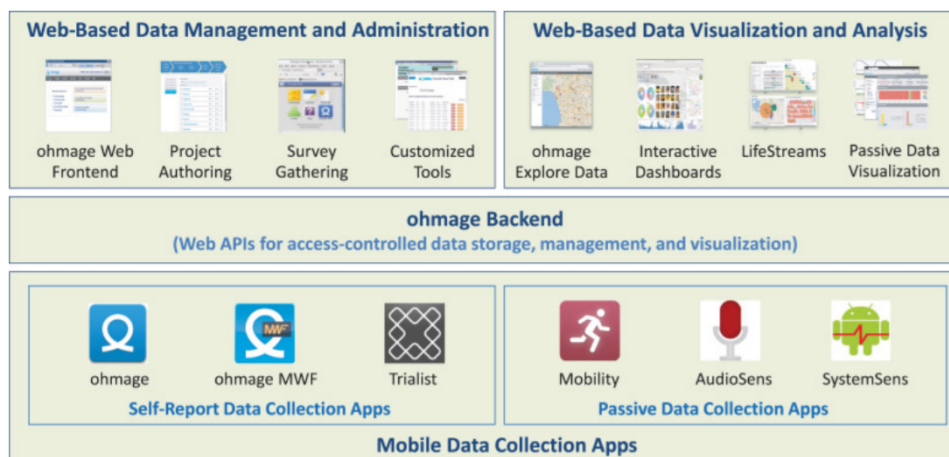


Figure 2.2: Ohmage system architecture[10]

There are four components which together creates the construct that is Ohmage[10]:

Ohmage backend

The first, and most central component, is the Ohmage backend. This component is a datastore that provides a unified interface for data access. It handles functions related to secure communication, authentication and data storage among others. This handling is done through an extensive set of backward compatible Ohmage web APIs.

Mobile data collection apps

The second main component is the data collection component. This is divided into two groups; self-reporting applications and passive data collection applications. All of the applications are used on the participants' mobile devices to collect data. Today's pmSys collects data through a self-reporting application.

Web-based data management and administration

The third is the web-based management and administration tool for the collected data. This web frontend is the main management portal for projects (surveys), data and users. It can be used to monitor incoming data, create or edit projects as well as managing access restrictions.

Web-Based data visualization and analysis

The final component is the web-based data analysis and visualization tool for reading, visualizing and analyzing the captured data. The whole point of this system is to make sense of the captured data and visualize the results and make it actionable to the end user. This component provides different visualization tools which dynamically retrieves the data from the Ohmage backend.

Ohmage is a product of many participatory sensing systems combined together to form a generic platform with the possibility of customization to fit different scenarios and purposes. It lets you create surveys and questionnaires which participants can answer on their devices, with or without internet connection. In addition to collecting data through mobile applications, Ohmage provides a web application for administering the data collected. In the web interface, the data can be visualized and analyzed at any given time with different tools that Ohmage provides. Ohmage also lets you easily export the data to use other analytical tools [11]. The platform provides unified data access across the applications built upon the Ohmage backend, with the pmSys-app and Shimmer being two examples of such applications.

2.5 Open mHealth

The open mHealth (OMH) organization, founded in 2011, describes itself as "a nonprofit start-up breaking down the barriers to integration and bringing clinical meaning to digital health data." [12] OMH works with clinical experts and system developers with the purpose of making digital health related data as useful and actionable as possible, and has built an open source system called ohmage-omh. This system is based on the Ohmage platform (see section 2.4), and is intended for rapid health data gathering through mobile applications. The platform also includes a data storage unit (DSU) which securely stores the gathered health data from

the users. The system is designed to be able to collect health data from separate data sources, and the whole idea behind the OMH initiative is to standardize these data sets to shared data standards. Every device manufacturer or system has its own idiosyncratic way of structuring data in its own silo, and may provide widely different presentation of the exact same data. Shimmer is an application developed by OMH, and is meant to serve as an application that gathers and normalizes data from different sources. By normalizing the data sets to a shared standard, the data sets will provide the same context and can be shared across other systems more easily.

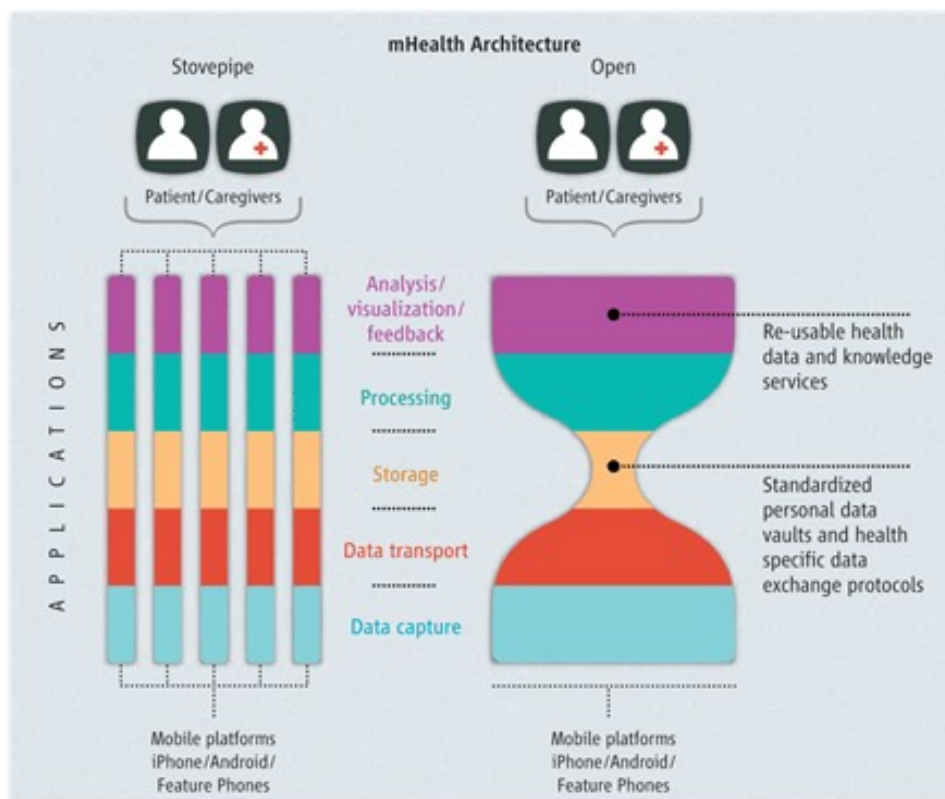


Figure 2.3: mHealth stovepipe architecture (left) and open mHealth’s open architecture (right) [13].

OMH has created a set of standardized frameworks with optimized data schemas for clinical usage. OMH is building a global community of developers, health IT staff and researchers. This community is together building and maintaining an open framework for digital health data usage available to all.

2.6 Docker

Docker is a software container platform, which allows applications to be deployed inside software containers. Docker packages an application and

all of its dependencies to avoid problems with compatibility and making it more portable. Everything required to run a software is packaged into isolated images. Packaging an application this way will guarantee that the software always will run as expected regardless of the environment to which it is deployed as long as the configuration is correct. A software developer can create a portable application which can be run anywhere the Docker Engine has been installed. This saves a lot of work both for the developers and system administrators as it no longer becomes necessary to support different platforms and operating systems. Amazon, Google and Microsoft all added support for Docker to their platforms, and are continuously contributing to the project.

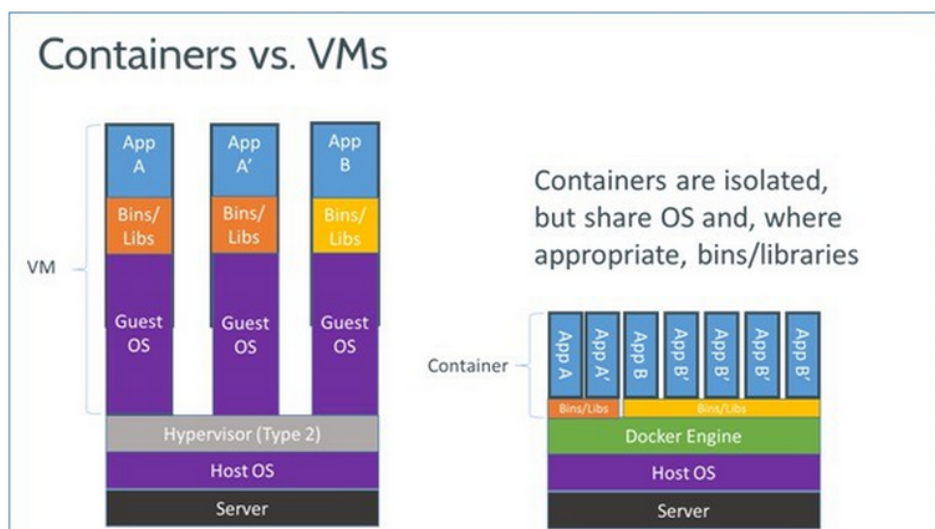


Figure 2.4: Illustration of the difference between the Docker and VM (virtual machine) environments. VMs run their own guest operating systems, whereas the Docker environment runs on the same operating system as its host allowing it to share a lot of the host operating system resources.

Shimmer(see chapter 3) can be deployed using the Docker platform, and consists of MongoDB, nginx and OpenJDK base images. You will need Docker and some components found in the Docker Toolbox[14], like docker compose and docker machine. You also need to have a running Docker machine, either locally or in cloud platforms like Amazon Web Services, Microsoft Azure or Digital Ocean.

2.6.1 Composing a docker container

Docker compose is a tool for packaging your own application into either a single container or multi-structured containers. To create a container, a docker-compose.yml file must be defined. From this file, installation and run features can be specified to combine everything that a user would need to use your application. This eliminates the need to manually

download required software from multiple third-parties, and ensures that the installation process is completed without causing problems for the users. Compose also functions as an excellent tool for staging new releases for software so it can be tested in a clean and isolated environment to discover possible bugs and limitations. The following example illustrates how a basic definition of a web app with a MongoDB dependency will use the `docker-compose.yml` file to download it and resolve the dependency:

```
version: '1'
services:
  web:
    build: ./dir
    ports:
      - "5000:5000"
    volumes:
      - ./code
    links:
      - mongo:mongo
  mongo:
    image: mongo
```

2.7 MongoDB

"MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling." [15] MongoDB is a NoSQL database variant. NoSQL's traits are partition tolerance, speed and availability at the cost of consistency. NoSQL rather operates under the idea of "eventual consistency", where data is eventually propagated to all nodes which expect the incoming change. NoSQL dates back to the 1960's, but haven't seen much use before companies like Facebook, Google and Amazon started storing massive amounts of data. These bulks of continuous input did not match the availability and characteristic of a tabular database which has complex logic constraints, but rather the simplicity of easy storage and retrieval of data without immediate logical constructs.

Entities in MongoDB is stored in documents, and consists of keys and corresponding values much like JSON data structure which is represented in figure 2.6. The document which represent an entity holds a single record per document, so for each create operation a new document is created. These documents are then mapped into a collections, which holds all the documents representing the same entities. An entity in the database world is any object that we wish to model, concrete or abstract. These entities are often recognizable concepts such as a person, an item or an activity. These entities are usually referencing each other through shared identifiers based on keys. Identifiers can consist of one or many keys with the sole purpose

of making the entities unique.

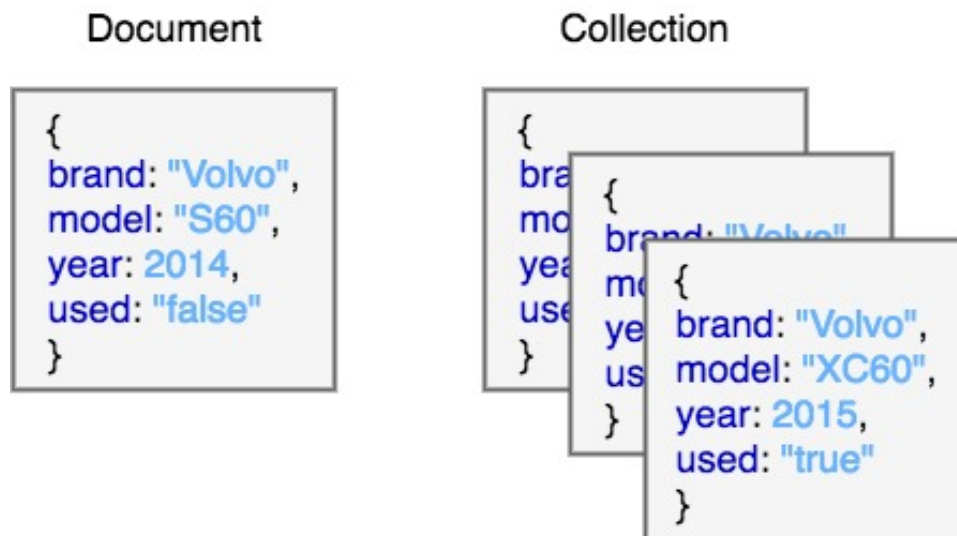


Figure 2.5: A car entity and multiple entities in a collection with different values but same keys. Figure inspiraiton: [16]

MongoDB also offers an option to make embedded data, storing related data in arrays that would normally be referenced using identifiers in a single document. The values corresponding to the keys are called BSON, which are a binary serialization format used for create, read, update, and delete operations(CRUD). MongoDB is highly adaptable as BSON supports many native programming languages, and exceeds at receiving JSON data as a direct input, skipping much of the process of heavy data manipulation as it is the natural data for MongoDB. The new objective data which pmSys will be receiving is pure JSON, and given that pmSys already has a MongoDB storage, all that is needed for the data is a new collection or database without immediate concern of the relations.

```
{
  name: {first: "Tom", last: "Tomson"},
  birth: new Date('May 17 1989'),
  employed: "True"
}
```

Figure 2.6: An example of a document with multiple data types for a person entity.

2.8 OAuth 2

The shims support authentication both through OAuth1.0 or 2.0 to authenticate the application and its users, since the third-party APIs use different authentication mechanisms. "The OAuth 2.0 authorization

framework enables a third-party application to obtain limited access to an HTTP service." [17] OAuth 2 provides the authorization flow, in our case, for a third-party application and delegates access tokens that the application can fetch and store data for user accounts. The framework consists of four primary roles: resource owner, resource server, authorization server and client.

Resource owner (User)

This is the owner of data/resources that third-party applications would like to access. The owner is the cog in the framework which must grant access, and choose the scope of access for applications, in general this is read-and/or-write access.

Resource and authorization server (API)

The API accomplishes both resource and authorization roles. A developer can access resources through use of API calls as long as the application has been validated by an API using OAuth2. The validation is achieved through verification of the user, and granting access tokens to the application.

Client (Application)

The application which tries to access the resources must be validated both by the resource and the user to be able to use the API.

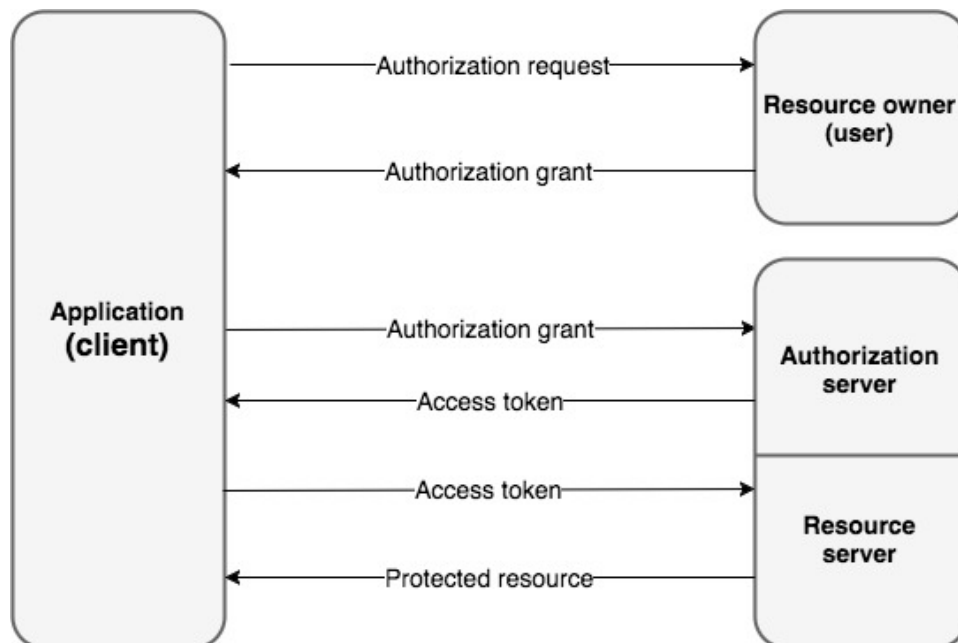


Figure 2.7: General OAuth 2.0 protocol flow interaction. Figure inspiration src: [18]

The protocol flow explained in figure 2.7 is a general flow, so the actual flow in Shimmer will differ slightly based upon what authorization grant type is needed.

2.9 Third-party data sources

When starting the research on integrating pmSys with OMH’s shims, it was important to cover what third party sources already had a collection of shims implemented in the API. Then the focus could be directed to those who could offer the correct data and features that was needed by pmSys. Already integrated in Shimmer was Fitbit, Google Fit, iHealth, Jawbone, Misfit, Runkeeper and Withings [19], with a fixed set of supported endpoints, and every APIs with their own limitations.

Shim	Measures
Fitbit	Activity, Step_count, Body_weight, Body_mass_index, Sleep_duration
Google Fit	Activity, Body_height, Body_weight, Heart_rate, Step_count, Calories_burned
Jawbone	Activity, Body_weight, Body_mass_index, Step_count, Sleep_duration, Heart_rate
Misfit	Activity, Step_count, Sleep_duration
Runkeeper	Activity, Calories_burned
Withings	Blood_pressure, Body_height, Body_weight, Heart_rate, Step_count, Calories_burned, Sleep_duration
iHealth	Activity, Blood_glucose, Blood_pressure, Body_weight, Body_mass_index, Heart_rate, Step_count, Sleep_duration

Table 2.1: Currently supported shims and requestable measures

The table 2.1 shows the data provider API library provided by Shimmer. It shows the shims/providers, as well as the measures that Shimmer makes available.

API \ Requirement	GPS data	Android	iOS	Wearables
Fitbit [20]	X	✓	✓	✓
Google Fit [21]	✓	✓	X	✓
iHealth [22]	X	✓	✓	✓
Jawbone [23]	X	✓	✓	✓
Misfit [24]	X	✓	✓	✓
Runkeeper [25]	✓	✓	✓	✓
Withings [26]	X	✓	✓	✓

Table 2.2: List of APIs supported in Shimmer.

In table 2.2, the Shimmer supported APIs are compared up against the requirements they needed to fulfill to be integrated in pmSys. The requirement to the GPS data is that the API needs to provide a continuous stream of geo locations during a workout. A few of the APIs provide recorded locations, but only the location of when the workout is ended and logged, which is insufficient.

The only third-party capable of logging sufficient GPS data on a cross platform basis, and has an API supporting extraction of that data is Runkeeper (see section 4). Finding an API that has cross platform support is important to cover all of pmSys' users. The PmSys application is directed at Android and iOS users, as these smartphone operation systems together, per third quarter 2016 cover above 99% of the market share worldwide (see figure 2.8).

Worldwide Smartphone OS Market Share (Share in Unit Shipments)

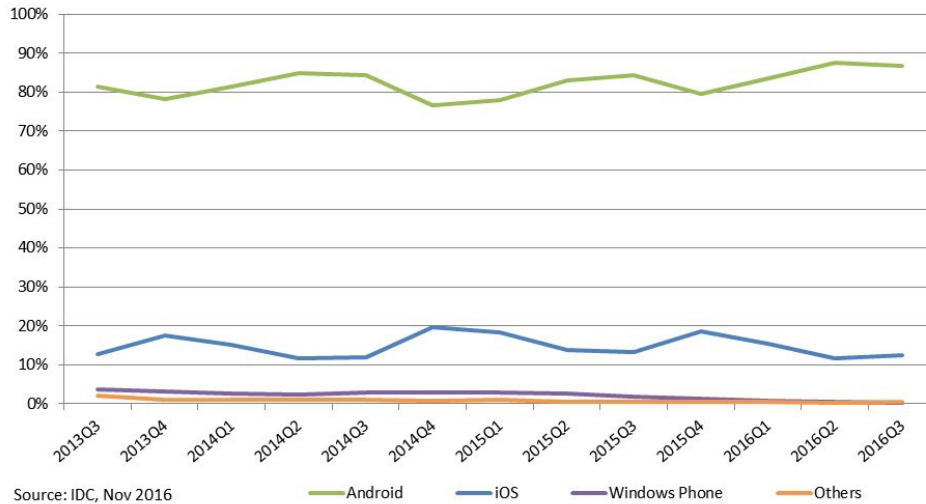


Figure 2.8: A chart of the smartphone OS market share worldwide. In the third quarter of 2016 Android and iOS together stood for 99,3% of the market share [27].

Runkeeper, which is an application with support for both Android and iOS smartphones, can be paired with smartwatches for both operating systems. While it's possible to fetch activities through the shim framework and Runkeeper API, it does not directly give details of an activity, but rather a list of all activities, meaning the shim framework must be expanded.

2.10 Athlete Monitoring

The Athlete Monitoring [28] is a system similar to pmSys. It is an application that uses the same concept of questionnaires to assess oneself and create a value on an RPE scale. It also provides much of the objective data that we would like to implement in pmSys. The functionality that Athlete Monitoring offers that pmSys currently doesn't, is objective data collecting and tracking. It supports collection of data like heart rate and GPS from multiple different sources by using spreadsheets. Our focus is implementing GPS data, and by doing some research on the Athlete Monitoring application we found that GPS data handling is a tedious process that requires an additional step in the process, by forcing the user to either insert data points manually in input fields, or format the data to comma separated value (CSV) files and then importing them. In result, if the GPS format is not already in the correct format, manual interaction is required regardless. We disregarded the use of this system because of the lack of flexibility and scalability. The system has no solution for integrating new data that is not already supported in the application. The Athlete Monitoring system would also imply a brand new stand-alone solution,

it cannot be integrated with the pmSys backend.

2.11 Summary

Our goal is to provide a proof of concept for injecting objective data into the pmSys backend, and for this pen and paper is not an option. By utilizing the systems and software covered in this chapter, combined with the following chapters, an automated process with minimal pre manual work can be created. One issue that might occur in the future if dealing with continuous stream of objective data is the transition into big data, which envelops new incoming data and all the historic data need to create an analysis of the collective objective health data.

When performing a GET request for the data provided by third-party data providers, there are much of that data which can be considered overhead regarding pmSys, and can be removed. For that the Ohmage based application Shimmer can be used to normalize the data using the common data schemas provided by OMH.

In the following chapters subjects regarding Shimmer, schemas and shims, Runkeeper and pmSys will be adressed and described in closer detail, followed by a conclusive proof of concept in chapter 7.

Chapter 3

Shimmer

In this chapter OMH's Shimmer application will be presented, and how it ties gathering, processing and presentation of health data and its usefulness to developers and end users together. The different components will be described, and lastly the implementation of the Shimmer framework in pmSys, and how it can provide objective data will be explained.

3.1 Shimmer components

Shimmer is a free open-source application that simplifies the process of gathering health data from third-party sources (see section 2.9). In other words, it is an open-source health data integration tool customizable to fit the individual product's needs. It will gather data and convert it into clean OMH compliant data that other applications can then utilize. The Shimmer application consists of several components, an individual shim for each of the already supported APIs along with JSON schemas for the data to be normalized into, a resource server and a frontend console.

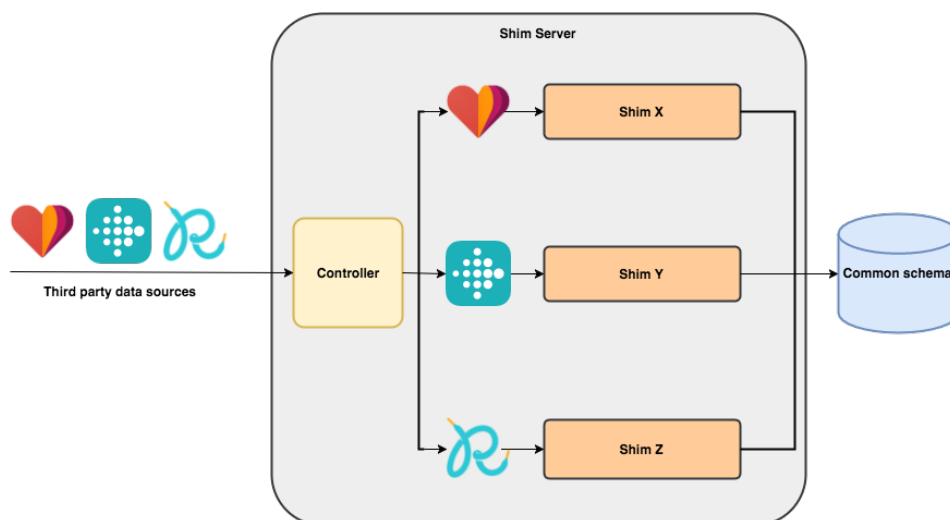


Figure 3.1: Architecture of the shim dataflow deconstructed from the greater infrastructure of OMH. Figure inspiration src: [29]

"Once the common API is implemented by a data provider, the shim for that provider is no longer used",[30], meaning the shims are a intermediate solution, rendering the shims void when directly implemented by the data provider. The shim framework is open source, and as such, all can develop new functionality for it. This means that the whole architecture can be placed in a solution, and shims can be added when it's necessary extract data not available at the time.

3.1.1 Shims

A shim is a library that communicates with third-party APIs, and in our case this API is Runkeeper. It handles the whole chain of interaction with the API from authentication, sending requests and mapping the data into an OMH compliant data format, and the schemas that describes what the normalized data should be formatted like. Data points are created by the shim, and these data points are self-contained pieces of data which includes a header with metadata such as creation date, data source and acquisition provenance to describe the data and where it comes from. The data points also contain the actual health data we want to retrieve. The shim also contains a mapper for each data point being made, access to a storage point containing credentials and access tokens, and finally a logic controller controlling the dataflow of authenticating and directing the incoming data to the correct shim as shown in figure 3.1.

3.1.2 Resource server

The resource server (shim server in figure 3.1) is responsible for handling the exposition of an API to retrieve data points from. The server also handles and delegates the API requests to the corresponding shim. As the number of developed shims added in the resource server increases, it is also becoming capable of providing additional data points from a growing number of third-party APIs. In addition to this, the resource server is also in charge of managing the third-party access tokens of behalf of the shims. The resource server is designed to be run within an existing infrastructure, and be accessed by this backend the same way that you'd deploy a database, an email server and other subsystems.

3.1.3 Console

The Shimmer application's console is a minimalistic and simplistic web user interface (UI) made to make the interaction with the resource server easier for the users. Here the users can change various configurations, add client id and client secrets (figure 3.2) and trigger the authentication flow between the application (Shimmer) and the API. The client id and secret is stored in an underlying MongoDB instance. The console also lets you request data from the APIs using date pickers and drop down menus.

Jawbone UP		
Jawbone UP clientId	Jawbone UP clientSecret	Save

Runkeeper		
f0cbfd831b764e19824f297df60c	331be02539564505a7e2ac4ab	Save

Google Fit		
508929859632-r353k0tpc6qhcr	cPnVQBO2ewoxlHQAEUJdUpj	Save

Figure 3.2: Client id and client secret added for the APIs connected with the available shims. Client secret and id is returned by Health Graph when registering an application in Runkeeper.

Find

👤

Withings
Connect

Runkeeper ✔
Disconnect

ACTIVITY

03/08/2017

📅

03/11/2017

📅

Raw

Normalized

CALORIES

03/08/2017

📅

03/11/2017

📅

Raw

Normalized

GPS

03/08/2017

📅

03/11/2017

📅

Raw

Normalized

Google Fit
Connect

Figure 3.3: List of the added APIs in the console. Each API has their own set of data available to request. For Runkeeper, these data sets are related to activities and calories burned, including the GPS data set we created for this research thesis. From here it is possible to request the raw data and the normalized OMH compliant data.

3.2 Schema

A schema is a specification of structure and format of data, and is structured as JSON format in OMH's attempt to unite health data into a common format. These schemas exist to express, process and gather health data as a single source of documentation regardless of where the data comes from. This is a measure to help increasing the usability and readability for health data, which is complex and in potentially in massive volumes. The purpose of these common schemas is to break down information into the smallest possible chunks, allowing data providers to minimize the amount of overhead, and giving the consumers the possibility to scrutinize individual pieces of data. The schemas is also a reference point for developers, enabling bootstrapping of an application quicker and more trivial. However, when dealing with health data, too much atomicity will

remove contextual meaning and important information can get lost.

3.2.1 Design principles

Schemas are designed to consider clinical measures and the gravity of their distinction in medical use. The schema should be able to measure one or multiple values where it is applicable and useful, so the aggregation of data is a simpler process. The schemas aim to offer an ideal format describing digital health data for clinical and self-care. The design principles are separated into these six categories [31]:

1. Atomicity

The schemas should present data at a granularity to be most useful, not restricted to traditional assumptions and standards about clinical care models. For instance, you don't need to know the prescribing doctor or other prescription data of a taken dose. This principle about granularity has often lead to more atomicity than the electronic health record (EHR) data standards.

2. Balancing parsimony and complexity

Health data can be highly complex. The schemas must be as comprehensive as needed for the majority of mHealth use cases, avoiding redundant complexity where it isn't appropriate. The schemas follow the closed-world assumption, implying that what is stated is true and what isn't stated is false.

3. Balancing permissiveness and constraints

Schemas that are too constrained and complex to use will not be adopted, but on the other hand, permissive and easy to use schemas may provide clinically meaningless data. This principle is to be pragmatic when balancing permissiveness and constraints. The ideal is to get accurate measures to the precision needed, and avoid internal consistency and inconsistency with absolute measures.

4. Designing for data liquidity

The Open mHealth schemas need to preserve the most important clinical meaning as mHealth data is passed along. The interchange also needs to preserve the meaning of the data, as provenance is equally important. To secure correct interpretation of the data, the context of the data points must be available alongside the actual data; the schema must keep track of the things done to it from its origin.

5. Alignment with clinical data standards

Open mHealth adopted widespread medical ontologies in their schemas, drawing from standard vocabularies where possible instead of reinventing sets. For instance, almost all of the units of measure used in the schemas come from Unified Code for Units of Measure (UCUM) Codes for Healthcare Units (see table 3.1)[32]. UCUM is a system of codes for unambiguously representing units of measure to humans and machines. This leads to developers not having to get into endless different medical terminology.

Valid UCUM code	Description
cm	CentiMeter
m	Meter
mL	MilliLiter
L	Liter
s	Second
min	Minute

Table 3.1: Examples of Unified Code for Units of Measure

6. Modeling of Time

The time perspective is a really important piece of information in medical data. The Open mHealth schema can represent both points in time and time intervals.

The balance between complexity and usefulness is important to note, and that it might be more appropriate to reduce complexity by removing less useful data, or move it to another schema, this also increases readability. The context of the original data measured need to be stored in a header included in the schema. This operational metadata ensures that the data is preserved when exchanged. All data need to follow the clinical standards, regarding vocabulary and granularity. Also by the use of enumerates the schemas has some attributes that only can be set to specific types or values. And giving all measurements timestamps will give the data more context.

```

{
  "activity_name": "walking",
  "distance": {
    "value": 3.1,
    "unit": "mi"
  },
  "effective_time_frame": {
    "time_interval": {
      "start_date_time": "2015-02-06T06:25:00Z",
      "end_date_time": "2015-02-06T07:25:00Z"
    }
  },
  "kcal_burned": {
    "value": 160,
    "unit": "kcal"
  },
  "met_value": 3.5
}

```

Figure 3.4: JSON schema structure including activity type, distance, timeframe and calories burned. Figure src: [33]

To be able to add a third-party API and request data from it, you must first visit the developer segment of the API you wish to add, and register a client application. In the registration process, you're able to request permissions regarding information retrieval, editing and retaining. You must also specify the name of the application, a short description and a redirect/callback URL to which the user is sent to after granting your application access to the data. The information you provide in this process is forwarded to the end users, and they can manage operational concerns like authorization and rate limits. Once the registration is completed, you will be given a set of credentials. These credentials are then used to identify your application in the API when it's used, and is entered as shown in figure 3.2.

3.3 Grant Type: Authorization Code

This is the grant type Health Graph operate with through shimmer, so an in depth explanation will be given here. "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." [34] This is commonly used for server-side applications, as the source code is not exposed. This is important since the confidentiality of the client secret must be maintained. The client secret is a token generated when a developer registers an application.

Step 1: Authorization Code

The user receives a link through some form of media, for example email. The link will look something like this:

```
https://runkeeper.com/apps/authorize?state=xxx&client_id=CLIENT_ID&response_type=code&redirect_uri=CALLBACK_URL
```

The `response_type` here is the code which specifies that your application is requesting an authorization code grant. The `CLIENT_ID` is your application id, which is how the API identifies your application. `CALLBACK_URL` is where the user is redirected after the code is granted.

Step 2: User authorization

When the user clicks the link given in step 1 and log into the service, they will be prompted to authorize or deny the application.

Step 3: Authorization code is provided

If the user agrees to authorize the application in step 2, the service will redirect to the specified `CALLBACK_URL`, with the authorization code.

```
https://runkeeper.com/apps/authorize
```

Step 4: Application request access token

Now that your application has all it need for requesting access, a POST request is done to the API token endpoint. Here the authorization code and authentication details must be passed.

Step 5: Application receives access token

If the POST request in step 4 validates, the API will respond with an access tokens and if applicable, optional information, for example a `expires_in` value or `scope`, the level of access the service is asking for.

```
{
  "access_token": "ACCESS_TOKEN",
  "token_type": "bearer",
  "expires_in": 2592000,
  "refresh_token": "REFRESH_TOKEN",
  "scope": "read"
}
```

Figure 3.5: OAuth access token response from a service.

Now the application is authorized to use the service, limited to its scope. The access token can be used until it expires, or a refresh token is used. A request using curl to the Runkeepers API will look like this:

```
curl -i https://api.runkeeper.com/$RESOURCE -H "Accept: */*"
-H "Authorization: Bearer $ACCESSTOKEN"
```

This will fetch whatever JSON data at that url.

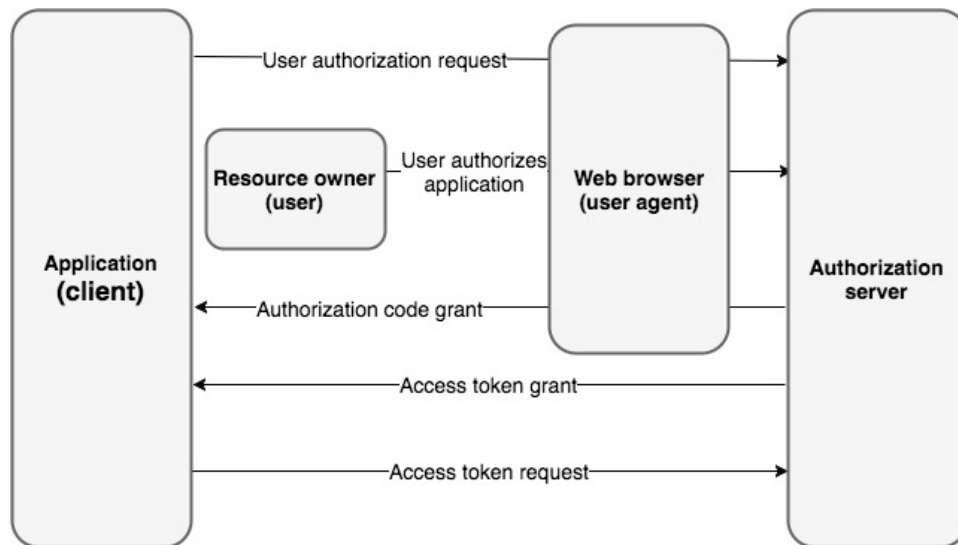


Figure 3.6: Interaction flow for the grant type authorization code. Figure inspiration src: [18]

3.4 Installation

For the sake of this research, we installed the Shimmer application locally, built the code natively and ran it in a terminal on an Ubuntu operating system. This way of running the application and console has a couple of requirements, you have to install a Java 8 or a higher Java Development Kit (JDK), Node.js, and have a running MongoDB instance. When run for the first time the bash build script resolves a few dependencies using npm. It installs Grunt and Bower and creates a symbolic link (symlink) in the Grunt output directory from source files in the application. When implementing and deploying Shimmer in pmSys, you can use Docker 2.6.

3.5 GPS in Shimmer

With OMH's initiative (See section 2.5) in mind, we wanted to explore the possibilities of making the data we retrieve structured in a way that made it easier to manage and process by others. OMH has developed a lot of schemas for this purpose, including schemas for heart rate, calories

burned, physical activity etc. [35]. Figure 3.7 and figure 3.8 show how raw data from a physical activity is shimmed to the physical activity schema.

```
{
  "size": 1,
  "items": [
    {
      "duration": 8,
      "start_time": "Tue, 28 Feb 2017 00:00:00",
      "total_calories": 3,
      "tracking_mode": "outdoor",
      "total_distance": 69.598770321565,
      "entry_mode": "API",
      "has_path": true,
      "source": "Developer's Console",
      "type": "Running",
      "uri": "/fitnessActivities/940568831"
    }
  ]
}
```

Figure 3.7: SON raw data before it is sent through the shim.

```

{
  "header": {
    "id": "43f467a4-9f38-45c3-b4dc-c7a3601d2f6e",
    "creation_date_time": "2017-03-01T14:38:20.576+01:00",
    "acquisition_provenance": {
      "source_name": "Runkeeper HealthGraph API",
      "external_id": "/fitnessActivities/940568831"
    },
    "schema_id": {
      "namespace": "omh",
      "name": "physical-activity",
      "version": "1.2"
    }
  },
  "body": {
    "activity_name": "Running",
    "distance": {
      "unit": "m",
      "value": 69.598770321565
    }
  }
}

```

Figure 3.8: JSON data after shimmed to OMH compliant data.

It was not developed any schema or shim for GPS data. This had to be developed by following the design principles[36] and a template [37]. The development is addressed in section 7.2.

3.6 Summary

In this chapter we have covered how Shimmer works by presenting its components and the logical construct. Shimmer is a processing software created by OMH to create compliant data which can be used in the schemas defined by OMH. By using Shimmer, the exchange of data between systems with the same schemas is an easier process as the sender and receiver has the same data formats. This is basically OMH's vision for the future of health data. Shimmer also provide the users with a solid authorization process that is the OAuth framework. This is necessary as the data in question is health data, which is considered sensitive data. By exchanging multiple "handshakes", and returning unique authentication tokens, the data can be processed according to API specifications.

By following the design principles covered, new schemas can be created for data that might not be covered in Shimmer at this time. This makes the scalability of Shimmer as mediator for data processing a great option

for all systems which has health data as input. PmSys has already implemented some of the other systems that OMH has developed, which would make Shimmer an easier installation as it will be a plug-in to the existing systems.

Chapter 4

Runkeeper and Health Graph

As Runkeeper is the focus regarding implementation with a third-party providing the new objective data, this chapter will explain a more detailed overview to what is currently accessible with the endpoints, and what useful data they can provide in the context of pmSys. Runkeepers endpoints is not directly exposed through their own services, but rather through Health Graph which is powered by Runkeeper, helping developers creating their application and visually display in their own formats.

4.1 Runkeeper

Runkeeper is an application for smartphones, which helps you set personal goals, track workouts and progress, follow a plan and help you stay motivated. The application is also focusing on creating a community, and currently has more than 50 million users. Using the application, users can connect to their social media like Facebook and follow their friends activities and progress. Users can also view their friends competition or motivation, and see how their activities stack up to other users. Runkeeper can also tailor a workout to suit your needs and act as a personal trainer.

As other similar applications, Runkeeper offers a "plus" subscription called Runkeeper GO. This is a payed subscription which will offer a user new features otherwise unaccessible. This includes live tracking of your GPS data for others to view, multiple advanced fitness reports and more.

4.1.1 Runkeeper application

The Runkeeper application is supported both for Android and iOS, and is geared towards runners, walkers and other spatiotemporal activities. The app takes advantage of the featured GPS technology within the smartphone or smartwatch, making route tracking possible. Utilizing smartwatches for route tracking requires the smartwatch to have its own built-in GPS (see section 4.1.2). The application also lets you add context to the workout with custom notes, describing equipment used and what the weather was like. In the Runkeeper web application, you can view a

detailed summary of a workout, including data about pace, distance, time and the route taken on a detailed map. Runkeeper also provides simple and effortless uploading to social media if so desired.

4.1.2 Wearable hardware

It is required for the wearable to have its own built-in GPS to be able to replace the smartphone's tracking ability. Such wearables include the Smart Watch 3[38] and the Moto 360[39]. By implementing this technology in smart watches, the consumers are able to go jogging, using for example Runkeeper, without needing to carry a heavy and impractical smartphone. In general wearables need to be synced with the phone, and the wearable should automatically synchronize with the Runkeeper application as long as the device is within bluetooth range of the smartphone. During an activity recording, the users are able to view stats or to pause or stop the tracking 4.1. This is a nice feature when dealing with football practice as the coach(es) might want to interrupt the session to give instructions, or if the staff only wants to monitor certain parts of the session [40].

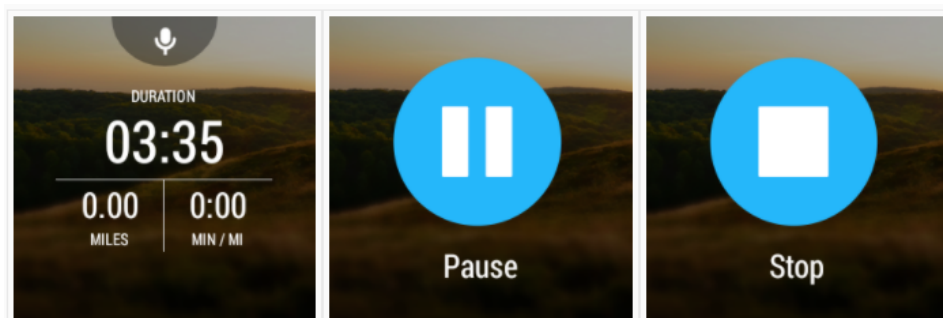


Figure 4.1: The user can pause or stop the recording during workout through a smartwatch, and view available stats [40].

4.1.3 GPS accuracy

Runkeeper has a few different options to optimize the capturing process as much as possible. Depending on the training environment, these can be adjusted to get better precision. Opening the Runkeeper application a few minutes before starting a session to let the wearable calibrate with the satellites and lock a GPS signal is preferable. This way the tracking can begin instantly when a player enters the pitch. In the Runkeeper app you can choose between "Device Only" or "High Accuracy" GPS mode.

If the session is planned to be indoors, the best choice is to set it to "High Accuracy", as this will not only utilize the device's internal GPS, but also WiFi/4G and cell signal. Newer phones and smartwatches have components with higher performance, and those based on the integrated circuit SoC (System on Chip) is better designed for this purpose as it reduces the possibility of component-interference with the GPS [41].

4.2 Health Graph

"In words, the Health Graph is: A system of individual health connections or interrelations - a digital map of your personal health." [42] Health Graph in its essence is an API providing access to the user data, completed fitness activities and health data captured by a number of different health tracking applications, including Runkeeper. Health Graph can provide users with snapshots of their current physical state, and their progress over time. This can help gain insight in how behavioral patterns can contribute to changes in health and lifestyle, this can also include how social interactions influence these changes. For example, you see your friends signing up for a 10K, and at a later time you do the same. All this data can establish correlations between nutrition, sleep, social motivation and the general activity frequency or performance, which can be visually displayed to you.

Just as Runkeeper is built upon Health Graph, developers can create their own applications, and by integrating it with Health Graph they can access data from the Health Graph supported applications like Runkeeper, Withings and Jawbone. The Health Graph API can in this way jumpstart a developers app, skipping the need to create their own endpoints. Developers has access to all available endpoints, and they can share their application through social media, or even incorporate the Runkeeper GO feature and earn revenue through sales and subscriptions.

4.2.1 API

"The Health Graph API is a portal to the Health Graph's robust data set." [42] Health Graph contains a collection of web-based resources, which can be accessed through use of the API, and are referred to as nodes in the Health Graph documentation [43]. The API consists of the OAuth2.0 token authentication (see section 2.8) and the collection of endpoints which contains a user's data, and his or hers activity sets. When registering an application with the Health Graph application portal, the application will be assigned a client id and client secret. The client id and client secret values must be sent along with grant type, code and redirect uri through a POST request to the Health Graph token endpoint in the format application/x-www-form-urlencoded. The response will contain an access token which will uniquely identify the user with the application. This access token is contained within the application and will never change unless the developer disconnects the application. The reason for this is that Health Graph does not use a refresh token or expire timer when authenticating users. Example of the response using the GET operations for fetching data sets:

```
GET /team HTTP/1.1  
  
Host: api.runkeeper.com
```

```
Authorization: Bearer xxxxxxxxxxxxxxxxxxxx
```

```
Accept: application/vnd.com.runkeeper.TeamFeed+json
```

This will fetch all the friends you have added in Runkeeper in a JSON structure with the fields:

```
{
  "size": 2,
  "items": [
    {
      "profile": "http://www.runkeeper.com/user/xxxxxxx",
      "name": "Username",
      "userID": "xxxxxxx",
      "url": "/team/xxxxxxx"
    }, {
      "profile": "http://www.runkeeper.com/user/xxxxxxx",
      "name": "Username",
      "userID": "xxxxxxx",
      "url": "/team/xxxxxxx"
    }
  ]
}
```

Figure 4.2: A JSON response when using the GET method on the TeamFeed endpoint in Health Graph.

4.2.2 Health Graph console

This is the developer tool for testing all the HTTP methods available: GET, POST, PUT and DELETE. GET is used to fetch data directly, POST is used when data is created and stored, PUT is for modifying data, DELETE is for deleting. For each of these operations is also a HEAD method which retrieves the header corresponding with the JSON message-body. Through this console a developer can test the different API responses before creating any parts of an application. The bearer token is automatically retrieved and stored in the console, and is ready to use without any configurations.

Method/Header	Value
GET	/user
Authorization:	Personal bearer token
Accept:	application/vnd.com.runkeeper.User+json
If-Modified-Since:	
Content-Type:	

Figure 4.3: The console with inputs for a GET response fetching individual user information[44].

4.2.3 Limitations

There are an abundance of endpoints which can be used to supply an application with data that can be stored or fetched directly from Health Graph's data storage, and be presented to the end user. But one major limitation, that directly correlates to our vision to supply pmSys with collections of data which can be used in comparison to a single or multiple users, is that there is no endpoint that directly performs a GET request on all detailed data. This can only be done through supplying an endpoint with a specific activity id for a user with his corresponding authentication bearer token. This results in multiple GET requests, and slows down the process of retrieving the data significantly.

Another limitation, which in theory could have solved the previous mentioned, is that there is no support for creating and adding new endpoints. Instead of just making the endpoints open and usable for third-parties, Health Graph could have presented their API as open-source, making it possible for developers to create new functionality.

Lastly the response returned by many endpoints returns much data which is considered overhead regarding pmSys. This can be a direct link with increasing the read operation when fetching data. By denormalizing the database (see section 7.5.4) you can increase the speed of read operations by putting relevant data in the same entity to reduce complex logics, but it decreases write, update and delete operations. For pmSys this results in more processing after getting the data to remove unnecessary values.

4.3 Summary

We put Runkeeper and its API created by Health Graph in focus as it is the only software that supports all our criteria for supplementing pmSys

with useful objective data. Runkeeper has a vast amount of users, and there is no limit to how many users can be a part of data collection using the API. This makes the perfect environment for scalability and expanding the user base in pmSys. As mentioned Runkeeper can be used in combination with many variations of physical activity tracking wearables, which makes the application great for equipping a minimal amount of hardware on your person when tracking physical activities. Most important is that the wearable support GPS to utilize the potential in objective data gathering. By setting up players with a Runkeeper account, accepting it through pmSys, and gearing the with hardware, one can with minimal effort start collection data. It is important to simplify the process so we can separate the user from underlying technical efforts. We also covered some of the limitations that presented themselves during our research. If these limitation are handled on Health Graph's side of the table, some processes can be changed and simplified significantly.

Chapter 5

PmSys

5.1 Current pmSys

5.1.1 PmSys Mobile Application

PmSys was developed as a tool for collecting, storing, analyzing and presenting the player's subjective opinions regarding their training load and current physical wellness, as well as a way of reporting injuries. Rating of Perceived Exertion (RPE) is a scale commonly used in both medical studies and as an intensity description in training sessions [45]. This is an important scale used in the surveys in pmSys to discover the intensity of a session. The scale ranges from 0 (rest) to 10 (maximal) as shown in figure 5.1. PmSys is built on the Ohmage Mobile Web Framework which supports multiple datatypes that the user can send in: number, string, timestamp and other data types. The data gathered with this scale gives the coaching team an overview of how a player subjectively perceives the workload of a given session. The coaching staff can then gather a general consensus of the intensity, and make appropriate adjustments if required.



Figure 5.1: The RPE scale ranging from 0-10 [46].

Each player must fill out surveys in the pmSys-app after each training session, answering questions related to their perceived training load and wellness 5.2. The questionnaires are developed in collaboration with a NiH PhD study, and is what gives the RPE scale a value for each survey taken for a player.

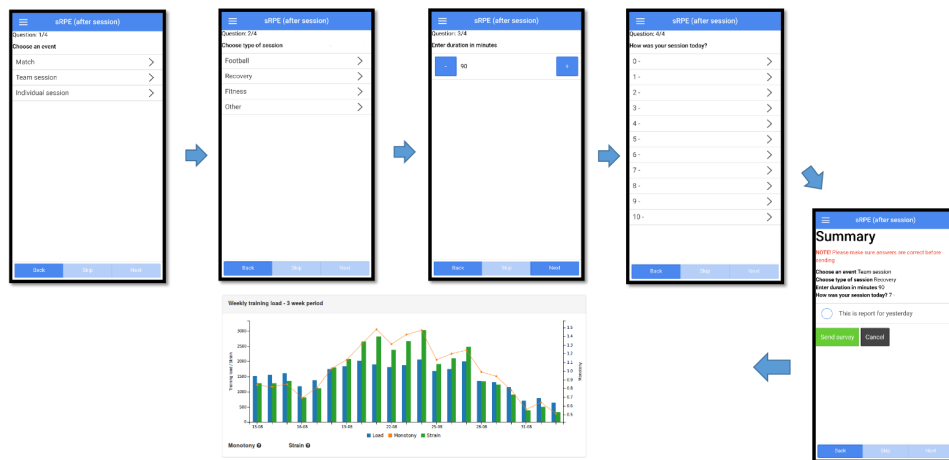


Figure 5.2: The workflow of which the players go through when reporting a training session.

5.1.2 PmSys-trainer

PmSys-trainer is a web portal that is used to read and present the data collected through the mobile application. The key part of the web portal is to present health data which has value for a trainer, meaning he can read useful data and adjust training or advice accordingly to make improvements for a player and for the team. All data, graphs and modules which is deemed unnecessary is excluded for the sake of keeping the web portal as simple, fast and easy to use as possible.

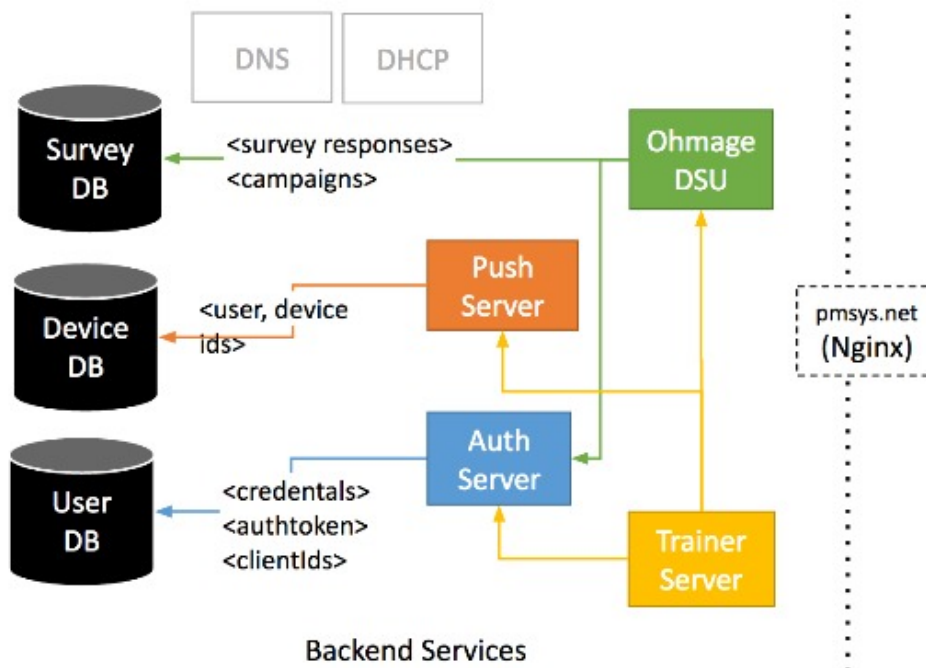


Figure 5.3: Representation of the backend structure in PmSys.

5.2 PmSys with objective data

5.2.1 Motivation

Accelerometers has been around since the 1950's, but was not adopted into physical activity monitoring systems until the 1980's [47]. Compared to today's hardware, which includes accelerometers and gyroscopes, the early accelerometers that was used to track activities was not very reliable. This stemmed from low accuracy and calibrations in the data, and a high price for it's applications. Today, multiple wearables can be used to gather more data by the second to assess with precision a player's actual physical condition. Most of these types of wearable comes with a 3-axis accelerometers to measure movement in all directions, and some includes a gyroscope to measure rotation and orientation. The subjective data which pmSys now gathers is not enough to prove the physical state, injuries or wellness of a player. The reason is that the subjective data is a personal assessment of oneself, and given the personal input it is easy to tamper with the data or lie during a survey.

By supplying the existing pmSys backend with objective data through wearable devices and mobile applications the trainer web portal can be extended to visualize even more useful data sets. The objective data compared to the subjective data, which are gathered through the surveys mentioned earlier, has next to no manual interaction, and will generate a continuous stream of data points based on monitoring a player during training. This data can be for example GPS coordinates, pulse, heart rate, blood pressure and calories burned. All these data can be stored in a database and be used in aggregation algorithms or queries to look for patterns during a player's training and create a representation in the trainer web portal. In example, a map can be created using the longitude and latitude data to see the movement of players, find bad habits and how the players move and position relevant to each other. This data can help to an even greater extent to fine tune the team's performance.

5.2.2 Injecting shimmer into the backend

Shimmer will mostly be a standalone plug-in to the existing pmSys backend, only needing access to the authorization server for storing the new OAuth tokens supplied by the Health Graph authorization API 5.4. A POST/GET request will be done to the authorization API, and a GET request to the data API for fetching a pmSys user's activity data.

The pmSys backend is currently structured using docker containers for isolating the systems running pmSys. Shimmer can be configured to run in a cloud or a docker container, but for integration purposes the natural choice would be to put Shimmer with the current backend. By placing Shimmer in the existing pmSys server it also offers higher capability and performance than it would by fragmenting it on a cloud based solution.

The different Docker containers that makes pmSys is built and run using Docker compose 2.6.1, this means that Shimmer could be added as a dependency in a docker-compose file making it automatically download and configured. This makes deployment of pmSys including Shimmer a simple process for any environment supporting Docker containers. Shimmer has it's own deployment procedure configured in a docker-compose.yml file, which handles the deployment of the three primary components: the individual shims, a resource server and the frontend console.

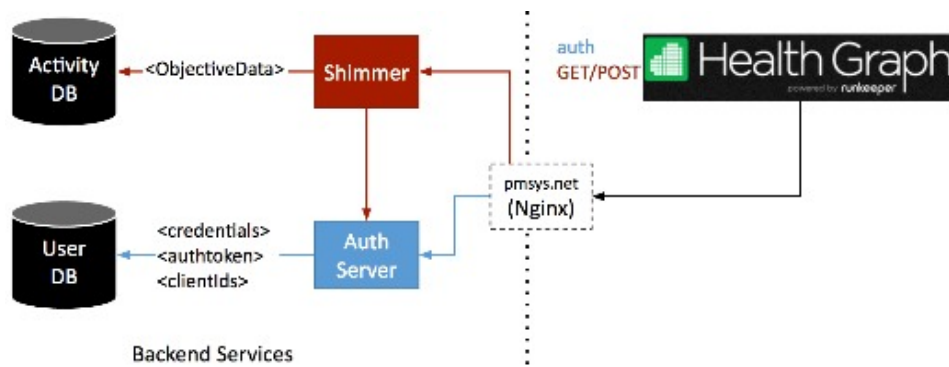


Figure 5.4: Representation of the backend structure in pmSys integrated with shimmer.

5.3 Summary

As pointed out in this chapter, the subjective data coming from users answering questionnaires through surveys is not enough to precisely evaluate a player's physical health. The RPE scale can only provide an understanding of a player's general assessment of themselves, and give a perspective of how they feel at certain points during a timeframe. The surveys are visualized to trainers and can evaluate their own exercise regime. By supplementing the backend with objective data we can contribute to a deeper understanding of physical performance.

Objective data also contribute to effortlessly provide data that has limitless uses. By applying aggregation algorithms and logical queries on the objective data, one can find correlating patterns in time, location, mood and frequency of exercise that otherwise is too obscure to detect. By comparing the aggregated objective data provided by wearables and application, and the subjective data from user input one can eventually create a clear picture of how it all ties together.

Chapter 6

Testing

The solutions tested are one using Shimmer, and one using bash scripts. Both which has their advantages and disadvantages. The data fetched from the Health Graph API contains a lot of unnecessary data, which means fetching it will be a more demanding HTTP request on the system. And the data must also be processed after the HTTP request has finished. The solutions for fetching data is using the same data set which contains 500 items. Each item contains 10 keys with corresponding values, which translates into 5000 lines of JSON data, and even more when working with the detailed activity data.

Test environment

- **Native OS:** OS X El Capitan
- **Virtual machine:** Ubuntu 14.04 LTS
- **Average download speed:** 80 Mbps
- **Java version:** 1.8.0_111
- **Curl version:** 7.35.0

6.1 Using Shimmer

This solution utilizes Shimmer, which is the intermediate data handler used for health data. This solution is the fastest given that the bearer token, client id and secret is stored in MongoDB. The table 6.1 shows how much time it takes to get a connection to the Health Graph API server and then download the content stored there. Time to first byte is a measure for the responsiveness of the server, from the HTTP request is sent until receiving the first byte on the client.

As shown in the table, the bottleneck here is API server response time, and the actual download is just a fraction of the total download time. The data being fetched is the undetailed data which does not contain GPS data, but

the activity id which can be used for a new GET request on the endpoint containing the detailed data which includes the data points for GPS.

Description \ Time	Shortest	Highest	Average
Time to first byte	332,22 ms	814,72 ms	428,02 ms
Content download	0,28 ms	4,12 ms	1,1 ms
Total download time	332,50 ms	818,84 ms	429,12 ms

Table 6.1: Test results for fetching 500 activity id's from Health Graph API using Shimmer.

By shimming the data fetched earlier, the activity id can be extracted and be supplied to the endpoint returning detailed data containing the needed GPS data points. This is much larger bulks of data containing much unnecessary data for the purpose of pmSys, but must still be downloaded in order to shim it as it is included in the response. The table 6.2 shows detailed timing for fetching the detailed data.

Description \ Time	Shortest	Highest	Average
Time to first byte	3,02 m	4,41 m	3,52 m
Content download	4,4 s	6,2 s	4,9 s
Total download time	3,06 m	4,47 m	3,57 m

Table 6.2: Test results for fetching the detailed raw data of 500 activities with Shimmer.

6.2 Using bash script

This solution is a bash script which utilizes curl, which is a command line tool which fetches or sends files using URL syntax. Curl makes a http GET request against a supplied website, which in this case is <https://api.runkeeper.com/fitnessActivities?pageSize=100000>, the API endpoint for all activities associated with a user, the pageSize parameter is defined as the standard response only will show the 25 items of the data set. Supplying curl with the correct headers: -H "Accept: application/vnd.com.runkeeper.FitnessActivityFeed+json" -H "Authorization: Bearer \$BEARER", where the variable BEARER is the unique bearer token for a user, will fetch all unsifted data belonging to the user identified with the bearer token.

The table 6.3 shows the time it takes to do a GET request for all the raw data, and have it returned by the Health Graph API. The script also extracts the relevant activity id's which can be stored and used in the MongoDB in pmSys. This is the first step to isolating the detailed activity data.

Description \ Time	Shortest	Highest	Average
Time to first byte	692,52 ms	1455,66 ms	796,98 ms
Content download	33,06 ms	141,05 ms	83,77 ms
Total download time	784,01 ms	1490,65 ms	880,76 ms

Table 6.3: Test results for fetching 500 activity id's from Health Graph API using curl.

The activity id's stored combined with the bearer token will be used to fetch the detailed activity data, including all GPS data points, distance, duration, heart rate and more. For each detailed activity there is 70 keys with values, and with the data set of 500 items, it calculates to 35.000 lines of JSON data. This GET request is from https://api.runkeeper.com/fitnessActivities/activity_id, using the same curl method as explained earlier. The table 6.4 shows the time it takes to fetch all 35.000 lines of JSON data. The data again includes much overhead, data unnecessary for pmSys, so for storing purposes much can be removed.

Description \ Time	Shortest	Highest	Average
Time to first byte	4,21 m	4,50 s	4,39 m
Content download	11,3 s	16,1 s	14,1 s
Total download time	4,32 m	5,06 m	4,53 m

Table 6.4: Test results for fetching the detailed raw data of 500 activities with curl.

6.3 Summary

Looking at the results in the tables presented in this chapter, Shimmer clearly offers higher performance for HTTP requests against the Health Graph API. The reason is as stated earlier that the client id, client secret and bearer token is more accessible as they are stored in the underlying data storage in Shimmer. But the downside of Shimmer is that it is a more tasking process to integrate it as a docker instance together with the other docker containers existing in the pmSys backend. Shimmer has to exist as an intermediate layer for processing and storing data from Runkeeper or other third-parties.

The bash script offer worse performance than Shimmer for processing HTTP requests, but is an easier installation that does not require a new docker container, only a plug-in to the existing Ohmage data storage unit (DSU) in the pmSys backend. A major limitation that applies to both solutions is that there is no endpoint in the existing Health Graph API that re-

quests all detailed activities for a user, which includes GPS data points. This results in one new HTTP request per activity requested, which includes the authorization process, this is why the request for detailed data is exponentially slower in total, as there is one new "time to first byte" for each detailed activity fetched.

Chapter 7

Proof of concept

In this chapter we present a proof of concept that was developed for this thesis, and it's system requirements. It will describe how to implement objective data with Shimmer, and present the schema and shim developed for GPS data specifically. In addition, this chapter will address each step on how to get GPS data from Runkeeper into pmSys, including sequence diagrams and a visual design mockup to describe the process and functionality, and a detailed design of the database handling.

7.1 System requirements

This section present the system requirements of the implementation of the Shimmer plugin in pmSys, and outlines the general requirements considering an actual implementation of a passive data collection application.

7.1.1 Functional requirements

By expanding pmSys with Shimmer, there are some functional requirements that have been outlined. This section presents an overview of the general functional requirements imposed to the system.

Connect pmSys user and Runkeeper user

By connecting the pmSys users to their corresponding Runkeeper users, we are able to map the activity data accordingly through data joins of specific ids. The connection process should be executed by the user through a simple button press using the pmSys application.

Collect data

The Shimmer application will do a GET request of the GPS data points from the Runkeeper API (Health Graph). This process will preferably be an automated process, by doing daily/weekly batch processing. An alternative solution, or additional feature, could include the fetching through an on-demand function in the pmSys-trainer web portal.

Normalize data

The response from the GET requests will be normalized by Shimmer to fit the Open mHealth common data schemas. Normalizing the data gathered from potentially many different sources to a single common format will enable the processing and aggregation of the data to be executed in the same manner.

Store data

The data gathered from Runkeeper should be stored in pmSys' own database system. This way the system won't rely on the uptime of Runkeepers' servers. Storing the data in a efficient way for aggregation is also a priority.

7.1.2 User stories

These user stories creates a simplified overview of the functional requirements in practice. The user stories are meant to describe the type of users utilizing the system, functionality the new implementation is offering the users, and why it is beneficial.

As a	I want to	so that
Player	connect my pmSys user to Runkeeper	recording sessions are possible
Player	start recording a session	my sessions are stored in pm-Sys
Trainer	see my players movement during training	I can evaluate their training performance
Trainer	view different player stats from sessions	I can select the starting line up

Table 7.1: User stories

7.1.3 Non-functional requirements

When expanding pmSys with Shimmer, there are some non-functional requirements that have been taken into consideration. This section forms the outline of the main non-functional requirements.

Usability

We want the implementation of Shimmer to be as seamless as possible, with an intuitive layout and only require simple interactions from the users. We cannot assume that all the users of the system have the same high

technical competence. By maintaining pmSys' high usability, and avoiding big changes in the user pattern, it's more likely that the players will adapt to and use the added functionality.

Scalability

The backend solution of the expansion must be scalable, as the Shimmer application supports many different third-party APIs. Different teams may want to use different wearable technology devices, and the system should provide this opportunity.

Integrity

To maintain data integrity across the different data storage units is essential. The system must have the capability to ensure that data is not modified or deleted without authorization or without pmSys detecting such transactions. PmSys must protect data integrity by performing data integrity checks.

Performance

When expanding pmSys, it is important that the impact on the current system performance is minimal. This is important for the user experience, and will affect how we design the implementation.

Robustness

Having a robust application and handle errors that arise during runtime properly, will ensure that fewer crashes happens in the application. It's not guaranteed that an application is used as intended every time, and therefore it is necessary to handle such behavior. Ensuring high robustness will lead to less inconvenient maintenance and troubleshooting.

Privacy and security

Privacy and security is important in pmSys since the system gathers personal data about the players' health. In the gateway of pmSys there is a mapper which maps the usernames to the internal id's. The reason it is constructed this way is to prohibit the opportunity of identifying the users and tie specific data to an individual. These internal user ids must be protected and hidden at all costs as requested by the Norwegian Data Protection Authority (Datatilsynet). The data stored in pmSys cannot be tied to an individual user, unless you have been granted permission by the Ministry of Justice and Public Security. Storing the new data sets will have to follow the same guidelines as the existing data.

7.2 Working with Shimmer

Shimmer is a software component that will add a specific feature to the existing pmSys application, and can be categorized as a plug-in installed as a Docker instance (see section 2.6). Shimmer will add objective data to pmSys, and adding the possibility of aggregating this data to create and generate meaningful data.

The open mHealth (OMH) organization briefly describes their Shimmer application as “an application that makes it easy to pull health data from popular third-party APIs like Runkeeper and Fitbit” [48]. Shimmer currently supports the APIs from Fitbit, Google Fit, Jawbone UP, Misfit, Runkeeper, Withings and iHealth. There are four APIs whose supports are in the works, which are Moves, Strava, FatSecret and Ginsberg. Out of all these APIs, the data provider most fitting for gathering GPS data from training sessions is Runkeeper. Runkeeper also supports devices and wearables for both Android and iOS, which is beneficial (see section 2.9).

Shimmer’s shim for Runkeeper only included data related to calories burned and physical activity, and trimming this data to their respective schemas. The Runkeeper shim in Shimmer had to be expanded to be able to serve its purpose. All the shims and schemas are written in Java, so the expansion of the Runkeeper shim and the GPS schema were also developed in Java. Shimmer is a free open-source application, and the newest version of application can be found in OMH’s repository on GitHub [49]. This repository can be forked and expanded accordingly.

7.2.1 GPS schema

Open mHealth has developed just over 90 common data schemas specifying the format and content of data, based on a set of design principles trying to balance simplicity and usefulness. This schema library defines meaningful distinctions for the clinical measures. The common data schemas used in the Runkeeper shim for Runkeeper data are calories burned and physical activity. To illustrate, the calories burned schema represents a single measure of the amount of calories burned in kilocalories (kcal) utilizing another schema, OMH’s Kcal unit value. Sample data is shown in figure 7.1.

```
{
  "kcal_burned":
  {
    "value": 160,
    "unit": "kcal"
  }
}
```

Figure 7.1: Sample data of calories burned pulled from Runkeeper, and shimmed with Shimmer.

Some of the schemas include references to other existing schemas, so that the values are formatted accordingly. The `kcal_burned` value in the sample data in figure 7.1 has been formatted to fit the `kcal` unit value schema, which provides the actual value and a description of which unit of measure being used. The physical activity schema represents a single episode of physical activity in a similar fashion (see figure 3.4), and is also utilizing the calories burned schema.

To follow the design principles of the common data schemas, we defined which data would be fitting to include in a GPS data point in `pmSys`. The core elements of a GPS signal is the numeric values of the longitude, latitude and altitude. After researching and discussions, we landed on including five types of data:

```
private String activityName;
private DurationUnitValue duration;
private Double longitude;
private Double latitude;
private LengthUnitValue altitude;
```

activityName

This value describes the context of the GPS data point using the existing schema activity name. This is not necessarily a value related to the actual GPS coordinates, but it gives a contextual understanding of the data.

duration

As described in the design principles 3.2.1, the time perspective is important in clinical data. The movement frequency of a player often deplete as time goes by, making this data important in the bigger picture. This value incorporates another schema, `Duration Unit Value`, which decomposes the given value and represents it in seconds, the appropriate time unit for this purpose. This way, it also follows the principle of atomicity.

longitude and latitude

Longitude and latitude are values related to the coordinate system that uniquely defines points on the surface of a sphere, in this case earth, and has no common synonyms in any code set. To avoid redundant complexity, these values are not made granular.

altitude

We described in 2.1 why altitude is an important piece of data. This also follows the principle of atomicity, as the value also incorporates another schema, Length Unit Value, which decomposes this specific value and represents it in meters, the correct length unit in this context.

```

{
  "shim": "runkeeper",
  "timeStamp": 1491325415,
  "body": [
    {
      "header": {
        "id": "0e6d7831-e862-4481-9a81-5aa455fd2020",
        "creation_date_time": "2017-04-04T19:03:35.993+02:00",
        "acquisition_provenance": {
          "source_name": "Runkeeper HealthGraph API"
        }
      },
      "schema_id": {
        "namespace": "omh",
        "name": "gps",
        "version": "1.0"
      }
    },
    "body": {
      "activity_name": "Running",
      "duration": {
        "unit": "sec",
        "value": 0
      },
      "longitude": -70.951823,
      "latitude": 42.31262,
      "altitude": {
        "unit": "m",
        "value": 8
      }
    }
  ]
}

```

Figure 7.2: Sample data of GPS data pulled from Runkeeper, and shimmed with Shimmer using the designed shim and schema.

7.2.2 GPS data point mapper

The shims in Shimmer has support for gathering different measures depending on which API it sends requests to (see table 2.1). These measures are retrieved from the different API endpoints implemented in each shim. The available measures from Runkeeper was activities and calories burned supplied by the the endpoint <https://api.runkeeper.com/fitnessActivities/>. The acceptable media type for these responses is:

application/vnd.com.runkeeper.FitnessActivityFeed+json.

To offer retrieval of activity details, which is the only way of retrieving the GPS data points, we had to add support for the endpoint:

https://api.runkeeper.com/fitnessActivities/activity_id

in the Runkeeper shim. The media type accepted from the response from this endpoint is application/vnd.com.runkeeper.FitnessActivity+json. Shimmer then maps the defined response values to the keys in the schema to normalize it:


```
String activityName = asRequiredString(itemNode, "type");
GPS.Builder builder = new GPS.Builder(activityName);

setEffectiveTimeFrameIfPresent(itemNode, builder);
asOptionalDouble(itemNode, "timestamp")
    .ifPresent(durationInSec -> builder.setDuration(new
        DurationUnitValue(SECOND, durationInSec)));
asOptionalDouble(itemNode, "longitude")
    .ifPresent(longitude -> builder.setLongitude(longitude)
    );
asOptionalDouble(itemNode, "latitude")
    .ifPresent(latitude -> builder.setLatitude(latitude));
asOptionalDouble(itemNode, "altitude")
    .ifPresent(altitudeInM -> builder.setAltitude(new
        LengthUnitValue(METER, altitudeInM)));
```

7.3 Connect pmSys user to Runkeeper


The first step to being able to use Runkeeper in pmSys is to connect each pmSys user to their corresponding Runkeeper user. The pmSys application has to be registered in Runkeeper and get a client id and secret so that the application can be authorized by the Health Graph API authorization endpoint(4.2). The registration process provides the possibility of specifying which permission we require (figure 7.3).

Permission Requests:

Read Health Information  Allows you to read a user's health and fitness data

** this permission is not necessary for reading data from this account*

Please explain why you want this permission. Specifically, if you are storing this data, how you plan on abiding by the [API Policy](#).

Edit Health Information  Allows you to edit a user's health and fitness data

** this permission is not necessary for editing data for this account*

Please explain why you want this permission.


Retain Health Information  Allows you to retain a user's health and fitness data if the user disconnects your app

Figure 7.3: During the registration process we can specify which permissions we need.

Most of pmSys' data is stored in a MongoDB, and the mapping of the Runkeeper ids to the correct pmSys ids can be accomplished by making new tables describing the service and authentication in the MongoDB as shown in figure 7.4.

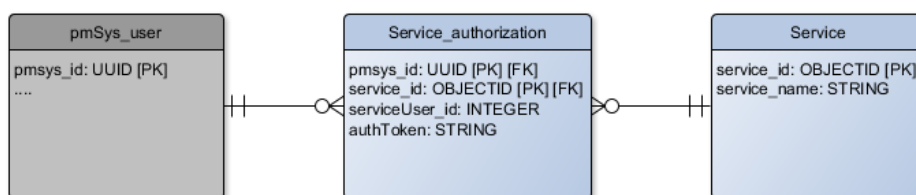


Figure 7.4: Logical model showing a possible solution of storing the connection between a pmSys user and the third-party service used. New tables are colored blue.

```
[
  {
    service_id: "507f191e810c19729de860ea",
    service_name: "Runkeeper"
  }
]
```

Figure 7.5: Example of a data collection containing a service.

```
[
  {
    pmsys_id: "dd6ccd66-25d2-11e7-93ae-92361f002671",
    service_id: "507f191e810c19729de860ea"
    serviceUser_id: "1234567890",
    authToken: "AbCdEf123456ccb00AJ3"
  }
]
```

Figure 7.6: Example of a data collection containing a service_authorization.

These tables are modeled with the thought of pmSys and Shimmer being able to connect with several third-parties other than Runkeeper in the future. The service table has attributes describing the third-party with an id and the name (Runkeeper, Fitbit etc.). The service_authorization table connects a service and a pmSys user together, with attributes that describes this connection. The primary key in this table is the pmsys_id and service_id attributes, as we want to avoid having a user with several connections to the same service. This table also contains the pmSys user's external serviceUser_id (Runkeeper id, Fitbit id etc.) and the corresponding bearer token. The reason for storing the bearer token is so that we are able to make requests to the Health Graph API on behalf of the users.

The system can store and use the bearer tokens since they never expire, unless the user manually disconnects the pmSys application via the settings page on the Runkeeper website. Health Graph have also stated that they intend on keeping it this way. Since this is the case, pmSys can use the user's bearer token when supplying the pmSys application client id to access and request each of their data. If a request for /user is sent (the initial entry point for any given user's account data), the Health Graph API will respond with the /user resource information corresponding to the specified bearer token. The value we are interested in is the Runkeeper user id. The program flow of how to connect Runkeeper and pmSys is described in the sequence diagram in figure 7.7.

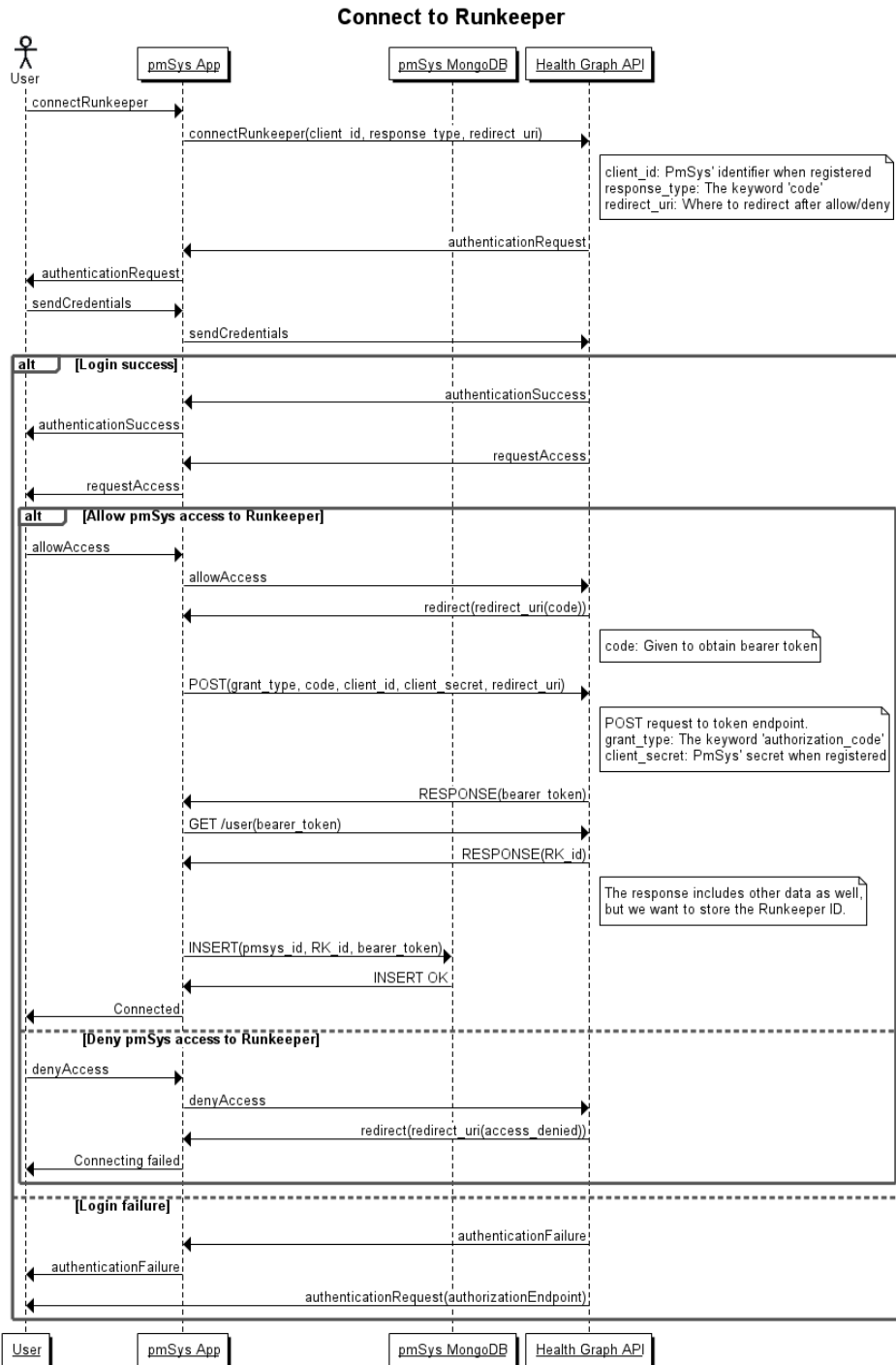


Figure 7.7: Sequence diagram describing the flow when a player connects his pmSys account to his Runkeeper account.

To make the connection process easier for the pmSys users, we wanted the players to be able to do it themselves. In our mockup, the players can login to the pmSys mobile app and connect to Runkeeper with their phones as shown in figure 7.8.

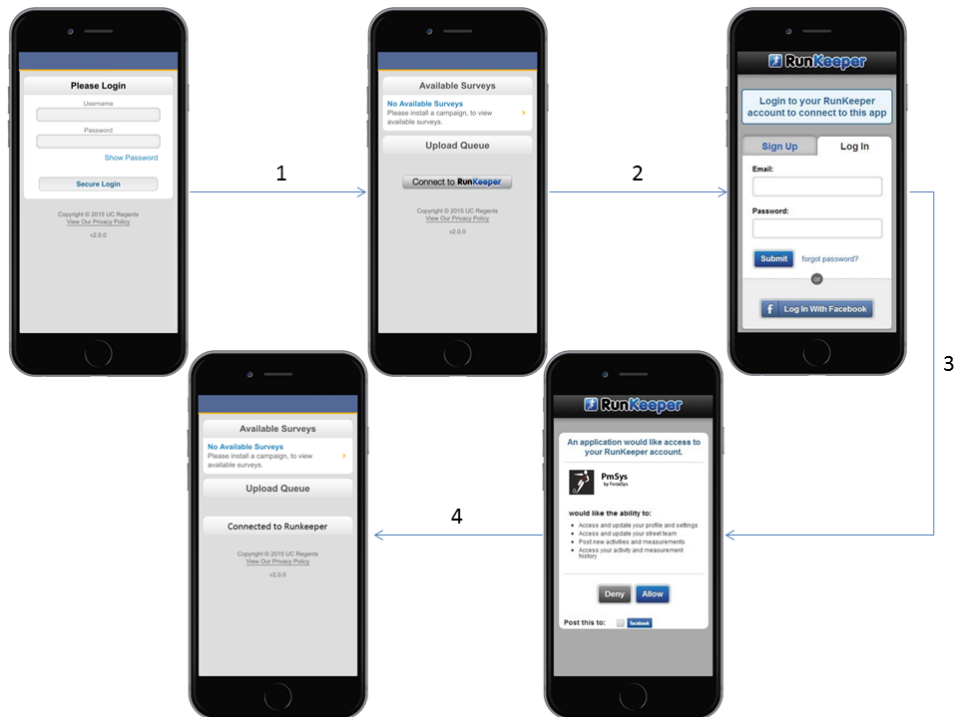


Figure 7.8: Possible solution of how the players can connect to Runkeeper with their phones. A more detailed description is given below.

1. The user must first log in and get authorized in the pmSys mobile application they use to fill out the questionnaires after training sessions. If successfully logged in, it is possible to later use the information about the logged in pmSys user when connecting it to their Runkeeper user. When logged in, the user is redirected to the home screen of the application with the list of available surveys. Below this list is the connect to Runkeeper option that the user has to press.
2. After clicking the connect button the user is prompted to either create or log in to their Runkeeper account. (If the Runkeeper app is already installed on the device and the user is logged in, this step is skipped due to single sign-on (SSO)).
3. After a successful login to Runkeeper the user gets a warning that an application (pmSys) wants to access his Runkeeper account. This has to be allowed to make the connection.
4. After hitting allow, the Health Graph API will redirect the user to the `redirect_uri` provided upon the initial application registration, with a one-time authorization code used to obtain the user's bearer token. PmSys will then make a POST request to the token endpoint of the API, supplying the `grant_type`, the authorization code received, `client_id`, `client_secret` and `redirect_uri`. The response of this request will include the parameter `'access_token'`, which is the bearer token

string which will be stored in pmSys' MongoDB and used to make requests to this user account. A GET request is made to the /user resource with that bearer token, and we'll get a response which includes the Runkeeper user id. A row with this connection is inserted in the MongoDB, and the established connection is shown as "Connected to Runkeeper" in the home screen of the application.

7.4 Fetch data from Runkeeper

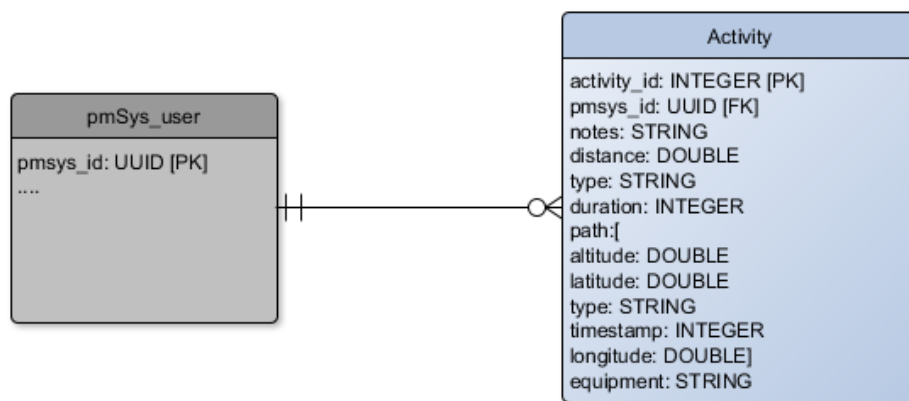


Figure 7.9: Logical model showing a possible solution of how to store the GPS data. The new table is colored blue. It illustrates a one-to-many relationship, stating that an activity belongs to a singular pmSys user, and that a pmSys user may have many activities.

```

{
  activity_id: "940568861",
  pmSysId: "dd6ccd66-25d2-11e7-93ae-92361f002671",
  notes: "First training in pre-season"
  distance: [{
    value: "59.60",
    unit: "m"
  }],
  type: "Running",
  duration: 8,
  path: [{
    altitude: [{
      value: "7.3",
      unit: "m"
    }],
    latitude: "42.31262",
    type: "start",
    timestamp: "0",
    longitude: "-70.951823"
  }, {
    altitude: [{
      value: "7.3",
      unit: "m"
    }],
    latitude: "42.312302",
    type: "end",
    timestamp: "8",
    longitude: "-70.952552"
  }],
  equipment: "Running shoes"
}

```

Figure 7.10: An example of a data collection containing an activity.

The GPS data set of each activity is not presented in the Health Graph's activity feed. To retrieve the path/GPS data from each activity, we have to crawl the Health Graph JSON REST API. That means we firstly have to fetch all the id's of the activities, to eventually be able to access the coordinates data set. A call to Health Graph's `FitnessActivityFeed` endpoint will respond with a JSON list of activities including their uri with the id's - which is one of the values we need to retrieve:

Call parameters

URL: /fitnessActivities

Method: GET

Headers:

- Host: api.runkeeper.com
- Authorization: Bearer xxxxxxxxxxxxxxxxxxxx
- Accept: application/vnd.com.runkeeper.FitnessActivityFeed+json

Success response

The activity list is returned.

HTTP code:

200 OK

Content (application/json):

```
{
  "size": 1,
  "items": [
    {
      "duration": 8,
      "start_time": "Tue, 28 Feb 2017 00:00:00",
      "total_calories": 3,
      "tracking_mode": "outdoor",
      "total_distance": 69.598770321565,
      "entry_mode": "API",
      "has_path": true,
      "source": "Developer's Console",
      "type": "Running",
      "uri": "/fitnessActivities/940568831"
    }
  ]
}
```

Error response

The command failed. The reason is provided in the content.

HTTP code:

500 Internal Server Error or
401 Unauthorized (wrong bearer token)

Given a successful response, the content will be a list of items. In the above example there is one singular activity in the list, with the uri being "/fitnessActivities/xxxx". This represents the activity id which we need to use when making a GET request to retrieve its path (GPS data):

Call parameters

URL: /fitnessActivities/{activity_id}

Method: GET

Headers:

- **Host:** api.runkeeper.com
- **Authorization:** Bearer xxxxxxxxxxxxxxxxxxxx
- **Accept:** application/vnd.com.runkeeper.FitnessActivity+json

URL parameters:

- **activity_id** (required): The id of the activity that is being accessed.

Success response

The activity details, including path is returned.

HTTP code:

200 OK

Response body 1/2(application/json):

```

{
  "next": "/nextFitnessActivity/xxxx/yyyy",
  "notes": "Test2",
  "distance": [{
    "distance":0,
    "timestamp":0
  },{
    "distance":69.5955835192464,
    "timestamp":8
  }],
  "activity":
    "https://runkeeper.com/user/xxxx/activity/yyyy",
  "share_map": "Everyone",
  "entry_mode": "API",
  "source": "Developer's Console",
  "nearest_nutrition":
    "/nearestMeasurement/NUTRITION/xxxx/yyyy",
  "type": "Running",
  "nearest_teammate_sleep": [
    "/nearestMeasurement/SLEEP/xxxx/yyyy",
    "/nearestMeasurement/SLEEP/xxxx/yyyy",
    "/nearestMeasurement/SLEEP/xxxx/yyyy"
  ],
  "userID": 54903982,
  "nearest_teammate_background_activities": [
    "/nearestMeasurement/BACKGROUND_ACTIVITY/xxxx/yyyy",
    "/nearestMeasurement/BACKGROUND_ACTIVITY/xxxx/yyyy",
    "/nearestMeasurement/BACKGROUND_ACTIVITY/xxxx/yyyy"
  ],
  "duration": 8,
  "climb": 0,
  "path": [{
    "altitude":8,
    "latitude":42.31262,
    "type":"start",
    "timestamp":0,
    "longitude":-70.951823
  },{
    "altitude":8,
    "latitude":42.312302,
    "type":"end",
    "timestamp":8,
    "longitude":-70.952552
  }],
  "nearest_teammate_nutrition": [
    "/nearestMeasurement/NUTRITION/xxxx/yyyy",
    "/nearestMeasurement/NUTRITION/xxxx/yyyy",
    "/nearestMeasurement/NUTRITION/xxxx/yyyy"
  ],

```

```

"nearest_teammate_diabetes": [
  "/nearestMeasurement/DIABETES/xxxx/yyyy",
  "/nearestMeasurement/DIABETES/xxxx/yyyy",
  "/nearestMeasurement/DIABETES/xxxx/yyyy"
],
"total_distance": 69.598770321565,
"share": "Everyone",
"nearest_general_measurement":
  "/nearestMeasurement/GENERAL/xxxx/yyyy0",
"nearest_diabetes":
  "/nearestMeasurement/DIABETES/xxxx/yyyy",
"nearest_weight":
  "/nearestMeasurement/WEIGHT/xxxx/yyyy",
"images": [],
"comments": "/fitnessActivities/xxxx/comments",
"nearest_teammate_weight": [
  "/nearestMeasurement/WEIGHT/xxxx/yyyy",
  "/nearestMeasurement/WEIGHT/xxxx/yyyy",
  "/nearestMeasurement/WEIGHT/xxxx/yyyy"
],
"previous": "/prevFitnessActivity/xxxx/yyyy",
"total_calories": 3,
"nearest_strength_training_activity":
  "/nearestStrengthTrainingActivity/xxxxx/yyyy",
"nearest_teammate_strength_training_activities": [
  "/nearestStrengthTrainingActivity/xxxx/yyyy",
  "/nearestStrengthTrainingActivity/xxxx/yyyy",
  "/nearestStrengthTrainingActivity/xxxx/yyyy"
],
"equipment": "None",
"heart_rate": [],
"nearest_sleep":
  "/nearestMeasurement/SLEEP/xxxx/yyyy",
"calories": [],
"uri": "/fitnessActivities/xxxx",
"start_time": "Tue, 28 Feb 2017 00:00:00",
"nearest_background_activity":
  "/nearestMeasurement/BACKGROUND_ACTIVITY/xxxx/yyyy",
"nearest_teammate_general_measurements": [
  "/nearestMeasurement/GENERAL/xxxx/yyyy",
  "/nearestMeasurement/GENERAL/xxxx/yyyy",
  "/nearestMeasurement/GENERAL/xxxx/yyyy"
],
"tracking_mode": "outdoor",
"is_live": false,
"nearest_teammate_fitness_activities": [
  "/nearestFitnessActivity/xxxx/yyyy",
  "/nearestFitnessActivity/xxxx/yyyy",
  "/nearestFitnessActivity/xxxx/yyyy"
]
}

```

This is the unmodified response provided from the Health Graph API data endpoint. As one can see there is much data provided which is not relative for the purpose of pmSys's health data monitoring system.

Error response

The command failed. The reason is provided in the content.

HTTP code:

404 Not Found or

403 Forbidden (wrong bearer token)

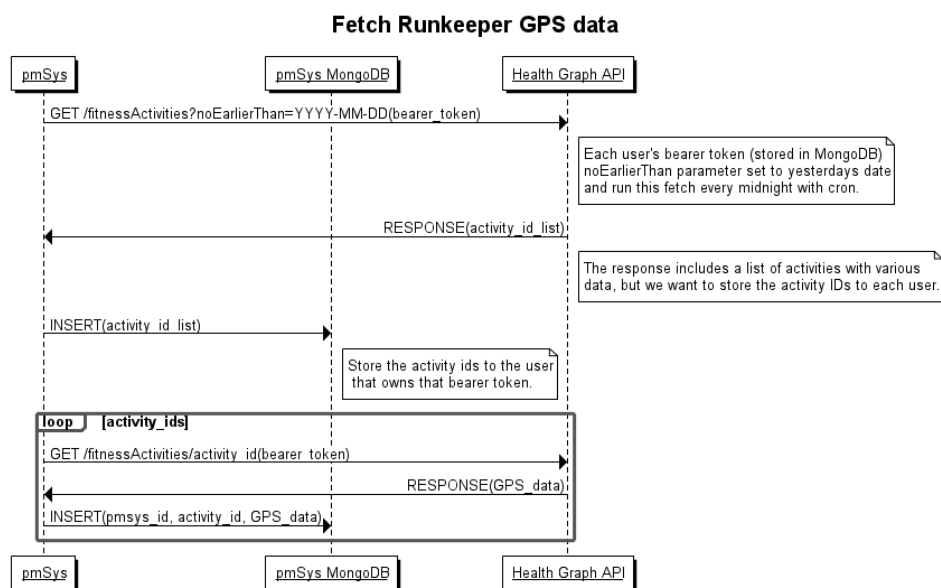


Figure 7.11: A sequence diagram describing the steps and flow of the daily fetch (at midnight) using cron (see section 7.4.1).

7.4.1 Scheduling requests

When a player fills out a questionnaire after a training session and submits it, he is also going to push his Runkeeper data to pmSys. Since the player has to be logged in on the mobile app with his user, the system already knows his Runkeeper id and the corresponding bearer token, provided that he has connected his pmSys account to Runkeeper 7.3. The bigger benefits of making the process of gathering data automated are keeping the database up-to-date and making this part of the system self-sustaining. It could happen that a player forgets to submit the questionnaire several days in a row, making the Runkeeper data in pmSys' database out-of-date. We want pmSys to gather and store the Runkeeper activities even when the players haven't pushed the data through the app. This can be achieved on the server through a UNIX and Linux utility called Cron [50].

Cron is used to execute selected tasks in the background at specified times

and intervals. Cron keeps a file called a crontab, which contains a list of all commands and scripts to be ran at individual given times. The Cron daemon reads the crontab and its scripts and run-times, and executes them in the background. A crontab may be specified for each user of the system, meaning that we can dedicate a crontab solely for the root user with scripts that has restricted read/write permissions and follows the principle of least privilege for security purposes. The general form of a crontab entry looks like this:

```
* * * * * /bin/execute/this/script.sh
```

Each asterisk from left to right corresponds to minute, hour, day of the month, month and day in the week respectively (See table 7.2).

No. *	Description	Values
1	Minute	0 - 59
2	Hour	0 - 23
3	Day of the month	1 - 31
4	Month	1 - 12
5	Day of the week	0 - 6 (Sunday = 0)

Table 7.2: Crontab entry parameters

In the crontab, asterisk is read as "any". For the example above, the file script.sh is ran every minute. There are also special strings and keywords, which defines all five parameters based on the keyword. If you wanted the script to be ran once a day at midnight to avoid high latency when the server workload most likely will be low, you could do:

```
0 0 * * * /bin/execute/this/script.sh

# Or the equivalent by using a keyword:

@midnight /bin/execute/this/script.sh
```

There are several keywords like this in crontab. The table 7.3 shows some of the different keywords usable in the crontab:

Keyword	Equivalent	Description
@reboot	NA	Run once at startup
@yearly	0 0 1 1 *	Run once a year (1st of January)
@monthly	0 0 1 * *	Run once a month (1st of the month)
@weekly	0 0 * * 0	Run once a week (every Sunday)
@daily	0 0 * * *	Run once a day (same as @midnight)
@hourly	0 * * * *	Run once a hour

Table 7.3: In the crontab you can use keywords to describe all parameters.

7.5 Database storage

The data we want to retrieve is initially only GPS coordinates. This data comes from and is stored by a third-party, in this case Runkeeper. PmSys wants to collect this data, and store it in their own database.

7.5.1 Replication

A lot of the new data pmSys want to store in their database will be a replication of data from Runkeeper modified to comply with the OMH schemas. We will refer to Runkeeper and pmSys as primary database and destination database respectively. One of the major challenges with replication of data occurs when data is changed at the primary database (Runkeeper) and not in the destination database (pmSys). The system needs a way of detecting these possible changes.

An event might occur that requires editing of a specific activity in the primary database. This activity may already have been pushed and stored in the destination database and made available in the system.

In the Health Graph API it is possible to include query parameters which describes the last modification timestamp, meaning the last time an activity has been modified. These query parameters may define a time scope and takes dates in a YYYY-MM-DD format and, per the HTTP specification, must be specified in Greenwich Mean Time (GMT). The request parameters available are the noEarlierThan, noLaterThan, modifiedNoEarlierThan, and modifiedNoLaterThan parameters. For detecting modifications made to the activities we can utilize one of the last two, or a combination. A request made with these parameters can look like this:

```
GET /fitnessActivities?modifiedNoEarlierThan=2017-03-01
&modifiedNoLaterThan=2017-03-03 HTTP/1.1
```

```
Host: api.runkeeper.com

Authorization: Bearer xxxxxxxxxxxxxxxxxxxx

Accept: application/vnd.com.runkeeper.FitnessActivityFeed+json
```

This request will give a response of the list of activities modified, or created, within the time scope of 1st - 3rd of March. The response can look like:

```
{
  "size": 2,
  "items": [
    {
      "type": "Running",
      "start_time": "Tue, 1 Mar 2017 08:00:00",
      "total_distance": 70,
      "duration": 10,
      "source": "RunKeeper",
      "entry_mode": "API",
      "has_map": "true",
      "uri": "/activities/100"
    },
    {
      "type": "Running",
      "start_time": "Thu, 20 Feb 2017 08:00:00",
      "total_distance": 70,
      "duration": 10,
      "source": "RunKeeper",
      "entry_mode": "Web",
      "has_map": "true",
      "uri": "/activities/40"
    }
  ]
}
```

Figure 7.12: A JSON response when using the `modifiedNoEarlierThan` and `modifiedNoLaterThan` parameters to create a time scope. The response is a list of all activities modified between the two dates.

The list includes an uri with the id of the activity, which we can use to update the activity in pmSys. The first item, with activity id 100, is an example of an activity created within the specified time frame. The activity with id 40, and the second item in the response list, is an activity created outside of the scope, which means that this activity has been modified subsequent to the creation date. In the MongoDB, we can then

utilize the update method defined as `db.collection.update(query, update, options)[51]`, with the optional `upsert` parameter set to `true`. When set to `true`, this parameter will make sure that a new activity is created if the activity id in the the query criteria doesn't match any of the id's in the pmSys database.

7.5.2 Concurrency in MongoDB

As explained in the MongoDB chapter 2.7, each record of data is stored in a single and unique document, but what happens if multiple CRUD operations are being performed at the same time on a single document or collection concurrently? There are two primary concurrency control measures, pessimistic and optimistic, to prevent clients to modify the same record simultaneously, these controls are in place to ensure consistency and correctness of data. Pessimistic concurrency control is often utilized in systems with locks, and is based on the principle that the worst will happen. This control locks a document immediately when any CRUD operation is done on it, making it impossible for multiple updates to happen. The lock will be placed no matter what, even though operations might not conflict at all.

Optimistic concurrency control assumes that conflict are very rare, though possible. The system looks indications that a record has had multiple CRUD operations done to it simultaneously, if that is true, one user's updates are discarded and informed about the issue. When a user or the server is trying to read or update a document, a version number is attached to it, when the transaction is committed the system validates the version number. If the number being returned with the commit is equal to the original number, the system can write without any issues. If the numbers are unequal it means that someone has updated the data during another transaction, the transaction coming in last is then discarded.

Pessimistic and optimistic concurrency control

Pessimistic concurrency control generates overhead for each CRUD operation done in a database, even when only a single user is trying to access a document. The system also checks each time if the requested document is already locked by another client or connection. The traits of pessimistic locking makes it useful in systems where the chances of updates happening on the same records is relatively high, and the documents are small with frequent updates. Optimistic concurrency control is more useful when the chances of conflict is minor. Generally used when there are many record, few users and mostly read and create operations. In most scenarios optimistic locking offers higher performance than pessimistic, but is more likely to cause concurrency issues if used in the wrong environment.

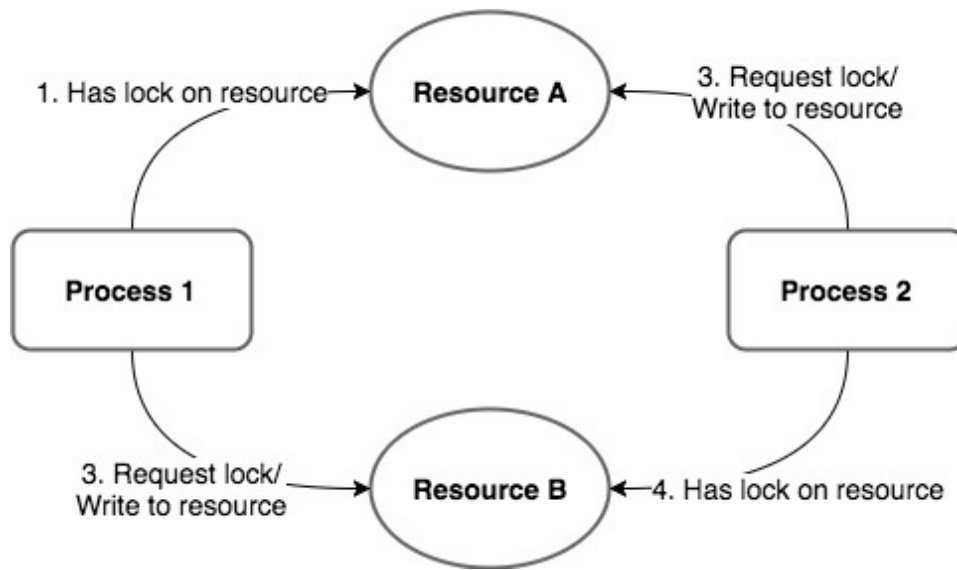


Figure 7.13: Multiple processes creating a deadlock by requesting and locking same resources.

When working with objective data coming from wearable devices and such, a deadlock should rarely happen. The figure 7.13 depicts a basic deadlock. Process 1 has locked resource A and is holding it for an update, and at the same time is requesting resource B for dependent update. But resource B has already been locked by process 2, which again is requesting a lock and write operation for resource A. This will create a loop which is called a deadlock, where no operation will ever finish as they did not know about other processes holding a resource before starting an operation, and both operation has to happen concurrently.

The reason is that the data belong to a unique user, which also is the only one able to update the data. A create or update operation will not create any concurrency or deadlock issues as the locks are not placed on a collection of data, but rather one record. And one record has it's own unique user identifier, meaning that only a read operation could be happening at the same time as a update operation. And in MongoDB, only compatible operations are given access at the same time. When comparing pessimistic and optimistic concurrency control in regards to implementing objective data in pmSys, the theory behind optimistic locking matches the characteristic of the activity database in pmSys making it the more reasonable choice.

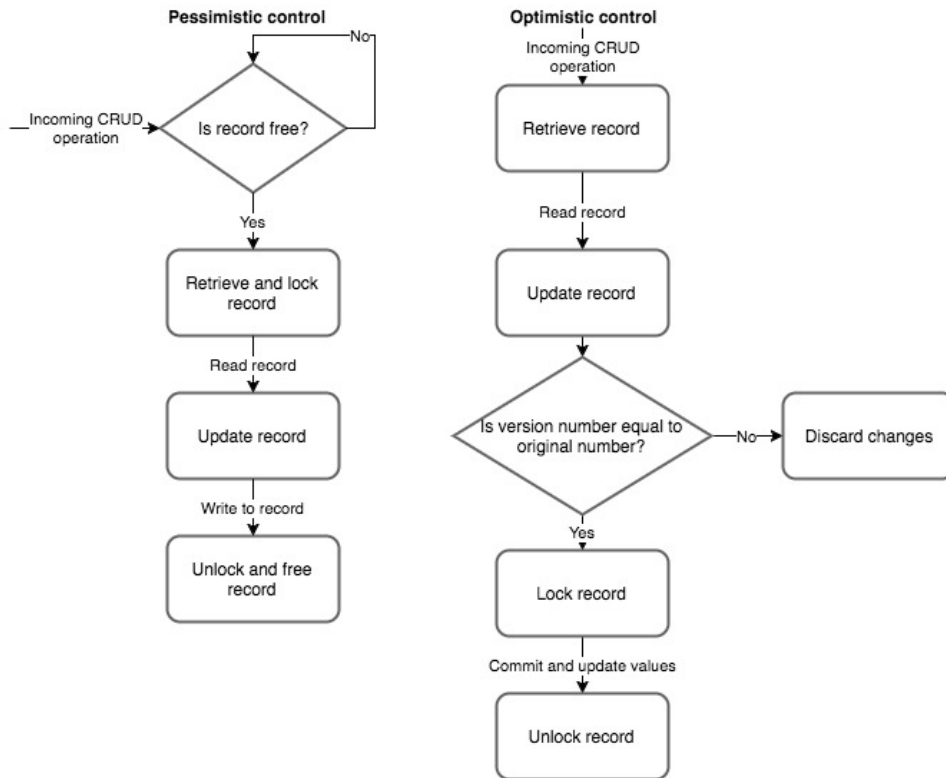


Figure 7.14: Flowchart representing the flow of a CRUD operation being performed on a database record, the lock occurs much earlier in the pessimistic approach.

To enhance concurrency in the database, MongoDB supports sharding. Sharding is the ability to partition collections over multiple database instances, allowing mongos processes to perform otherwise conflicting operations concurrently. Locks are only applied to the shards, not the whole cluster of shards, making the instances and operations independent to one another. Applying sharding to a database makes it horizontally scalable which may also improve the time to crawl the database. The figure 7.17 is a sharded version of a Team collection, making it possible to read and write in all three collections concurrently without risking any conflict, as opposed to multiple operations done on the Team table.

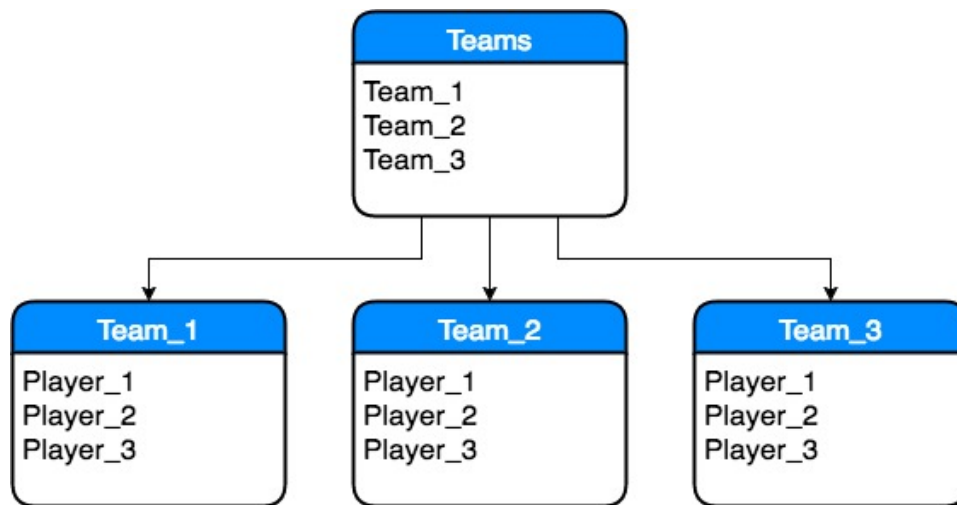


Figure 7.15: Team collection sharded into the different teams.

WiredTiger storage engine

The WiredTiger storage engine is the default engine shipped with MongoDB 3.0 and up, and is what lies in the bottom of the pmSys MongoDB. WiredTiger uses optimistic concurrency control on the document-level and results in clients being able to modify documents in the same collection simultaneously. WiredTiger utilizes intent lock only on collection, global and database level. Intent locks are only acquired when there is an indication to modify a document, but does not conflict with read operation as it is compatible with with write operations. As WiredTiger uses MultiVersion Concurrency Control to create snapshots of the data at the start of a transaction to create a consistent view of the data, the intent lock is only acquired if the document is unmodified by concurrent transactions.

These snapshots can also function as checkpoints for the database, and the checkpoints also ensures that the data is consistent up to the last created snapshot/checkpoint. If a transaction has gone wrong, creating inconsistent data in a collection of documents, the last valid checkpoint can be used as a point of recovery. When a checkpoints becomes permanent as the latest valid point, it is updated in WiredTiger's own metadata schema, making it accessible to other transactions and freeing the collections. Creating checkpoints for your database are all atomic transaction, one of the ACID transaction properties: Atomicity, Consistency, Isolation, Durability. Atomic transactions are multiple steps of database operations where either all occur or nothing, partial checkpoints or data transaction can be more of a problem in a database than discarding a conflicting transaction.

7.5.3 Big data

The term big data is something that have gotten a lot of attention in the past years, and can be the chokepoint of many data structures if no anticipated

and handled correctly. The age of digitalization has pressed forward unlimited possibilities for connecting almost everything to the internet. Big data in general envelops the need for new methods to process and analyze data sets which are magnitudes larger than the average company database, and the complexity of the data sets renders normal processing methods void. For example, Google has developed new types of databases which can spread and store massive amount of data over thousand of machines and can process trillions of record in mere seconds [52].

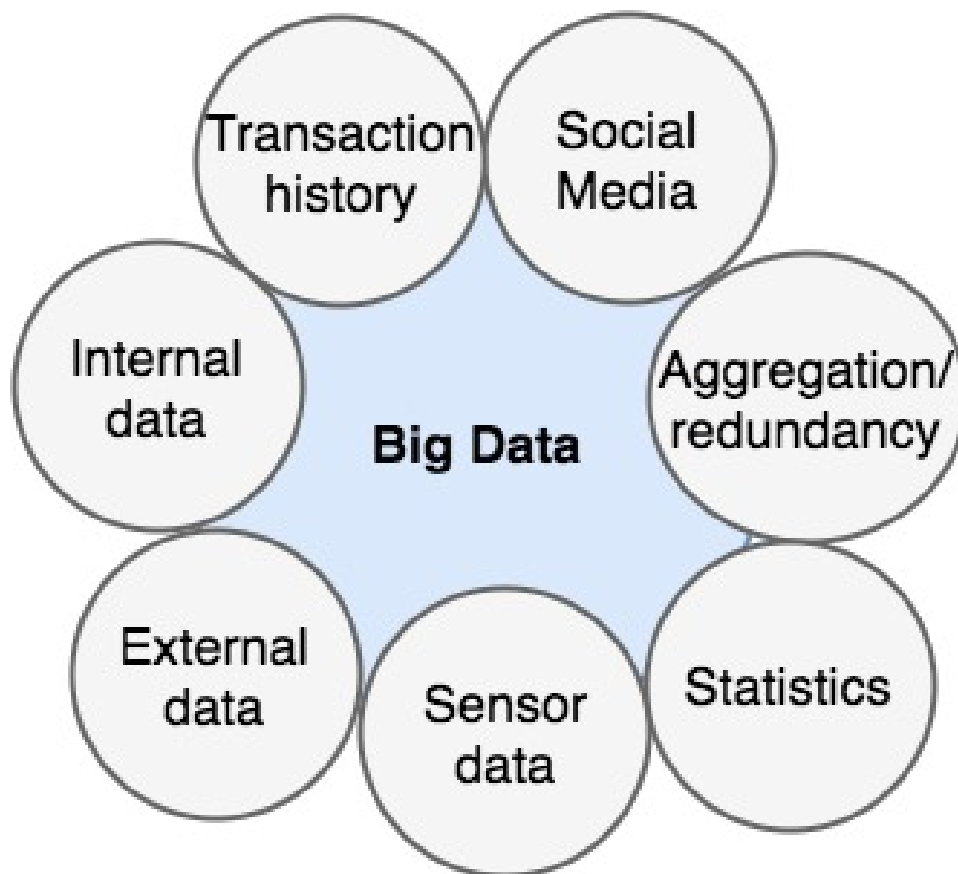


Figure 7.16: Big data envelops all possible data collected included the aggregated data.

One concern with big data is mechanisms of searching, storing, aggregation and visualization of data which associate with each other by keys and values. The data in question for this thesis is objective data, and the input is not necessary sorted or processed in a way that is beneficial for the system, this is why it is imperative to create data models where correlations between data sets are handled in a constructive manner. For example, the correlation between multiple activity data sets can provide an indication to when a player is at peak performance during the day or week. The concept of collecting data is only relevant if the data can be processed and returned to the user within a suitable timeframe.

One challenge with keeping inside the suitable timeframe is when the old hardware is not adequate for processing the objective data which has accumulated into a big data construct. What has taken seconds to process might take minutes, hours or even days to process if the growing data set has no scalability or adaptation for new hard- or software. There is existing tools for processing massive data sets, such as Hadoop [53] and Spark [54], which also can be run in combination. These tools are software libraries which are created for the purpose of processing vast data sets across clusters of servers and storage units, and offers scalability for single or multiple units. These tools draws their capability by utilizing the potential of sharding 7.17, processing data on each shard which makes up a cluster. One strategic consideration is whether to put the data on disk, or in-memory. Data only accessible on disk is much cheaper, but the drawback is the limitation of processing speed. By loading the data in-memory, on the RAM, the queries that are performed can return data thousand of times faster compared to data on disk, but is much more costly.

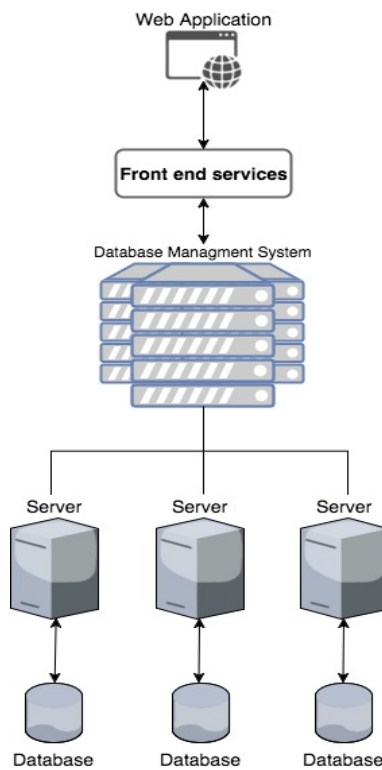


Figure 7.17: Distributed database system with data physically stored across multiple sites.

Data mining and patterns

The task of processing large data sets is often referred to as data mining. This entails sorting through the data and identifying patterns and correlation between data that is too vague to comprehend without the right tools. Association rules are created by analyzing if-then patterns,

and then applying parameters like frequency for number of occurrences, and confidence for the amount the if-then patterns are accurate. Sequence parameters is used to predict events which leads to another event, this can be applied in pmSys to prevent fatigue or overtraining. Data mining can also be used to determine inaccuracy and errors in the data models or data input.

7.5.4 Database optimization

In a database that will contain magnitudes of historical data, and has massive amounts of inputs there are some strategic choices to consider regarding design and modeling. You have two very important factors to consider when implementing database structure, normalization and denormalization. This is a MongoDB aspect.

Normalization

This is the process to efficiently organize your data through primary and foreign keys which creates correlation between data sets or database entities. One of the main goals of normalizing data is to keep redundancy as low as possible, only storing data once in a single or multiple documents. Another goal is to keep data which make sense in the same document, in example, weather data has no place in a document depicting a person entity, but the person's name does. Keeping these goals in mind can greatly increase the logic in a database, and decrease the amount of space used for all documents and collections. These conditions can also improve the CRUD operations create, update and delete. Normalization is achieved by following the guidelines of normal forms [55], and is relative to the final construct of logical complexity needed for the final result.

Denormalization

This is the process of optimizing the last CRUD operation read. By adding redundant data(replica), and creating views for grouping of data the read operation can be increased for normalized databases with a complex logical structure. Views are read only groupings, aggregation or joins of data which otherwise could have slow query performance in a normalized database. The views has a drawback that all the replicated data stored increases the space absorbed by the database. Denormalization in general reduce the write, update and delete operation because of potential aggregations which are done on a create or update to create a fields which contains data which are valuable for fast access. Deletes are in similar fashion slower for the reason that more fields has to be removed.

In NoSQL databases, in particular MongoDB, denormalization is achieved by embedding documents with data that otherwise could be referenced with one or multiple keys. One drawback with denormalization is consistency. In a normalized table, data which are referenced would know about

a change in data elsewhere. In example, if an author entity is referenced in a book entity, and the author data changed, the data would still be consistent as the data is joined on query time. If the data author data is embedded in the same document as the book entity it would be a denormalization of two documents, but it would cause inconsistency if the original author entity is updated as the embedded data would not change. The consistency has to be handled in some way, shape ,or form, but the read and update operations required for query could potentially be process heavy and should be performed during system downtimes, as it will cause reduced response time for users.

7.6 Summary

The PmSys application is built upon the Ohmage platform (section 2.4). One of Ohmage's components is the Mobile data collection apps. This component is divided into two groups; self-reporting applications and passive data collection applications, providing subjective and objective data respectively. The current PmSys uses only an instance of a self-reporting mobile application (pmSys-app), that provides easy manipulative subjective data. With the implementation of Shimmer, we introduce an instance of the passive data collection applications, being able to supplement the subjective data with objective context or vice versa. The objective data provides interesting and useful data in itself, and combining this with the subjective data will give a widely detailed picture of the players general fitness, providing the coaching team more information about the players and their behavior during training sessions.

The proof of concept is meant as useful preparatory work for future implementation of a passive data collection application in pmSys. We have tried to cover the implementation in general and as a whole, presenting the relevant obstacles that might occur, and possible solution to solve these challenges. The prototype is specifically targeting the implementation with Runkeeper as the third-party data provider. In spite of this, we have based our concept on the possibility of using other data providers, having to tweak as little logic as possible.

We believe that by utilizing processing and aggregation tools, a big data construct might be preferable to transition into. By distributing the data across multiple servers, a distributed database, can partly solve problems that might occur with slow write and update operations for MongoDB documents. PmSys fulfills multiple of the requirements, as there is already an underlying structure ready for this regarding the storage units in the pmSys backend. One important strategic choice when constructing the new logic for the activity storage unit will be the general complexity and database optimization 7.5.4. This is in direct correlation with what CRUD operations are important, if it will be a database which is skewed toward many write/updates, or if it is important with immediate consistency because of

an abundance of read operations.

Chapter 8

Conclusion

8.1 Summary

Our problem definition in section 1.2 stated that we wanted to explore the possibility of introducing objective data to add more meaning to the subjective data already captured by the self-reporting pmSys-app. We also wanted to find an efficient way of gathering and storing this data. We have done so with a concrete example of a data provider in Runkeeper. We were going to describe how a passive data collection app in the form of Shimmer could be used to collect the objective data on the behalf of pmSys.

After researching possible data provider with focus on the supported APIs in Shimmer, we arrived at the conclusion that Runkeeper was the best provider for illustrating a solution of answering our research question. Using Runkeeper as the data provider for the purpose of this research thesis required us to explore their API, Health Graph, and its end points. Shimmer was going to be responsible for requesting and handling the responses on behalf of pmSys due to the desire to contribute to the OMH initiative (see section 2.5). Therefore, choosing a third-party already supported by Shimmer made the solution more applicable and required less tweaking of the current Shimmer application. The more tweaking done to Shimmer, the less likely it would be to uphold the standards set by OMH.

Researching OMH's cause and vision provided useful insight and gave us motivation on contributing as much as we could with regards to our thesis. This lead to us designing a common data schema for GPS data to be used by Shimmer to normalize the responses to the given format. We based our decisions on the principles described by OMH (see section 3.2.1), focusing on the granularity of the schema, and to not be over-complex when designing it.

8.2 Main Contributions

In this thesis we have shown how Open mHealth's application, Shimmer, can be implemented in pmSys to supplement the system with objective

data. We have described all the different components, and how they can work together to create an environment where objective data from third-party providers can be fetched, processed and stored to complement the existing captured data set. Our research has led to a solution that we found most suitable for capturing data and storing it for further use, as well as making it expandable.

To reach this goal we had to work with Shimmer and expand the application to make it able to request from the correct endpoints the correct objective data that is required to perform a detailed analysis of players from their spatiotemporal patterns. The main data we wanted to retrieve was the GPS coordinates to make a detailed map of run patterns, and provide the opportunity to compare physical perception against actual effort and discover deviations. Included in Shimmer is also the possibility of getting measures like calories burned, body weight and heart rate which can be applied due to the expandability to provide an even more detailed picture of the players total health status.

The objective data is also normalized to a common data schema for GPS. This was created from scratch by following strict principles composed by Open mHealth. The advantage of creating a common format for this data in pmSys is that if teams want to use other third-party data providers to track movement in the future, Shimmer can be used to normalize the data and give it the same contextual meaning in the same format. This also removes the need of creating other logic for aggregating these data sets in the future.

All this data can be joined through logic and be returned to coaches or other administrators for a visual representation which is human readable. This includes numbers, graphs, charts and more. A good example of how pmSys can be used in a scenario is after a training session with the whole team. All the players start their application to register the start of an activity session. During the session the hardware they use captures all data that is possible for the third-party provider. At the end of training, all the players mark the end of the activity session as well. The players then fills out the self-reporting surveys answering how they feel after the session. Post-workout, both the subjective and objective data sets are presented on the pmSys-trainer website. The coaches can then study how the perceived workload correlates to the actual workload. This can provide useful insight for trainers, making it possible to detect fatigue and overtraining before a possible injury occurs.

8.3 Future work

During our research we have covered what is essential to create an environment that can support the inputs from devices tracking objective user data. However there are some cases which will be dependent on how the architecture of a final solution is constructed. In this section we present

some ideas and features that can be beneficial in the future.

8.3.1 PmSys frontend

Visualize GPS data

The pmSys frontend has not been the focus of this thesis, but a visual representation of the data is absolutely necessary for a complete monitoring solution. At this time pmSys uses the D3.js [56] library to create graphs, charts and other data representations. This library provides almost all the necessary figures for objective data, with the exception of maps. For map support there are libraries like Mapbox [57] or Google Maps [58] that might be fitting to use.

8.3.2 PmSys backend

Capture data from other third-parties

Capture data from other third-parties that supports Shimmer in work for the future. This will require different logic around fetching, as the different third-parties handles authentication differently. The other third-parties would also have to normalize the incoming data to the common data schema we've created. That way, you can store data from multiple different inputs and get the same format.

API calls

The API calls does not support clean GET requests for all detailed activity data sets. Resulting in a more demanding workload for the system. If a GET request for all detailed data were to be introduced to the API, requesting process can be simplified.

Implementation of other APIs

Implementation of other APIs like Fitbit, Google Fit, iHealth, Jawbone, Misfit, and Withings which are supported in Shimmer could provide variety in the technology used by football team. Different systems might appeal more to different teams, but given that the data is normalized to the common data schema, this won't cause many problems. The support can be extended to support additional APIs.

Modify stored activities

Modify stored activities in the activity MongoDB could provide useful if there appears to be some sort of error or accident during activity recording. This can be handled by players modifying their own data in the Runkeeper application, given that a crontab job can be run at midnight fetching new or modified data each day. Nevertheless, it might add useful functionality to be able to do a PUT request to the right API for a trainer or administrator to have access and supplementing the data with their own notes.

Deleting stored activities

Deleting stored activities in pmSys if there is data not supposed to be there, i.e a player forgot to stop their monitoring or accidentally started a recording. This can't be handled by deleting the activity in the Runkeeper application, because there is no indication given through a GET request that something is deleted. By running a crontab job at a fixed time it can determine if there are activity id's in the pmSys database which does not exist in Runkeeper, and act accordingly.

Processing data

Processing data through Shimmer is not currently provided, and therefore requires another solution. OMH has plans to develop a processing unit in the future [59], so a potential dedicated processing module can be excluded from the backend.

A big data solution

A big data solution might be necessary in the future since there might be a great deal of data produced by a continuous stream of objective data in the future.

On-demand get requests

On-demand get requests might provide useful functionality. Trainers from the different teams should be able to login to the pmSys-trainer web portal and request data from the activities performed by their players on-demand. This could be solved by a simple button press in the web portal, following a combination of same logic as the connect function and the cron request.

Logging

Logging of database transactions is an important measure for listing changes to the database and storing it in a stable storage format. If there is an inconsistent state detected, the database management system might want to review the database logs for uncommitted transactions. Then we might want to perform a rollback to undo the changes made by these transactions. MongoDB is at this moment a suboptimal choice for a scalable logging solution. We have not investigated possible solutions regarding the monitoring and logging of database transactions, and consider this future relevant work.

Live GPS tracking

In this specific research thesis, we used Runkeeper to illustrate the implementation of movement data after the activity had finished. Runkeeper also

offers a "plus" subscription called Runkeeper GO. If the desire to implement live tracking of training sessions arise, we recommend investigating this, including the big data section covered in section 7.5.3.

8.3.3 Aggregation of data

When pmSys has stored all GPS coordinates created during a session, aggregating the data becomes a possibility and can be used for statistical analysis. The purpose is to get more information out of the data that has been collected. For instance, calculating the distance in meters and kilometers between two GPS location can be done like this in java:

```

//Calculate the distance in meters between two coordinates
private static double distanceInMeters(double lat1, double lon1
    , double lat2, double lon2)
{
    double longs = lon1 - lon2;
    double dist = Math.sin(degToRad(lat1)) * Math.sin(
degToRad(lat2)) + Math.cos(degToRad(lat1)) * Math.cos(
degToRad(lat2)) * Math.cos(degToRad(longs));
    dist = Math.acos(dist);
    dist = radToDeg(dist);
    dist = dist * 60 * 1.1515;
    dist = dist * 1609.344;
    return (dist);
}

//Calculate the distance in kilometers between two coordinates
private static double distanceInKilometers(double lat1, double
lon1, double lat2, double lon2)
{
    double longs = lon1 - lon2;
    double dist = Math.sin(degToRad(lat1)) * Math.sin(
degToRad(lat2)) + Math.cos(degToRad(lat1)) * Math.cos(
degToRad(lat2)) * Math.cos(degToRad(longs));
    dist = Math.acos(dist);
    dist = radToDeg(dist);
    dist = dist * 60 * 1.1515;
    dist = dist * 1.609344;
    return (dist);
}

//Decimal degrees to radians
private static double degToRad(double deg)
{
    return (deg * Math.PI / 180.0);
}

//Radians to decimal degrees
private static double radToDeg(double rad)
{
    return (rad * 180 / Math.PI);
}

```

When finished calculating the distance in the preferred unit, it is possible to calculate several different types of information. Much can be learned about a player when knowing how much distance he has covered during a session, but the total distance says nothing about the intensity. With the GPS data we can now calculate the average speed between the data points,

in addition to acceleration and deceleration.

Appendix A

Accessing the source code

The source is code divided into shimmer and bash scripts.
Access to the repository is given upon request.

<https://github.com/DanielGJohnsen/thesis-source-code>

Bibliography

- [1] TV2. Langrennsekspertens brannfakkell: – det er mange som tror at norsk fotball er toppidrett, 2013. URL <http://www.tv2.no/a/3996469/>.
- [2] Haavard Wiig. Research project: Load monitoring in football., 2016. URL <http://www.nih.no/en/research/projects/all-projects/load-monitoring-in-football/>.
- [3] Garmin. About gps, 2017. URL www.garmin.com/aboutGPS/.
- [4] The18. The most intimidating soccer stadium in the world, 2016. URL <http://the18.com/news/most-intimidating-soccer-stadium-world>.
- [5] LIVESTRONG. What effects do high altitude have on the body, 2011. URL <http://www.livestrong.com/article/455572-what-effects-do-high-altitudes-have-on-the-body/>.
- [6] Maria J. Brosnan, David T. Martin, Allan G. Hahn, Christopher J. Gore, and John A. Hawley. Impaired interval exercise responses in elite female cyclists at moderate simulated altitude. *Journal of Applied Physiology*, 89(5):1819–1824, 2000.
- [7] CSS insight. Fitness bands and basic smartwatches fuel sales of wearable devices, 2017. URL <http://www.ccsinsight.com/press/company-news/2702-fitness-bands-and-basic-smartwatches-fuel-sales-of-wearable-devices>.
- [8] CSS insight. Wearables momentum continues, 2016. URL <http://www.ccsinsight.com/press/company-news/2516-wearables-momentum-continues>.
- [9] World Health Organization (WHO). mhealth - new horizons for health through mobile technologies, 2011. URL http://www.who.int/goe/publications/goe_mhealth_web.pdf.
- [10] H. Tangmunarunkit, C. K. Hsieh, J. Jenkins, C. Ketcham, J. Selsky, F. Alquaddoomi, D. George, J. Kang, Z. Khalapyan, B. Longstaff, S. Nolen, T. Pham, J. Ooms, N. Ramanathan, and D. Estrin. Ohmage: A general and extensible end-to-end participatory sensing platform. *UCL Computer Science Technical Report*, 2014.
- [11] Ohmage. Ohmage | open data collection, 2016. URL <http://ohmage.org>.

- [12] Open mHealth. About us, 2015. URL <http://www.openmhealth.org/organization/about/>.
- [13] Deborah Estrin and Ida Sim. Open mhealth architecture: An engine for health care innovation. *Science*, 330:759–760, 2010.
- [14] Docker. Docker toolbox, 2017. URL <https://www.docker.com/products/docker-toolbox>.
- [15] MongoDB. Introduction to mongodb, 2017. URL <https://docs.mongodb.com/manual/introduction/>.
- [16] MongoDB, 2016. URL <https://docs.mongodb.com/v3.2/core/databases-and-collections/>.
- [17] OAuth. The oauth 2.0 authorization framework, 2012. URL <https://tools.ietf.org/html/rfc6749>.
- [18] IETF Tools. The oauth 2.0 authorization framework, 2012. URL <https://tools.ietf.org/html/rfc6749#section-1.3.1>.
- [19] Open mHealth. Data provider api library, 2015. URL <http://www.openmhealth.org/documentation/#/data-providers/data-provider-api-library>.
- [20] Fitbit. Activity & exercise logs, 2017. URL <https://dev.fitbit.com/docs/activity/>.
- [21] Google Fit. Fitness data types, 2017. URL <https://developers.google.com/fit/rest/v1/data-types>.
- [22] iHealth. Request for data of activity report, 2017. URL http://sandbox.ihealthlabs.com/dev_documentation_RequestfordataofActivityReport.htm.
- [23] Jawbone. Workouts, 2017. URL <https://jawbone.com/up/developer/endpoints/workouts>.
- [24] Misfit. Api references, 2017. URL https://build.misfit.com/docs/cloudapi/api_references.
- [25] Health Graph. Fitness activities, 2017. URL <https://runkeeper.com/developer/healthgraph/fitness-activities>.
- [26] Withings. Withings api reference, 2017. URL <https://oauth.withings.com/api/doc>.
- [27] IDC. Smartphone os market share, 2016 q3, 2016. URL <http://www.idc.com/promo/smartphone-market-share/os>.
- [28] Athlete Monitoring, 2016. URL <http://www.athletemonitoring.com/>.
- [29] Health Graph. Developer’s console, 2017. URL <http://www.openmhealth.org/app/uploads/2015/05/Data-flow-architecture.jpg>.

- [30] Open mHealth. About shims, 2015. URL <http://www.openmhealth.org/documentation/#!/data-providers/about-shims>.
- [31] Open mHealth. Schema design principles, 2017. URL <http://www.openmhealth.org/documentation/#!/schema-docs/schema-design-principles>.
- [32] UCUM. Commonly used ucum codes for healthcare units, 2017. URL <http://download.hl7.de/documents/ucum/ucumdata.html>.
- [33] Open mHealth. Physical activity, 2017. URL http://www.openmhealth.org/documentation/#!/schema-docs/schema-library/schemas/omh_physical-activity.
- [34] OAuth. The oauth 2.0 authorization framework, 2012. URL <https://tools.ietf.org/html/rfc6749#section-1.3>.
- [35] Open mHealth. Schema library, 2017. URL <http://www.openmhealth.org/documentation/#!/schema-docs/schema-library>.
- [36] Open mHealth. Schema design principles, 2017. URL <http://www.openmhealth.org/documentation/#!/schema-docs/schema-design-principles>.
- [37] Open mHealth. Write your own quantitative measure schema, 2017. URL <http://www.openmhealth.org/documentation/#!/schema-docs/write-a-schema>.
- [38] Sony mobile. Smartwatch 3 swr50, 2015. URL <https://www.sonymobile.com/global-en/products/smart-products/smartwatch-3-swr50/>.
- [39] Wearable. Moto 360 sport: Everything you need to know about the new gps smartwatch, 2015. URL <https://www.wearable.com/android-wear/new-moto-360-sport-2-price-release-date-specs>.
- [40] Runkeeper. How to use runkeeper with android wear, 2017. URL <https://support.runkeeper.com/hc/en-us/articles/202988016-How-to-use-Runkeeper-with-Android-Wear>.
- [41] Liferhacker. How accurately do running apps track your distance, 2015. URL <https://www.liferhacker.com.au/2015/10/how-accurately-do-running-apps-track-your-distance/>.
- [42] Health Graph. Introducing the healthgraph, 2017. URL <https://runkeeper.com/developer/healthgraph/introducing-the-health-graph>.
- [43] Health Graph. Overview, 2017. URL <https://runkeeper.com/developer/healthgraph/overview>.
- [44] Health Graph. Developer's console, 2017. URL <https://runkeeper.com/developer/healthgraph/console>.

- [45] Carl Foster, Jessica A. Florhaug, Jodi Franklin, Lori Gottschall, Lauri A. Hrovatin, Suzanne Parker, Pamela Doleshal, and Christopher Dodge. A new approach to monitoring exercise training. *Journal of Strength and Conditioning Research*, 15(1):109–115, 2001.
- [46] Kim-Edgar Sørensen. *Ruoksat. A system for capturing, persisting and presenting the digital footprint of soccer knowledge and expertise*. PhD thesis, University of Tromsø, 2013. URL <http://munin.uit.no/handle/10037/5434>.
- [47] Richard P. Troiano, James J. McClain, Robert J. Brychta, and Kong Y. Chen. Evolution of accelerometer methods for physical activity research. *Br J Sports Med.*, 48(13):1019–1023, 2014.
- [48] Open mHealth. Install shimmer, 2015. URL <http://www.openmhealth.org/documentation/#/data-providers/install-shimmer>.
- [49] Open mHealth. Shimmer, 2016. URL <https://github.com/openmhealth/shimmer/>.
- [50] Ubuntu. Cronhowto, 2016. URL <https://help.ubuntu.com/community/CronHowto>.
- [51] MongoDB. `db.collection.update()`, 2017. URL <https://docs.mongodb.com/manual/reference/method/db.collection.update/>.
- [52] Shoshana Zuboff. Big other - surveillance capitalism and the prospects of an information civilization. *Journal of Information Technology*, 30(5): 75–86, 2015.
- [53] Apache. What is apache hadoop?, 2017. URL <http://hadoop.apache.org/>.
- [54] Apache. Spark, lightning-fast cluster computing, 2017. URL <http://spark.apache.org/>.
- [55] William Kent. A simple guide to five normal forms in relational database theory. *Communications of the ACM*, 1982.
- [56] D3.js. Data-driven documents, 2017. URL <https://d3js.org/>.
- [57] Mapbox. Mapbox, 2017. URL <https://www.mapbox.com/maps/>.
- [58] Google Maps. Google maps apis, 2017. URL <https://developers.google.com/maps/>.
- [59] Open mHealth. Start processing data, 2017. URL <http://www.openmhealth.org/documentation/#/process-data/process-overview>.