Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Eivind Fjeldstad

A Proof-of-Concept Digital Twin

Combining data from off-the-shelves health and fitness trackers

Master's thesis in Computer Science
Supervisor: Frank Lindseth
June 2019

NTNU
Norwegian University of
Science and Technology

Sammendrag

Tingenes internett-produkter utstyrt med sensorer for å spore fysiologiske signaler har blitt stadig vanligere de siste årene. Ved å kombinere data fra flere slike enheter kan det være mulig å konstruere en digital sanntidsrepresentasjon av brukerens fysiologiske tilstand, en digital tvilling. Når en slik representasjon kombineres med moderne maskinlæringsteknikker kan den bli et kraftig verktøy for personlig helsestyring.

Et konseptbevis på en digital tvilling ble utviklet for å kunne samle helsedata fra tre kommersielt tilgjengelige helse- og treningssporere. Et system for å varsle brukere om helserelaterte uregelmessigheter ble implementert, og teknikker for å holde brukeres helsedata private og trygge ble utforsket. Systemet ble testet for å evaluere gjennomførbarhet og lagringskrav, og dataene samlet inn under testene ble brukt til å trene en rudimentær maskinlæringsmodell.

Denne avhandlingen viser hvordan en digital tvilling kan implementeres som en nettjeneste med en JSON API og en tilhørende mobilapplikasjon, og hvordan brukere kan bli varslet om potensielle helseproblemer ved bruk av push-meldinger. Testing avslørte at en forsinkelse på 0 til 20 minutter, noen ganger mer, må forventes ved synkronisering av helseprøver fra de utvalgte helse- og treningssporerne til den digitale tvillingen, og at gjennomsnittsstørrelsen på en lagret helseprøve var 220 byte.

Resultatene viser at digital tvilling-konseptet er gjennomførbart, men at å innhente sanntidsinformasjon fra kommersielt tilgjengelige helse- og treningssporere kan være vanskelig på grunn av tilgangsbegrensninger og synkroniseringsforsinkelser. De demonstrere også hvordan push-meldinger kan benyttes for

å gi brukere informasjon om deres helsetilstand, samt hvordan data fra digitale tvillinger kan brukes til å trene maskinlæringsmodeller. Testfasen ga informasjon om lagringskapasiteten som kreves av systemet, og en formel ble foreslått for å kalkulere lagrinskravene til en større utplassering.

Abstract

Internet of Things devices equipped with sensors to measure physiological signals have become increasingly common in recent years. By combining data from multiple health and fitness tracking devices, it could be possible to construct a real-time, digital representation of the user's physiological state in the cloud, a Digital Twin. When combined with modern machine learning techniques, this data could become a powerful health management tool.

A proof-of-concept implementation of a Digital Twin was developed in order to collect data from three off-the-shelf health and fitness trackers. A system for alerting users to health-related irregularities was implemented, and techniques for keeping users' health data private and secure were explored. The system was tested in order to assess feasibility and storage requirements, and a proof-of-concept machine learning model was trained using data gathered by the system.

This thesis shows how a Digital Twin can be implemented as a web service with a JSON API and a companion mobile application, and how users can be alerted to potential health problems using push notifications. Testing revealed that delays of 0 to 20 minutes, sometimes more, must be expected when synchronizing health samples from the chosen health and fitness trackers to the Digital Twin, and that the average size of a health sample stored by the system was 220 bytes.

The results show that the Digital Twin concept is feasible, but that obtaining real-time information from off-the-shelf health and fitness trackers can be difficult due to access restrictions and synchronization delays. They also demonstrate how push notifications can be used to give users insights about their health, and

how data from Digital Twins can be used to train machine learning models. The testing phase provided insights about the storage requirements of the system, and a formula was proposed for calculating storage requirements for a large deployment.

Acknowledgments

I would like to thank my supervisor, Prof. Frank Lindseth, for his guidance and advice throughout the project. I would also like to thank my friends and family for all their support during my time at NTNU.

Contents

List of Figures

List of Tables

Listings

1. Introduction

1.1 Motivation

Internet of Things (IoT) devices containing sensors that can sample physiological signals from users have become increasingly common in recent years. Smart watches often contain optical and electrical heart rate monitors, accelerometers, gyroscopes and GPS [7, 19, 54]. There are also IoT consumer products for measuring blood pressure, blood glucose levels, weight, body fat percentage, and a number of other dimensions related to health and fitness. When used together, these devices gather an enormous amount of data about the user's health. By analyzing data from such devices with modern machine learning techniques, researchers have been able to reliably predict a number of different medical conditions [9, 47]. Devices like the Apple Watch already use information from it's own sensors to alert users to atrial fibrillation and other heart irregularities [7]. By combining data from multiple off-the-shelf health and fitness tracking devices, it could be possible to construct a near real-time physiological snapshot of the user's body, a Digital Twin. With the addition of machine learning models and other predictive systems, a Digital Twin powered by sensors from off-the-shelf health and fitness trackers has the potential be a powerful tool for consumers, health care professionals and researchers.

1.2 Objectives and Research Questions

The main objective of this thesis to explore the concept of a Digital Twin by implementing a proof-of-concept system in order to investigate the feasibility of the idea. Another objective was to investigate how data from the Digital Twin can be utilized in useful ways, and what the the storage requirements of such a system would be. The research questions are as follows:

- **RQ1:** Is it feasible to construct a real-time, digital representation of physiological state in the cloud, a Digital Twin, using various off-the-shelf health and fitness trackers?
- **RQ2:** How might a Digital Twin be implemented, and what are the storage requirements of such a system?
- **RQ3:** How can Digital Twins be useful to users, health care professionals and researchers?

1.3 Contributions

This project demonstrates how a Digital Twin can be implemented, and how data from multiple health and fitness trackers can be collected. A proof-of-concept machine learning model is used to show how predictive technologies can be integrated into the system. The thesis also demonstrates ways in which a Digital Twin could help inform users about their health situation. Additionally, insights are given into the storage requirements of the system, and a formula is proposed for calculating the storage requirements of larger deployments.

1.4 Structure

The thesis is divided into six chapters. The remaining chapters are structured as follows:

2. **Background:** Introduces the devices, technologies, frameworks and concepts that are used in the project, as well as relevant, previous research.

3. **Methods and Implementation:** Describes how the system was developed and tested.

4. **Results:** Presents the results of the development process, as well as the results gathered from the tests.

5. **Discussion:** Discusses the results as they relate to the research questions.

6. **Conclusions and Future Work:** Contains some concluding remarks, and presents suggestions for future work.

2. Background

This section presents background information on a selection of health and fitness tracking devices, frameworks, APIs and other technologies used during the project, as well as relevant, previous research.

2.1 Prior Art

2.1.1 iOS Health

Apples Health app, which is available on iOS, is an application that aims to consolidate health data from Apple devices, like the Apple Watch and iPhone, and third-party apps installed on those devices [1]. The app gives the user the opportunity to view and manage their health data. The App is a front-end to the HealthKit store, which is described in section 2.5.6. Some of the categories that can be tracked through the Health app are sleep, activity, mindfulness, nutrition, cardiovascular health, body measurements and vitals.

2.1.2 Google Fit

Google Fit is a collection of APIs and services for interacting with fitness data from various sources [36]. A central repository is used for storing the data, which can be accessed via a web API [38]. However, the developer documentation for Google Fit explicitly states that the Google Fit API should not be used to store medical or biometric data [36]. Google also provides a mobile application with the same name, avaliable on iOS and Android, through which users can view and interact with their data.

2.2 Health and Fitness Trackers

2.2.1 Fitbit Charge 3

Figure 2.1: Fitbit Charge 3. Image from [19].

The Fitbit Charge 3 is an armband that functions as a fitness tracker [19]. It comes equipped with an optical heart rate monitor, an accelerometer, an altimeter and a pulse oximeter, allthough the pulse oximeter is not currently used by the software. Users can track heart rate, activity and sleep, and synchronize data to compatible devices, via Bluetooth, by installing the Fitbit app on the target device. The Fitbit app is available on iOS, Android, macOS and Windows. Data from the Fitbit app is regularly uploaded to Fitbit's servers, and can be accessed via the Fitbit Web API. The Fitbit Web API is described in section 2.3.1.

2.2.2 Apple Watch Series 4

Figure 2.2: Apple Watch Series 4. Image from [8].

The Apple Watch Series 4 is a smart watch produced by Apple. It is equipped with both optical and electrical heart sensors, an accelerometer, a gyroscope and a GPS [7]. The electrical heart sensor is used for electrocardiography, while the optical heart sensor is used to measure heart rate. The watch is able to detect and alert users to various heart irregularities, like unusually high or low heart rates and atrial fibrillation. Activity tracking is done using the accelerometer, gyroscope and GPS. Health data from the Apple Watch is synchronized with the user's iPhone, and is stored in the HealthKit store. HealthKit is described in section 2.5.6.

2.2.3 Withings BPM

Figure 2.3: Withings BPM. Image from [53].

The Withings BPM is a wireless blood pressure cuff [54]. The cuff measures the users blood pressure and heart rate, and sends the measurements via Bluetooth to the Withings Health Mate app instlled on the user's iOS or Android device. The Health Mate app integrates with both HealtKit and Google Fit. Data can also be accessed using the Withings Web API, which is described in section 2.3.2.

2.3 Web APIs

2.3.1 Fitbit Web API

Fitbit give developers access to data from Fitbit devices through their Web API [20]. In order to use the API, an application has to be registered with Fitbit. The types of data the application has access to depends, in part, on the type of application. Personal applications are given full access, which includes access to intraday time-series data belonging to the owner of the the application. Commercial and multi-user applications are limited to aggregate time-series data. Fitbit does grant access to intraday time-series data on a case-by-case basis for research applications. API requests are rate-limited to 150 requests per hour,

per user. The Fitbit API makes use of the OAuth 2.0 Authorization Framework, described in section 2.5.17, in order to authorize access to a user's data.

2.3.2 Withings Web API

Data from Withings devices can be accessed via the Withings Web API [52]. Like Fitbit, Withings require that applications using the API be registered in order to obtain an Application ID. The Withings Web API uses the Oauth 2.0 Authorization framework for authorization. Requests are rate-limited to 120 requests per minute, per application.

2.4 The HTTP protocol

The Hypertext Transfer Protocol (HTTP) is a *stateless application-level protocol for distributed, collaborative, hypertext information systems*[39]. It is a request/response protocol. Typically, a client, such as a web browser, sends a request message to a server, which returns a response message. A request message contains a verb, indicating the request method, request headers and a message body. A response message contains a status code, response headers and a message body.

2.4.1 Request Methods

The verbs representing the request methods used in this thesis are are summarized in table 2.1.

Verb	**Description**.
GET	Requests the transfer of a resource. GET is commonly used when loading a website.
POST	Requests that the message payload be processed by the resource. POST is commonly used when submitting a form on a website.
PUT	Requests that the target resource be created or updated with the message payload.
PATCH	Requests that the target resource be partially updated with the message payload.
DELETE	Requests that the target resource be deleted.

Table 2.1: HTTP methods used in this thesis

2.4.2 Status Codes

There are a number of valid status codes [40]. In general, status codes can be divided into five categories based on their leading digit:

- **1xx:** Informational
- **2xx:** Successful
- **2xx:** Redirection
- **2xx:** Client error
- **2xx:** Server error

The status codes used in this project are summarized in table 2.2. For a full list of status codes and their semantics, see RFC 7231 [40].

Status Code	**Description**.
200 (OK)	The request was successful.
201 (Created)	One or more resources have been created.
301 (Found)	The resource requested is temporarily located elsewhere. The location of the resource should be provided in the `Location` header of the response.
400 (Bad Request)	The client has committed an error. The reason for the error should be specified.
401 (Unauthorized)	The request did not provide authentication credentials, or the credentials were refused access to the resource.
404 (Not Found)	The resource was not found.
500 (Internal Server Error)	The server encountered an error and was unable to fulfill the request.

Table 2.2: HTTP status codes used in this project.

2.4.3 Headers

HTTP headers are used to provide additional information with a request or a response. A 302 (Found) response, for example, should contain a `Location` header specifying the location of the resource that was requested. HTTP headers are also used to specify the media type of a resource by setting the `Content-Type` header. A request containing a JSON body, for example, would have the `Content-Type` set to `application/json`.

2.4.4 Basic Authentication

HTTP Basic Authentication allows a client to provide a username and password when making a request to a server [41]. If a server receives a request for a URL in a protected space, the server can respond with a challenge by us-

ing the 401 (Unauthorized) status code and setting an HTTP header named `WWW-Authenticate`. The value of the header should be of the form `Basic realm="<realm>"`, where `<realm>` is a string indicating the name of the protected space. The client then responds to the challenge by setting an HTTP header named `Authorization`. The value of the header should be of the form `Basic <credentials>`, where `<credentials>` is a base64 encoded string containing the username and password, concatenated by a colon (":").

Because HTTP Basic Authentication does not include an encryption scheme, credentials are sent in clear text across the network. It is not recommended to use HTTP Basic Authentication without enhancements such as HTTPS [41].

2.5 Software Frameworks and Libraries

2.5.1 Xcode

Xcode is an Integrated Development Environment (IDE) for macOS [55]. It provides various tools for developing software for Apple platforms, including code autocompletion, device simulators, application templates and instrumentation and debugging solutions.

2.5.2 Swift

Swift is an open source programming language maintained by Apple [44]. It is one of the officially supported languages for developing software targeting the iOS platform. Apple also provides a number of Swift-compatible SDKs and frameworks to developers.

2.5.3 Foundation

The Foundation framework for Swift and Objective-C provides developers with basic data types and operating system services when developing software for Apple platforms [21].

2.5.4 UIKit

UIKit is a framework that provides a foundational infrastructure for building iOS applications using Swift or Objective-C [48]. The framework includes components for implementing user interfaces, reacting to touch gestures and handling events, among other things.

2.5.5 UserNotifications

UserNotifications is a framework for Apple platforms that contains push notification-related functionality [49]. There are APIs for, among other things, requesting permission to use push notifications from the user, scheduling push notification locally, registering for remote notifications, and handling notification-related actions performed by the user.

2.5.6 HealthKit

HealthKit is a framework interacting with and manipulating health data in Swift or Objective-C, on the iOS and Watch-OS platforms [22]. HealthKit provides a central store for health data, and APIs to read and write data to and from the store.

Quantity Types

There are several different data types that can be queried from the HealthKit store [13]. Relevant for this thesis are quantity types, which represents data that can be described by a numeric value, like heart rate, weight, and body temperature. A single instance of a quantity type is referred to as a sample.

Queries

HealthKit provides a number of ways to query samples from the store [37]. The ones that are relevant to this thesis are listed below.

- **Observer queries** are long-running queries that execute a callback whenever new data has been added to, or removed from, the HealthKit store

[24]. Observer queries only inform the application that something changed, not the specific samples that were involved in the change. Observer queries can also be configured for background delivery. In practice, this means that even if the user were to exit the application, the callbacks would still be executed in the background.

- **Anchored queries** are queries that, when executed, will return all samples of a particular type that were added to the store after the last time a query was executed [23]. This is achieved by making used of a query anchor parameter, which represents the last sample returned from the previous query. In addition to any new data, an anchored query will return a new query anchor which can be used to limit the scope of subsequent queries.

2.5.7 Keychain Services

Keychain Services are part of the Security framework for Apple platforms [28]. Keychain is an encrypted database and password management system available on Apple operating systems such as iOS and macOS. Keychain services APIs give applications a mechanism to store sensitive information, such as passwords or other secrets, in a secure way.

2.5.8 Node.js

Node.js is a JavaScript runtime built on top of Chrome's V8 JavaScript engine [4]. It is available for a variety of operating systems, including MacOS, Windows and Linux.

2.5.9 Yarn

Yarn is a package manager for JavaScript [18]. Dependencies are managed in a `package.json` file, which specifies the name and version of the packages Yarn should install. Yarn can pull packages from the npm registry, which is a public registry where developers can publish open source JavaScript libraries.

2.5.10 Koa

Koa is a framework for Node.js that provides components and APIs for building web services using middleware [29]. In the context of Koa, middleware are functions that are called in a stack-like manner for each incoming request. Examples of middleware functions are routers, loggers and request body parsers.

2.5.11 Mongoose

Mongoose is an object-document mapper for Node.js and MongoDB [32]. It wraps MongoDB documents in objects that expose methods for interacting with the underlying data in various ways. An application interacts with MongoDB collections by defining models that represent the underlying document type. A Mongoose model provides functionality to query, insert, update and delete documents with object oriented APIs. MongoDB is described in section 2.6.1.

2.5.12 Agenda

Agenda is a job scheduling library for Node.js [5]. It contains APIs for scheduling and executing jobs at regular intervals, or on specific dates.

2.5.13 Luxon

Luxon is a JavaScript library that provides a wrapper for native JavaScript date types [31]. The wrapper class has a number of convenience methods for manipulating dates, and has built-in support for doing time zone conversions.

2.5.14 node-apn

node-apn is a Node.js library for interacting with the Apple Push Notification Service [34]. The library provides an object oriented interface through which an application can send push notifications without worrying about the details of the the underlying networking. The Apple Push Notification Service is described in section 2.6.3.

2.5.15 SuperAgent

SuperAgent is an HTTP request library for Node.js and the browser [43]. It handles much of the low level plumbing invloved in making HTTP requrests, and has an API that reduces the amount of code needed for common use-cases.

2.5.16 TensorFlow.js

TensorFlow.js is an open source library for machine learning [6]. It provides APIs for training and running machine learning models using Node.js, or in the browser.

2.5.17 OAuth 2.0

OAuth 2.0 is an open standard for secure authorization [45]. The OAuth 2.0 authorization framework allows a user of an OAuth provider to grant limited access to a third-party application. When granted such access, the third-party application can access the service's resources on behalf of the user. Several different authorization strategies are typically available, one of which is the Authorization Code Grant flow. A high-level overview of the Authorization Code Grant flow is given by the steps outlined below.

1. The client redirects the resource owner, typically an Internet user, to the OAuth provider's authorization server along with a client identifier. The client identifier uniquely identifies the third-party application.

2. The resource owner authenticates with the authorization server and accepts or denies the client's access request.

3. The authorization server redirects the resource owner back to the third-party application along with an authorization code.

4. The third-party application uses the authorization code to obtain an access token and an optional refresh token. The access token is used to access the resources to which the application has been granted access, while the refresh token is used to renew the access token once it expires.

2.5.18 Nginx

Nginx is an open source web server, load balancer and reverse proxy [33]. It is available for a wide variety of operating systems, including Windows, macOS, Linux, and Solaris.

2.6 Platforms and Infrastructure

2.6.1 MongoDB

MongoDB is a schemaless, NoSQL document database [27]. MongoDB documents are JSON-like objects that can be queried using the MongoDB query language. Documents reside in collections, which are comparable to tables in relational databases. The database also supports a number of index types and strategies [26]. Relevant to this thesis are:

- **Unique indexes:** Ensures that the database rejects duplicate values for the indexed fields.
- **TTL indexes:** Automatically deletes the indexed documents after a certain amount of time.

2.6.2 Digital Ocean

Digital Ocean is a cloud platform that offers a variety of different cloud products [51]. Among them are so-called *droplets*, which are virtual machines that can be provisioned in a variety of configurations [17]. Other available products are object storage, firewalls, load balancers, DNS services and managed databases [14].

2.6.3 Apple Push Notification Service

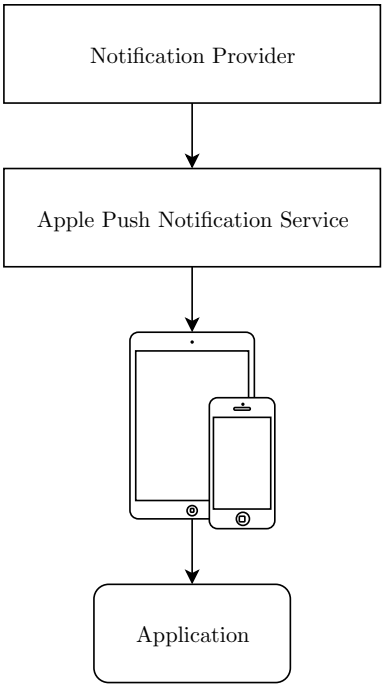


Figure 2.4: Delivering notifications using APNs

The Apple Push Notification Service, or APNs, is a service for delivering push notifications to Apple devices [42]. By connecting to the APNs, notification providers can deliver notifications to users' devices remotely. The APNs maintains a persistent connection to the user's device, which is used to deliver the notifications. Figure 2.4 illustrates the delivery process.

2.6.4 Docker

Docker is a tool that allows software to run in virtual, isolated environments called containers [16]. Containers make use of operating system-level virtualization, which allows isolated user-space instances to run on a single kernel. A container image is a template for creating a container, and a `Dockerfile` is used to define the steps needed to build the image. By embedding it in a container image, a piece of software can be run on any machine where OS-level virtualization is available without having to install application dependencies on the host

operating system.

Docker Compose

Docker Compose *is a tool for for defining and running multi-container Docker applications* [35]. A `docker-compose.yml` file is used to define and configure the services of the application, as well as any volumes and networks that should be available to those services [12]. Docker Compose can then be used to run and manage the entire application.

Docker Hub

Docker Hub is public registry for publishing and distributing pre-built container images [15].

2.7 Security

2.7.1 Hashing and Encryption

Bcrypt

Bcrypt is a password hashing function based on the Blowfish encryption algorithm [2]. A JavaScript implementation of Bcrypt is available via the Yarn package manager under the name *bcrypt* [10].

Transport Layer Security

The goal of Transport Layer Security, or TLS, is to allow secure communication over the Internet [46]. It consists of a handshake protocol and a record protocol. The handshake protocol authenticates the parties and negotiates which cryptographic modes and parameters to use. The record protocol protects traffic between the parties by using the modes and parameters agreed to during the handshake. The HTTP protocol does not, by default, employ any means of encryption, and all data is sent in plain text over the network [39]. When used

together, TLS provides a layer of security on top of HTTP. The resulting protocol is often referred to as the Hypertext Transfer Protocol Secure, or HTTPS [25].

As part of the handshake protocol, the server typically presents the client with a certificate that is signed by a Certificate Authority [46]. This allows the client look up the certificate with the Certificate Authority to confirm the identity of the server. Let's Encrypt is a Certificate Authority that provides free certificates for domain names [30].

2.7.2 Random Numbers

UUID

A UUID is a 128-bit number, represented by 32 hexadecimal digits, used as a universally unique identifier [3]. There are four versions of UUID defined in RFC 4122. Only version 4 is used in this thesis. A version-4 UUID has 122 randomly generated bits for a total of $2^{122} \approx 5.317 \times 10^{36}$ possible values. For this reason, UUIDs are assumed to be unique across time and space. A JavaScript implementation of the UUID generation algorithm is available via the Yarn package manager under the package name *uuid* [50].

2.8 Relevant Research

2.8.1 DeepHeart: Semi-Supervised Sequence Learning for Cardiovascular Risk Prediction

Ballinger et al. used heart rate data from the Apple Watches of 14 011 study participants to train and validate a semi-supervised, multi-task LSTM to identify diabetes, high cholesterol, high blood pressure and sleep apnea [9]. Their results are summarized in table 2.3. The study showed that using data from off-the-shelves health and fitness trackers can be used to detect multiple medical conditions with high accuracy.

Condition	C-statistic
Diabetes	0.8451
High Cholesterol	0.7441
Sleep Apnea	0.8298
High Blood Pressure	0.8086

Table 2.3: Results from Ballinger et al [9]. The c-statistic is the area under the ROC curve, which is the true positive rate plotted against the false positive rate for every cutoff rule.

2.8.2 Passive Detection of Atrial Fibrillation Using a Commercially Available Smartwatch

Tison et al. used heart rate and step count data from the Apple Watches of 9750 study participants to train a neural network to identify atrial fibrillation [47]. They achieved a sensitivity of 98.0% and a specificity of 90.2% in a validation cohort of 51 patients, measured against ECG-diagnosis. The study showed that detecting atrial fibrillation using heart rate and step data from an Apple Watch is possible, but with some loss of specificity and sensitivity compared to ECG.

3. Methods and Implementation

In order to answer the research questions, I decided to develop a proof-of-concept implementation of a Digital Twin system. The system would collect data from a selection of off-the-shelf health and fitness trackers. The data would be used to monitor the health of the user and provide health related alerts and notifications. The system would be tested by multiple users for a period of time in order to gain insights about the system. Data gathered from the system would then be used to train a proof-of-concept machine learning model.

3.1 Selecting Devices

A Fitbit Charge 3, an Apple Watch Series 4 and a Withings BPM were selected as sources of data for the project. Some of the devices, like the Apple Watch and the Fitbit, had overlapping capabilities, but all three had different data access possibilities and restrictions. Data from the Apple Watch is written to the HealthKit store on a companion iPhone device, and is only accessible through APIs provided by the HealthKit SDK for iOS. The Fitbit requires a companion application on iOS, Android, macOS or Windows, but its data is regularly uploaded to Fitbit's servers, and can be accessed using the Fitbit Web API. The Withings BPM synchronizes data with an iOS or Android device, and data can be accessed through the Withings Web API, Google Fit or HealthKit.

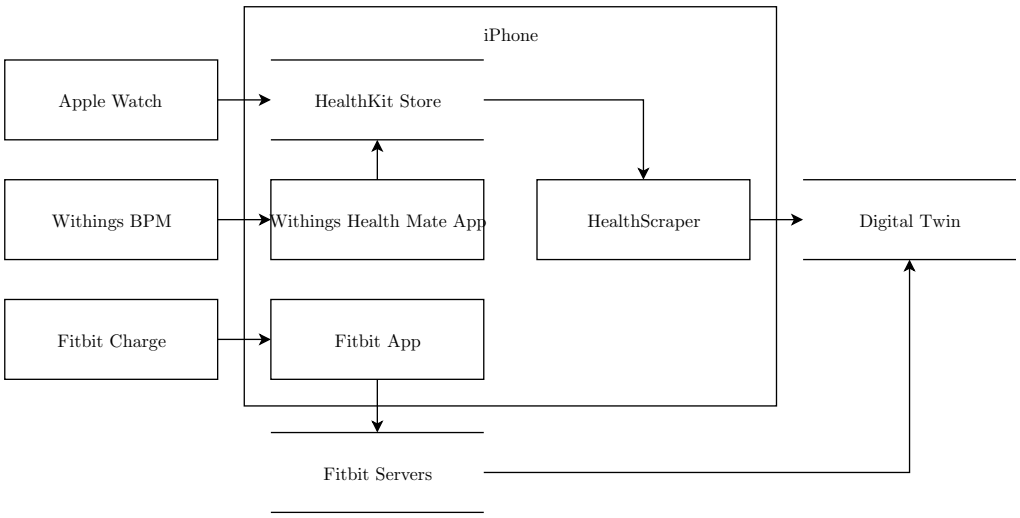


Figure 3.1: Data flow of the system

3.2 Developing Software

Because of the data access limitation of the Apple Watch, an iOS app had to be developed. The iOS app would have a dual purpose:

1. Upload data from the HealthKit store to the Digital Twin.
2. Serve as the main interface between the user and the Digital Twin.

To that end, the system would be composed of two main components:

- The Digital Twin, located in the cloud.
- The iOS application, named HealthScraper, located on the user's iPhone.

Because the Withings Health Mate app could be configured to write data to the HealthKit store, the system needed to be integrated with HealthKit and the Fitbit Web API in order to access the data from all three devices. The data flow of the system is summarized in figure 3.1.

3.2.1 The Digital Twin

To resemble the concept of a Digital Twin outlined in the introduction to this thesis, the proof-of-concept would need to have have the following capabilities:

- Collect health data from multiple sources.
- Store the collected data.
- Process and analyze the collected data.
- Provide users with alerts and notifications.
- Allow users to access their data.

Languages and Frameworks

The software for the Digital Twin was written in JavaScript, using Node.js and the Koa web framework. JavaScript and Node.js were chosen because of the large ecosystem of frameworks and tools for building web apps that are available through the Yarn package manager, which was used for dependency management. JavaScript also has native support for JSON, which is a major component of many web APIs.

Database

A MongoDB database was used to store application data. Because MongoDB is a schemaless NoSQL database, health samples with different data types could be stored, along with arbitrary metadata, without having to create and maintain a separate table for each permutation. MongoDB was also chosen because it is open source, and free to install on any compatible system. Mongoose, the object document mapper for MongoDB and Node.js, was used to model the structure of, and interact with, the database from Node.js.

Web API

The public-facing part of the Node.js application was designed as a JSON API. JSON was chosen because it works natively in Node.js, and because it is human-readable, making requests and responses easy to inspect and debug. As a starting point, API endpoints were created for the following purposes:

- Creating user accounts
- Updating user accounts
- Writing health samples
- Reading health samples

3.2.2 The HealthScrapper iOS Application

In order to fulfill its dual purpose as a middle-man and an interface to the Digital Twin, the HealthScraper application needed the following capabilities:

- Allow the user to create an account on the Digital Twin, or sign in using an existing account.
- Allow the user to enable or disable Fitbit and HealthKit synchronization.
- Transfer data from the HealthKit store to the Digital Twin.
- Allow the user to browse health data stored on the Digital Twin.
- Receive alerts and notifications from the Digital Twin.

The application would utilize the public-facing web API to communicate with the Digital Twin.

Languages and Frameworks

The HealthScraper application was developed using Xcode and the Swift programming language. The user interface was based on components provided by UIKit. Networking code was implemented using APIs from the Foundation framework.

Local Storage

User settings and application state was stored in the user's default database, which is a key-value store for persisting data across app launches. The APIs for interacting with the user's default database were provided by the Foundation framework.

3.2.3 User Accounts

The user accounts on the Digital Twin were identified by a unique username. Users would also have a password in order to authenticate with the web API, which is described in detail in section 3.3.1. Because MongoDB does not enforce a strict schema, other data, like third-party authorization tokens and device tokens, could also be stored, when needed, together with the user's account information

3.2.4 Integrating with HealthKit

When a user enabled HealthKit synchronization using the HealthScraper application on the iPhone, the app would register long running queries on a number of different HealthKit quantity types. These long running queries, called observer queries, would notify the application whenever new data was entered into the HealthKit store. Using observer queries allowed the application to be notified of new data even if the user had exited the application. The queried quantity types were:

- Heart rate
- Step count
- Systolic blood pressure
- Diastolic blood pressure
- Distance walking/running

Synchronization

Whenever an observer query notified the application of new data, an anchored query would be used to fetch the latest samples from the HealthKit store. Anchored queries accept an argument referred to as a query anchor, which acts as a pointer to the last sample received from any previous query. In this way, the query anchor would ensure that only new data was returned. When the anchored query returned with new samples, the application would serialize the samples into JSON, and make an HTTP(S) POST request, with the serialized samples as the request body, to the Digital Twin. The Digital Twin would process and store the samples in the MongoDB database. The anchored query would also return a new query anchor, which would be serialized and stored in the user's default database. The next time an observer query would notify the application of new data, the query anchor would be de-serialized and passed along with a new anchored query.

3.2.5 Integrating with Fitbit

Data from the Fitbit was synchronized automatically with the Fitbit app on iOS, which would upload the data to the Fitbit servers. The data could then be accessed via the Fitbit Web API. To make use of the API, a personal app was registered with Fitbit and a Fitbit App ID was obtained. The Fitbit App ID would serve as the client ID during the OAuth 2.0 authorization flow. Registering a personal app was necessary in order to get access to intraday time-series data, which was not available through other app types without explicit permission from Fitbit. Authorization was done using OAuth 2.0 and the Authorization Code Grant Flow. The steps of this process are described below.

1. Using the HealthScraper application, the user would toggle a switch, enabling an option to synchronize Fitbit data with the Digital Twin.

2. The HealthScraper application would request a one-time, temporary authentication token from the Digital Twin.

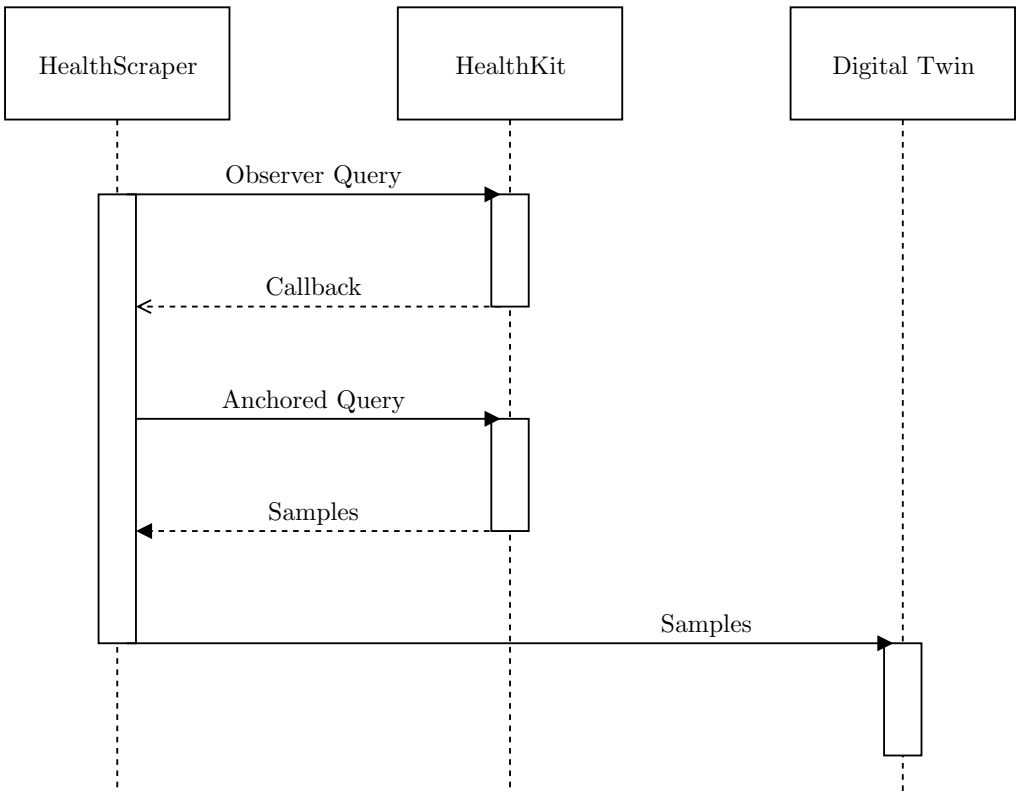


Figure 3.2: Sequence Diagram of the HealthKit synchronization process

3. The HealthScraper application would open a browser window to a specific endpoint exposed by the Digital Twin, passing the authentication token as a URL parameter.

4. Given a valid authentication token, the Digital Twin would redirected the browser to the Fitbit authorization page, passing the authentication token along as a special URL parameter named `state`.

5. The user would log in using their Fitbit credentials, and select which categories of data to share.

6. Fitbit would redirect the user's browser back to the Digital Twin, passing along an authorization code and `state`, the temporary authentication token, as a URL parameters.

7. The Digital Twin would use the authorization code to obtain an access token and a refresh token from Fitbit.

8. The Digital Twin would use the temporary authentication token to look up the user, and store the access and refresh tokens along with the user's account information. These tokens would allow the Digital Twin to make requests to the Fitbit API on the user's behalf.

9. The Digital Twin would respond with a 302 status code and redirect the browser using a custom URL scheme, `healthscraper://`. This custom URL scheme instructed iOS to open the link in the HealthScraper app and execute a callback. From this callback, the HelathScraper app would close the browser window, completing the authorization process.

The Fitbit authorization flow could have been handled in multiple ways. As an example, the Fitbit authorization page could have been opened directly from the HealthScraper app, without using the Digital Twin as a middle-man, passing the temporary authentication token as the `state` parameter. The approach outlined above was mainly chosen in order to avoid implementing OAuth-specific logic in two places. It also allowed the HealthScraper application to be decoupled

from the specifics of the Fitbit App registration process. In this way, the Health-Scraper app would be usable with any Digital Twin installation, independent of its Fitbit App ID.

Synchronization

Using the Agenda library for job scheduling and SuperAgent for making HTTP requests, the Digital Twin was configured to request data from the Fitbit API at regular time intervals. It was found that the Fitbit iOS app sent data to the Fitbit servers every 15 to 20 minutes. The Fitbit API enforced a strict API rate limit of 150 requests per user, per hour. Based on these factors, an interval of 10 minutes was chosen. The synchronized data types were:

- Heart rate
- Distance travelled
- Elevation
- Steps
- Sleep logs

Every 10 minutes, a request would be made to the Fitbit server for each data type and user. Whenever possible, request parameters would be set in order to limit the results to samples dated after the latest known sample. The results would then be processed, and each unique sample would be stored in the MongoDB database. The data processing is described in more detail in section 3.4.

When a user's Fitbit access token would expire, which would happen every 8 hours, the Digital Twin would automatically renew the access token using the refresh token obtained during the Authorization Code Grant flow.

3.3 Security

Due to the sensitive nature of health data, security was a major point of emphasis during the development of the system.

3.3.1 Authentication

Requests to the Digital Twin API, with the exception of account creation and Fitbit authorization requests, were authenticated using HTTP Basic Authentication. When an account was created on the Digital Twin, the account password was hashed using bcrypt and stored in the MongoDB database along with the username. When the user entered their credentials into the HealthScraper app, the username and password was verified with the Digital Twin before being stored on the iOS keychain. Whenever a request to a protected API endpoint the Digital Twin was made, the HealthScraper application would retrieve the user's credentials from the iOS keychain and send them along with the request by setting the `Authorization` header. When receiving the request, the Digital Twin would decode the credentials from the `Authorization` header, look up the username, hash the plaintext password from the request, and compare it to the password hash stored in the database. Comparisons were made using a constant-time algorithm provided by the node-bcrypt library. Hijacking of usernames was prevented by requiring usernames to be unique. Uniqueness was enforced by using a MongoDB unique index.

Password Policy

To encourage users to pick strong passwords, a minimum password length of 8 characters was chosen. If an account registration was attempted with a password shorter than 8 characters, the Digital Twin would respond with a 400 (Client Error) status code, and a message describing the password policy.

Temporary Authentication Tokens

The temporary tokens used to authenticate a user during the Fitbit Authorization Code Grant Flow were generated using the UUID library for Node.js. Tokens were stored in a MongoDB collection with a TTL index, which automatically deleted tokens one hour after creation. All tokens were of type UUID version 4, which are randomly generated, as is is described in RFC 4122 [3].

3.3.2 Authorization

An authenticated user on the Digital Twin could only retrieve data associated with their own account. This policy was enforced by storing a unique ID, associated with the user, along with every sample. When an authenticated user would request a list of samples using the web API, only the samples associated with their unique ID would be returned.

3.3.3 Transport Security

Because of the sensitive nature of health data, and because base64 encoded passwords were sent with nearly every request, all data transfer to and from the Digital Twin was done over the Hypertext Transfer Protocol Secure (HTTPS).

3.4 Data Processing

All health data sent to the Digital Twin was processed in order to enforce a common format. All numerical values were converted to common units, which are summarized in table 3.1.

Sample type	**Unit**
Heart rate	Beats per minute (bpm)
Steps	Raw number
Systolic blood pressure	Millimeter of mercury (mmHg)
Diastolic blood pressure	Millimeter of mercury (mmHg)
Distance	Meters (m)
Elevation	Meters above sea level (m a.s.l.)

Table 3.1: Units used for the Digital Twin

In addition to the raw sample value, the samples often came with additional metadata. The fields that were stored in the database are summarized in table 3.2.

Field	**Description**
`type`	The sample type.
`value`	The value of the sample.
`startDate`	The start date and time of the sampling.
`endDate`	The end date and time of the sampling
`source`	Where the sample came from.
`metadata`	An object containing any additional metadata not covered by the other fields.

Table 3.2: Database fields for samples

For the step, distance and elevation samples, zero values were discarded in order to minimize the amount of storage space needed. All dates were converted to UTC before being stored in the database. Time zone conversions were done using the Luxon library. The format used by the Fitbit API to express dates did not include time zone information. To work around this limitation, a separate request had to be made to the Fitbit API to obtain the user's profile settings, which contained a time zone setting. The time zone obtained from the user's profile was then used to convert the dates to UTC. Intraday time-series data

from the Fitbit API did not include a date, only a time stamp of the format `HH:mm`. However, because the Digital Twin synchronized data with Fitbit every 10 minutes, it was possible to reliably deduce the date based on the timestamp. If the synchronization interval had been more than 24 hours there would be a potential for overlapping time stamps, and it would have been impossible to deduce the date.

3.5 Push Notifications

Functionality for sending push notification to iOS devices with the HealthScraper application installed was implemented using the node-apn library in conjunction with the Apple Push Notification Service, or APNs. In order to make use of the APNs, the HealthScraper app had to be registered for usage with the service. This was done by enabling the Push Notifcation capabilities for the app using Xcode.

Configuring the HealthScraper app for recieving notifications was done in the following way:

1. When a user opened the HealthScraper app for the first time, the app would prompt them for permission to send push notifications.

2. If the user accepted the request for permission and subsequently logged in, the app would register the device for remote push notifications by calling a method provided by the UserNotifications framework.

3. When the application had been successfully registered for remote push notification, a callback would be invoked with an argument containing a unique device token. This token would then be sent to the Digital Twin using a designated API endpoint, and stored in the MongoDB database with the user's account information.

Using the node-apn library, the Digital Twin would maintain a persistent connection to the Apple Push Notification service. An authentication token obtained from Apple was used in order to establish that the Digital Twin was

authorized to send push notification on behalf of the HealthScraper application. Whenever a push notification was to be delivered to a particular user, the Digital Twin would POST a request to the APNs, supplying the details of the notification along with the user's device token. The APNs would then deliver the notification to the user's device.

Implementing Heart Rate Alerts

As a proof-of-concept, the Digital Twin was configured to monitor the user's synchronized data in order to detect abnormally low heart rates. This was done by using the Agenda library to schedule a job that would run every 20 minutes. The job consisted of querying the MongoDB database for heart rate readings below a certain threshold, limited to heart rate samples taken during the last 20 minutes. If such a value was found, the Digital Twin would send a push notification to the user's device, notifying them of the abnormality. The threshold was arbitrarily set to 40 beats per minute.

3.6 Deployment

In order to easily distribute the Digital Twin web application, a container image was built using Docker. The container image was based on the official Node.js image running Alpine Linux. The MongoDB database ran in a container based on the official MongoDB image. Both images were pulled from Docker Hub. A `docker-compose.yml` file was created, and the two services, the Digital Twin and the MongoDB database, were defined using their respective container images. The Digital Twin container image was then published to Docker Hub.

A virtual machine was provisioned on the Digital Ocean platform. The virtual machine had 1 GB of memory, a 25 GB disk, and was running Ubuntu 18.04. Docker and Docker Compose were installed, and the `docker-compose.yml` file was used to start the application on the virtual machine, pulling the pre-built images from Docker Hub.

A domain name was purchased, and DNS records were configured in order

to point the domain name to the IP address of the virtual machine. Finally, a signed TLS certificate was obtained from the certificate authority Let's Encrypt and installed on the Digital Twin.

3.7 Testing the System

Three users participated in the testing phase. All users were using iPhones with the latest version of iOS installed (version 12.3.1). The HealthScraper app was installed on all three phones, and used to create a user account on the Digital Twin deployed on the Digital Ocean platform. Two of the users wore an Apple Watch Series 4 for three days, while the third user wore an Apple Watch and a Fitbit Charge 3 for three days each. The third user also used the Withings Blood Pressure monitor to measure blood pressure two times per day, for a total of 12 times. The testing procedure consisted of:

1. Log in to the Digital Twin using the HealthScraper app.
2. Enable HealthKit or Fitbit synchronization, depending on which device was worn by the user.
3. Enable push notifications.
4. Verify that data is being synchronized by browsing collected samples with the HealthScraper app.

The Apple Watch-users were also instructed to open the Health app on the iPhone and manually insert a heart rate sample into the HealthKit store. The value of the heart rate sample was intentionally set to 39 beats per minute in order to trigger an abnormality alert from the Digital Twin. The users would then observe whether or not the push notification arrived.

3.8 Machine Learning

In order to demonstrate potential applications of the Digital Twin, a proof of concept machine learning model was created. The purpose of the model was to

demonstrate how machine learning technologies could be used within the system. The goal of was to predict the next measured heart rate of the user given the five latest heart rate readings. Because this was a proof-of-concept, the performance of the model was considered to be unimportant, and no serious attempts were made to optimize the model. In reality, there are of course a lot of factors that influence a persons heart rate.

Using Tensorflow.js, a simple neural network was constructed. The network had an input layer with five neurons, a hidden layer with three neurons, with bias, and an output layer with one neuron. Sigmoid was used as the activation function for the hidden layer, while the output layer used a linear activation function.

3.8.1 Training

The MongoDB database was queried for all heart rate samples, sorted by date. Each sample returned from the database, except the first five, were combined with the preceding five samples to form sequences of six. All timestamps were examined to make sure there were no gaps larger than one minute between successive samples in a sequence. If such gaps were found, the sequence was discarded. All raw values were normalized by dividing them by 250, which was naively assumed to be an upper limit for heart rate values. The first five values of a sequence were used as inputs to the network, with the sixth being used as the label. The dataset was then split into a training set and a validation set, where the training set contained of two-thirds of the original data, and the validation set contained the remaining third. The network was trained for five ephocs, with a batch size of 32, using Mean Squared Error as a loss function and the Adam algorithm as optimizer.

3.8.2 Deploying the Network

The network was deployed to the Digital Twin and made available through an API endpoint. When the endpoint received a POST request with a time-series of five heart rate samples, the Digital Twin would feed the samples to the model

and respond with a prediction.

4. Results

This section presents the proof-of-concept Digital Twin Web API and the Health-Scraper application, as well as results obtained during testing of the system.

4.1 The Digital Twin

In this section, an overview is given of the Digital Twin Web API.

4.1.1 Authentication

All requests to the API, apart from the endpoints for creating a new account, require some form of authentication. HTTP Basic Authentication is used for all protected endpoints, with the exception of those related to Fitbit authorization.

4.1.2 Errors

The Digital Twin Web API uses HTTP status codes to communicate error states. The status codes used are summarized in table 4.1.

Status Code	Description
400	Is used to indicate client error. The JSON response contains an `error` field with a message describing the error.
401	Is used to indicate that the given credentials were invalid, or that no credentials were given. A 401 response also presents an authentication challenge through the `WWW-Authenticate` header, which has the value `Basic realm="API"`.
500	Indicates server error. The JSON response contains an `error` field with a generic error message.

Table 4.1: Status codes used to communicate Digital Twin Web API errors

4.1.3 Endpoints

All endpoints of the Digital Twin Web API, as well as their parameters, are described below.

Create User

`POST /user`

Creates a new user account with the given parameters. Request parameters are described in table 4.2.

Parameter	Description
`username`	The user's username.
`email`	The user's email address.
`password`	The user's password.

Table 4.2: Create User request parameters

Update User

`PATCH /user`

Updates the user account of the authenticated user with the given parameters. Request parameters are described in table 4.3.

Parameter	Description
`email`	The user's email address.
`password`	The user's password.

Table 4.3: Update User request parameters

Create Authentication Token

`POST /user/token`

Creates a temporary authentication token for the authenticated user. The JSON response body contains a `token` field containing the generated token. An example JSON response is shown in listing 4.1.

```
{
    "token": "864bfa1e-17df-45f7-a5ec-abcc2e44f17e"
}
```

Listing 4.1: Example JSON response from the Create Authentication Token endpoint

Add Device Token

`POST /user/ios/device`

Saves the given device token with the authenticated user's profile information. The device token is used to send out push notifications using the Apple Push Notification service. Request parameters are described in table 4.4.

Parameter	Description
`token`	The device token.

Table 4.4: Add Device Token request parameters

Create Samples

`POST /samples`

Stores the given samples for the authenticated user. Request parameters are

described in table 4.5.

Parameter	Description
`samples`	An array of samples.
`samples[i].value`	Value of the sample.
`samples[i].startDate`	Start date of the sample.
`samples[i].endDate`	End date of the sample.
`samples[i].type`	Type of sample.
`samples[i].source`	Source of the sample.
`samples[i].metadata`	Any extra metadata for the sample.

Table 4.5: Create Samples request parameters. The letter i represents an array index.

List Samples

`GET /samples`

Returns samples belonging to the authenticated user. URL parameters are described in table 4.6. An example JSON response is shown in listing 4.2.

Parameter	Description
`startDate`	Start date for samples (optional).
`endDate`	End date for samples (optional).
`type`	Type of samples (optional).
`limit`	Maximum number of samples to return (optional).
`offset`	Number of samples to skip (optional).
`source`	Source of samples (optional).

Table 4.6: List Samples URL parameters

```
[
  {
    "type": "heartRate",
    "value": 57,
    "startDate": 1555888399000,
    "endDate": 1555888399000,
    "source": "healthKit",
    "metadata": {
      "uuid": "86E5D88C-72DB-4222-8E39-384F13F52983",
      "device": "Apple Watch"
    }
  },
  {
    "type": "stepCount",
    "value": 10,
    "endDate": 1555877769000,
    "startDate": 1555877769000,
    "source": "fitbit"
  }
]
```


Listing 4.2: Example JSON response from the List Samples endpoint. In reality, these responses often contain hundreds of samples, depending on the URL parameters used to limit the results.

Predict Heart Rate

`POST /prediction/heart-rate`

Returns a heart rate prediction generated by the machine learning model described in section 3.8, using the given sequence of heart rates as inputs. Request parameters are described in table 4.7. An example JSON response is shown in listing 4.3

Parameter	Description
`samples[]`	An array containing five time-ordered heart rate values, where the last element is the most recent.
`samples[i]`	A numeric heart rate value.

Table 4.7: Predict heart rate request parameters. The letter i represents an array index.

```
{
    "prediction": 63
}
```

Listing 4.3: Example JSON response from the Predict Heart Rate endpoint

Fitbit Authorization

`GET /fitbit/auth`

Starts the Fitbit Authorization Code Grant flow. This endpoint uses the `token` URL parameter to authenticate the user, and redirects to the Fitbit authorization page. URL parameters are described in table 4.8.

Parameter	Description
`token`	Temporary authentication token.

Table 4.8: Fitbit Authorization URL parameters

Fitbit Authorization Callback

`GET /fitbit/callback`

Acts as a callback for the Fitbit Authorization Code grant flow. The Fitbit authorization page redirects here when authorization is completed. The `state` parameter has the same value as the `token` parameter from the `GET /fitbit/auth` endpoint. It is used to authenticate the user before redirecting back to the HealthScraper iOS application. URL parameters are described in table 4.9.

Parameter	Description
`state`	Temporary authentication token.

Table 4.9: Fitbit Authorization Callback URL parameters

4.2 The HealthScraper iOS Application

The HealthScraper application for iOS consists of eight main views. In this section, an overview of the application's functionality is given by going through each view.

4.2.1 Initial Setup

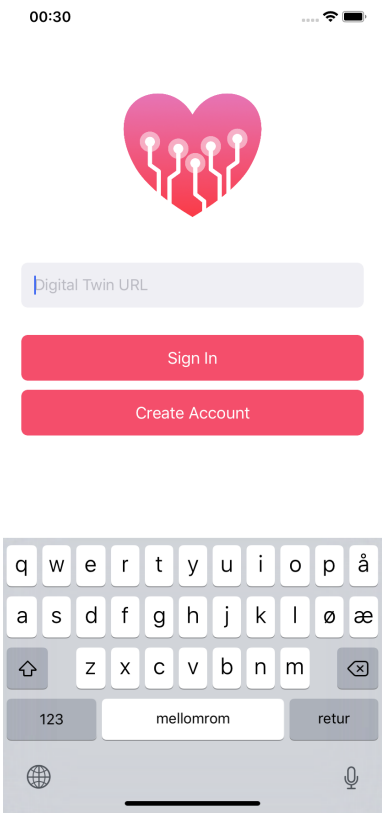


Figure 4.1: Setup screen

When first opening the application, the user will be presented with the Setup view. This view contains an input field for entering the URL of the Digital Twin. This allows the application to work with any Digital Twin installation that is available via the Internet. The user then has the option to tap one of two buttons. The first button opens the Sign In view, while the second button opens the Create Account view. A screenshot of the Setup view is provided in figure 4.1.

4.2.2 Signing In

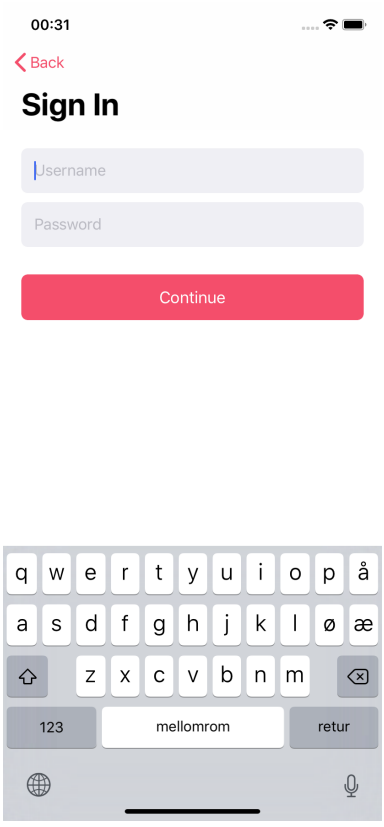


Figure 4.2: Sign in screen

If the user tapped the Sign In button after completing the initial setup, they are presented with a view containing input fields for a username and a password, as well as a button labeled Continue. When the user has entered their credentials and tapped the Continue button, the credentials are verified with the Digital

Twin, and the sign in process is completed. If verification fails, the user will be presented with an error message. A screenshot of the Sign In view is provided in figure 4.2.

4.2.3 Creating an Account

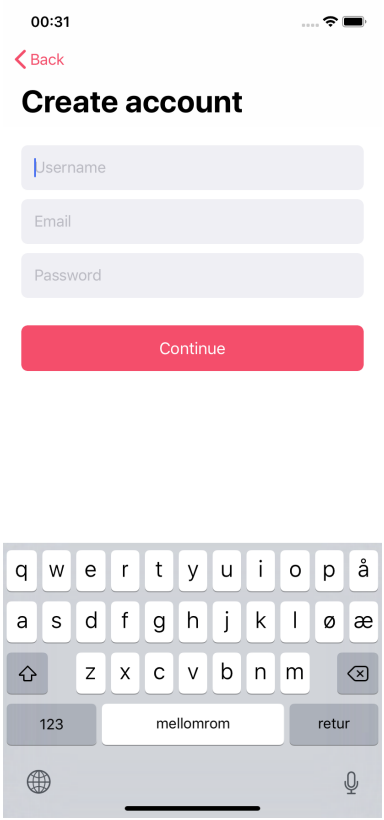


Figure 4.3: Create Account screen

If the user tapped the Create Account button after completing the initial setup, they are presented with the Create Account view. This view contains input fields for a username, an email address and a password, as well as a Continue button. The user proceeds by filling out the fields and tapping the Continue button, after which they are logged in automatically, and the sign up process is completed. If the username is already in use, or if the password did not meet the minimum length requirements, the user will be presented with an error message. A screenshot of the Create Account view is provided in figure 4.3.

4.2.4 Main View

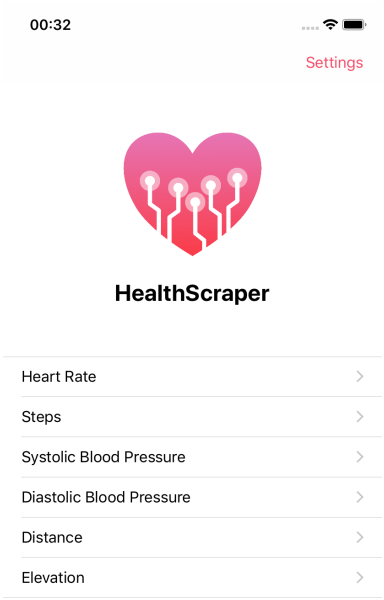


Figure 4.4: Main screen

After the user has signed in successfully, they are presented with the main view. This view contains a list of all the sample types available through the Digital Twin Web API. When the user taps a sample type, they are taken to a view showing a list of the samples that are currently stored on the Digital Twin. There is a button on the navigation bar labeled Settings that, when tapped, opens the Settings view. A screenshot of the main view is provided in figure 4.4.

4.2.5 Settings

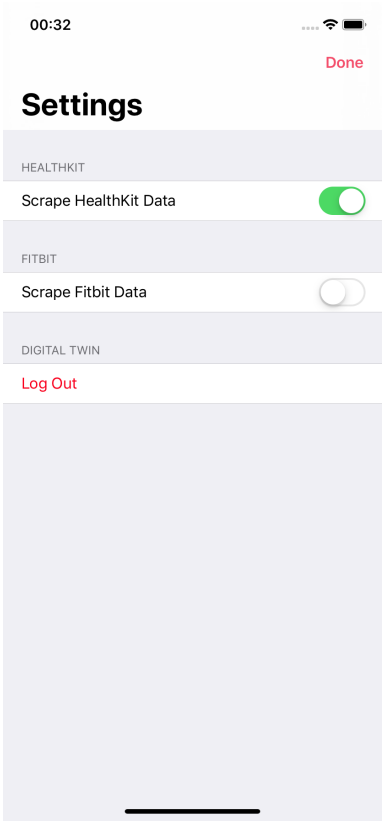


Figure 4.5: Settings screen

The Settings view contains switches for toggling HealthKit and Fitbit synchronization. When the Fitbit switch is toggled, the user is presented with a Safari window that opens the Digital Twin's Fitbit Authorization endpoint, which redirects to the Fitbit authorization page. The screen also contains a button labeled Log Out, which, when tapped, logs the user out and resets the application to its initial state. A screenshot of the Settings view is provided in figure 4.5.

4.2.6 Browsing Samples

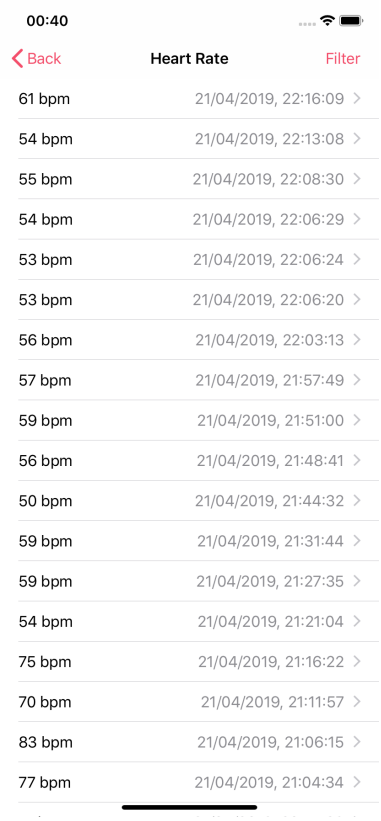


Figure 4.6: Samples list screen

A list of samples fetched from the Digital Twin API is shown when the user taps a sample category in the main view. Each list item displays the value and date of the sample it represents. The samples are sorted by date in descending order. When a sample is selected, the user is presented with a view containing more information about the sample. There is also a Filter button on the navigation bar which, when tapped, displays a view with options for filtering samples by date. A screenshot of the Samples view is provided in figure 4.6.

4.2.7 Filtering Samples

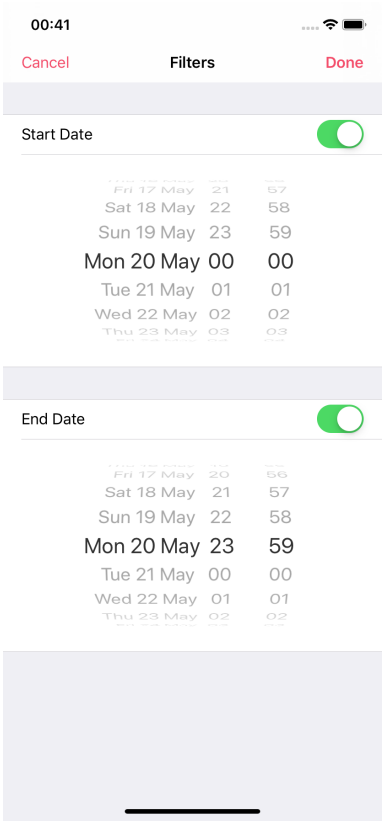


Figure 4.7: Filters screen

When the user taps the Filter button on the navigation bar of the Samples view, they are shown a view containing options to filter samples by start date and end date. When dates are selected and the Apply button, which is located on the navigation bar, is tapped, the user is taken back to the Samples view, where only samples matching the filters are displayed. A screenshot of the Filters view is provided in figure 4.7.

4.2.8 Viewing Sample Details

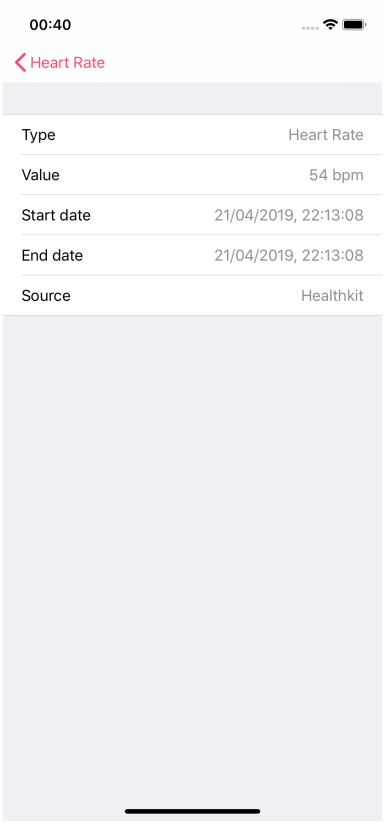


Figure 4.8: Sample details screen

When the user taps a sample from the Samples view, they are shown a view containing additional information about the sample. The information is presented in a table containing cells for the sample type, value, start date, end date, and source. A screenshot of the Sample Details view is provided in figure 4.8.

4.2.9 Heart Rate Alert

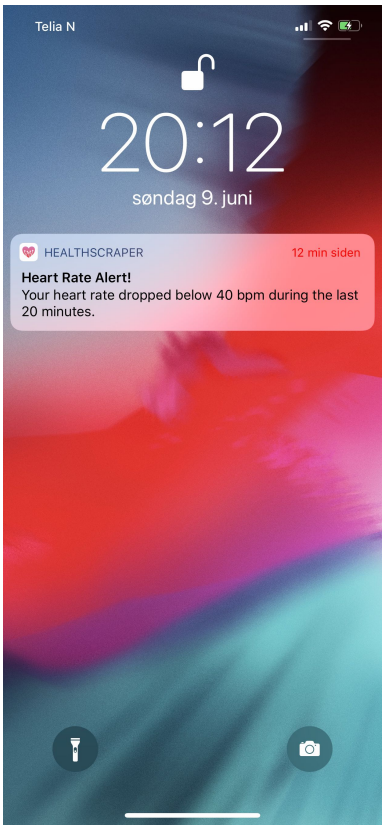


Figure 4.9: Heart Rate Alert Notification

If the Digital Twin perceives that the user's heart rate has dropped below 40 beats per minute during the last 20 minutes, a notification is sent to the user's device. The notification warns the user that their heart rate has been unusually low during the last 20 minute period. A screenshot of the notification is provided in figure 4.9.

4.3 Testing the System

All three testers reported that the system worked as expected. All three were also able to trigger a heart rate alert by manually entering a heart rate below 40 beats per minute, which arrived in the form of a push notification a few minutes later. The Digital Twin reliably synchronized data with both the Fitbit API and

the HealthKit store, and no problems with data synchronization were reported.

4.4 Synchronization Rate

The synchronization delay between the Fitbit application and the Fitbit servers was observed to be anywhere between 0 to 20 minutes. This fact, combined with the 10 minute interval between each request from the Digital Twin to the Fitbit API, meant that samples from the Fitbit Charge 3 would usually arrive somewhere between 0 to 30 minutes after they were taken. If, however, the iPhone with the Fitbit application installed was without a network connection when a synchronization event was scheduled, the delay could be even longer. Data from the Apple Watch and Withings BPM was delivered almost instantaneously as long as the iPhone was in an unlocked state. However, because the HealthKit store is encrypted when the iPhone is locked with a passcode, HealthScraper could not access new health samples until the iPhone was in an unlocked state. This was not a problem with data from the Withings BPM, which can only be used while the Withings Health Mate app is open on the user's device. For the Apple Watch however, this meant that the synchronization rate would be entirely dependent upon how often the user would check their phone.

4.4.1 Collected Data

During the testing period, more than 14 000 samples were collected by the Digital Twin, of which the vast majority, roughly 85%, were heart rate readings. In any 24-hour period, the average user uploaded more than 3500 individual samples. The average size of a sample in the MongoDB database was 220 bytes, and the total size of the collection of samples, excluding database indexes, was on the order of 3.1 Megabytes.

5. Discussion

5.1 Research Question 1

> Is it feasible to construct a real-time, digital representation of physiological state in the cloud, a Digital Twin, using various off-the-shelf health and fitness trackers?

The Digital Twin implementation presented in this thesis managed to gather data from multiple off-the-shelf health and fitness trackers. The system worked reliably throughout the testing period. However, having a real-time representation of physiological state in the cloud, with data collected from the chosen ensemble of devices, would prove difficult due to data access restrictions imposed by the various device manufacturers. The synchronization delays introduced at all stages of the data pipeline made sure that samples would arrive minutes, and sometimes, although very rarely, even hours after they were taken. This problem might have been mitigated somewhat by switching out some of the devices, particularly the Apple Watch, with competing products. However, it seems unlikely that the synchronization delay can be completely eliminated when using off-the-shelf health and fitness trackers.

5.2 Research Question 2

> How might a Digital Twin be implemented, and what are the storage requirements of such a system?

As demonstrated by this project, a Digital Twin can be implemented as a

web application with a companion application on a suitable mobile device. The web application can expose a public API, which the companion application can use to interact with the system. This is definitely not the only way such a system can be designed. The need for a mobile companion application arose because of the specific access restrictions placed on data from the HealthKit store and, by extension, the Apple Watch. One could easily imagine a scenario where the interface between the user and the Digital Twin is a regular web site, where HTML, CSS and client-side JavaScript is used to create the user interface.

5.2.1 Privacy and Security

Privacy and security was a priority during the development of the Digital Twin concept. All network traffic was encrypted using TLS, and users' health data were kept separate by enforcing a strict authorization policy for API requests. Passwords were hashed using bcrypt before being stored in the database, and validation rules were put in place to ensure that users chose passwords that were at least eight characters long. Credentials on the iPhone, which were needed for authentication with the Digital Twin Web API, were stored in the iOS keychain, which is encrypted by the operating system. As an extra security measure, all health data belonging to a user could have been encrypted using the user's password as an encryption key. This could have been done before the samples were entered into the database, which would have made it virtually impossible to access users' health data without permission, even with full access to the database. One reason for not implementing such security measures was that unencrypted health samples were needed in order to implement the slow heart rate alert. Unencrypted samples was also needed to train machine learning models. Another approach to keeping user data private is having a separate Digital Twin instance for each user. This would, however, require that each user set up their own infrastructure with a cloud provider. Because of this, it was deemed impractical for the purposes of this project.

UUIDs as Authentication Tokens

UUIDs were used as temporary authentication tokens in order to identify users returning from the Fitbit authroization flow. RFC 4122 specifically warns that UUIDs should not be assumed to be hard to guess, and discourages using them for security capabilities [3, p. 15]. However, because the temporary authentication token is only used for Fitbit integration, and does not authorize the user to perform any other action on the Digital Twin, they were deemed sufficient. If the authentication tokens are to be used for other tasks, like accessing health data, a different solution should be considered.

5.2.2 Storage Requirements

Based on the data gathered during the testing of the system, estimations can be made about the amount of storage space needed to sustain a Digital Twin over time. Assuming the storage requirements grow linearly with the number of sensors and users that are added to the system, the storage requirements can be estimated using the equation:

$$\begin{aligned}\text{Storage needed} \approx{}& \text{Average sample size} \times \text{Average number of sensors per user} \\ &\times \text{Number of users} \times \text{Average sampling rate of sensors} \times t\end{aligned}$$

For the system developed during this project, assuming three users would use it for three years, with an average sampling rate of 2000 per day, and with 10 sensors each, the total amount of storage needed in order to store the samples would be approximately:

$$220\,\text{B} \times 3 \times 10 \times \frac{2000}{\text{days}} \times 1095\,\text{days} \approx 14.5\,\text{GB}$$

This is likely an upper estimate, as the average sampling rate observed during testing was lower than 2000 per day. Additionally, because heart rate samples account for approximately 85% of the samples, it is perhaps not unreasonable to assume that the average sensor has a lower sampling rate than a heart rate monitor, and that the average sampling rate would decrease as more sensors are added to the system.

In this estimation, it is assumed that the size of other types of data, like user profiles and database indexes, are negligible compared to the size of the collected samples.

5.3 Research Question 3

> How can Digital Twins be useful to users, health care professionals and researchers?

5.3.1 Alerts and Notifications

As demonstrated by the proof-of-concept slow heart rate alert, having a Digital Twin has the potential to help make users aware of various medical conditions by delivering notifications if abnormalities are detected. Based on the results of previous research [9, 47], it is reasonable to assume that having a Digital Twin equipped with sophisticated prediction models could provide potentially life-saving information to the user. As more sensors are added to the system, the number of diseases and conditions that can be reliably identified is sure to increase. To that end, having a Digital Twin could help with early detection, and reduce the likelihood of serious illness or death. Some companies are already working on similar technologies. The Apple Watch, for example, can alert users to atrial fibrillation and other heart irregularities [7].

Another area where a Digital Twin might prove useful is with preventative health care. In the same way a Digital Twin could notify the user of serious medical conditions, it could provide intelligent suggestions in order to help users improve their health before serious conditions arise.

5.3.2 Data Sharing

In addition to providing helpful insights to the user, a Digital Twin could also share these insights with doctors and care-takers. In the same way a user could get a notification about a possible health problem, their doctor could too. The Digital Twin can even share the patient's raw data with the doctor to aid in

the diagnostic process. In this way, the doctor could be made aware of potential problems the patient might have before they enter the clinic. Apple has already done some work in this area with their CareKit framework [11], which aims to make it easier to develop applications with such functionality. Data sharing could also be useful for research projects. Machine learning applications require large amounts of data for training, and a large deployment of Digital Twins could provide an enormous amounts of health data to researchers and data scientists.

6. Conclusion and Future Work

6.1 Conclusion

The proof-of-concept system demonstrated how a Digital Twin can be implemented as a web service with a mobile companion application. Data was successfully synchronized to the Digital Twin from the Fitbit Charge 3, the Apple Watch Series 4 and the Withings BPM using various techniques. However, due to delays introduced at various stages of the data pipeline, real-time synchronization could not be achieved. All data sent to and from the Digital Twin was encrypted using TLS, and passwords were hashed using bcrypt. A strict authorization policy was enforced to prevent unauthorized access to a user's health data. The system also demonstrated how push notifications can be used to alert users to abnormalities observed by the Digital Twin, and how the gathered data can be used to train machine learning models. Data from user testing was used to create a formula for estimating the storage space needed when scaling the system further.

6.2 Future Work

6.2.1 Adding Support for More Devices

In order to collect more and different types of data, support more health and fitness trackers could be added to the system. This would likely also require adding integrations for other third-party APIs.

6.2.2 Machine Learning

Using results from previous research, it should be possible to train machine learning models that can make useful predictions about user health, using data from one or more Digital Twins. Such models could be scheduled to make predictions every few minutes, and send out push notifications to users if desired. This can be done using functionality that has already been implemented.

6.2.3 Security Testing

In order to ensure that user data is kept private and secure, a real security test should be performed. Such a test could include vulnerability scanning, penetration testing, risk assessment and auditing.

6.2.4 Developing an Android Application

The current implementation has an iOS companion app. Developing an Android version of this app would accommodate more users. The Android app would not have HealthKit integration, but could fulfil the role as an interface between the user and the Digital Twin.

6.2.5 Developing a Web Interface

A web interface could allow users to interact with their Digital Twin through a regular web browser. Currently, the only way a user can manage their Digital Twin is by using the iOS application. The web interface could be an accessible, alternative interface for users who don't have access to an iOS device. In this scenario, notifications could be delivered via email or SMS.

Bibliography

[1] *A bold way to look at your health.* URL: https://www.apple.com/ios/health/ (visited on 06/02/2019).

[2] *A Future-Adaptable Password Scheme.* URL: https://www.usenix.org/legacy/events/usenix99/provos/provos_html (visited on 05/30/2019).

[3] *A Universally Unique IDentifier (UUID) URN Namespace.* URL: https://www.ietf.org/rfc/rfc4122.txt (visited on 06/04/2019).

[4] *About Node.js.* URL: https://nodejs.org/en/about (visited on 05/07/2019).

[5] *Agenda.* URL: https://github.com/agenda/agenda (visited on 05/07/2019).

[6] *An end-to-end open source machine learning platform.* URL: https://www.tensorflow.org (visited on 06/06/2019).

[7] *Apple Watch Series 4.* URL: https://www.apple.com/apple-watch-series-4 (visited on 05/07/2019).

[8] *Apple Watch - Space Gray Aluminum Case with Black Sport Band.* URL: https://www.apple.com/shop/buy-watch/apple-watch/space-gray-aluminum-black-sport-band?preSelect=false&product=MU662LL/A&step=detail (visited on 06/11/2019).

[9] Brandon Ballinger et al. "DeepHeart: Semi-Supervised Sequence Learning for Cardiovascular Risk Prediction". In: *CoRR* abs/1802.02511 (2018). arXiv: 1802.02511. URL: http://arxiv.org/abs/1802.02511.

[10] *bcrypt.* URL: https://yarnpkg.com/en/package/bcrypt (visited on 04/12/2019).

[11] *CareKit*. URL: http://carekit.org/ (visited on 06/07/2019).

[12] *Compose file version 3 reference*. URL: https://docs.docker.com/compose/compose-file/ (visited on 05/15/2019).

[13] *Data Types*. URL: https://developer.apple.com/documentation/healthkit/data_types (visited on 05/09/2019).

[14] *Designed for developers. Built for businesses.* URL: https://www.digitalocean.com/products/droplets/ (visited on 06/05/2019).

[15] *Docker Hub Quickstart*. URL: https://docs.docker.com/docker-hub/ (visited on 05/15/2019).

[16] *Docker overview*. URL: https://docs.docker.com/engine/docker-overview/ (visited on 05/15/2019).

[17] *Easy, fast, and flexible compute*. URL: https://www.digitalocean.com/products/ (visited on 06/05/2019).

[18] *Fast, Reliable, and Secure Dependency Management.* URL: https://yarnpkg.com/en/ (visited on 05/10/2019).

[19] *Fitbit Charge 3*. URL: https://www.fitbit.com/charge3 (visited on 05/07/2019).

[20] *Fitbit Web API Reference*. URL: https://dev.fitbit.com/build/reference/web-api (visited on 05/07/2019).

[21] *Foundation*. URL: https://developer.apple.com/documentation/foundation (visited on 06/05/2019).

[22] *HealthKit*. URL: https://developer.apple.com/documentation/healthkit (visited on 05/07/2019).

[23] *HKAnchoredObjectQuery*. URL: https://developer.apple.com/documentation/healthkit/hkanchoredobjectquery (visited on 05/09/2019).

[24] *HKObserverQuery*. URL: https://developer.apple.com/documentation/healthkit/hkobserverquery (visited on 05/09/2019).

[25] *HTTP Over TLS*. URL: https://tools.ietf.org/html/rfc2818 (visited on 06/04/2019).

[26] *Indexes*. URL: https://docs.mongodb.com/manual/indexes/ (visited on 05/10/2019).

[27] *Introduction to MongoDB*. URL: https://docs.mongodb.com/manual/introduction/ (visited on 05/10/2019).

[28] *Keychain Services*. URL: https://developer.apple.com/documentation/security/keychain_services (visited on 05/10/2019).

[29] *Koa*. URL: https://koajs.com (visited on 05/07/2019).

[30] *Let's Encrypt*. URL: https://letsencrypt.org (visited on 06/04/2019).

[31] *Luxon*. URL: https://moment.github.io/luxon/ (visited on 05/11/2019).

[32] *mongoose*. URL: https://mongoosejs.com (visited on 05/12/2019).

[33] *nginx*. URL: https://nginx.org/en/ (visited on 04/12/2019).

[34] *node-apn*. URL: https://github.com/node-apn/node-apn (visited on 06/05/2019).

[35] *Overview of Docker Compose*. URL: https://docs.docker.com/compose/overview/ (visited on 05/15/2019).

[36] *Platform Overview*. URL: https://developers.google.com/fit/overview (visited on 05/02/2019).

[37] *Queries*. URL: https://developer.apple.com/documentation/healthkit/queries (visited on 05/09/2019).

[38] *REST API*. URL: https://developers.google.com/fit/rest/ (visited on 05/02/2019).

[39] *RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. URL: https://tools.ietf.org/html/rfc7230 (visited on 05/30/2019).

[40] *RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. URL: https://tools.ietf.org/html/rfc7231 (visited on 05/30/2019).

[41] *RFC 7617 - The 'Basic' HTTP Authentication Scheme*. URL: https://tools.ietf.org/html/rfc7617 (visited on 05/30/2019).

[42] *Setting Up a Remote Notification Server*. URL: https://developer.apple.com/documentation/usernotifications/setting_up_a_remote_notification_server (visited on 05/10/2019).

[43] *SuperAgent*. URL: http://visionmedia.github.io/superagent/ (visited on 06/05/2019).

[44] *Swift Documentation*. URL: https://swift.org/documentation/ (visited on 05/08/2019).

[45] *The OAuth 2.0 Authorization Framework*. URL: https://tools.ietf.org/html/rfc6749 (visited on 06/02/2019).

[46] *The Transport Layer Security (TLS) Protocol Version 1.3*. URL: https://tools.ietf.org/html/rfc8446 (visited on 06/04/2019).

[47] G. H. Tison et al. "Passive Detection of Atrial Fibrillation Using a Commercially Available Smartwatch". In: *JAMA Cardiol* 3.5 (May 2018), pp. 409–416.

[48] *UIKit*. URL: https://developer.apple.com/documentation/uikit (visited on 05/10/2019).

[49] *UserNotifications*. URL: https://developer.apple.com/documentation/usernotifications (visited on 05/10/2019).

[50] *uuid*. URL: https://yarnpkg.com/en/package/uuid (visited on 06/04/2019).

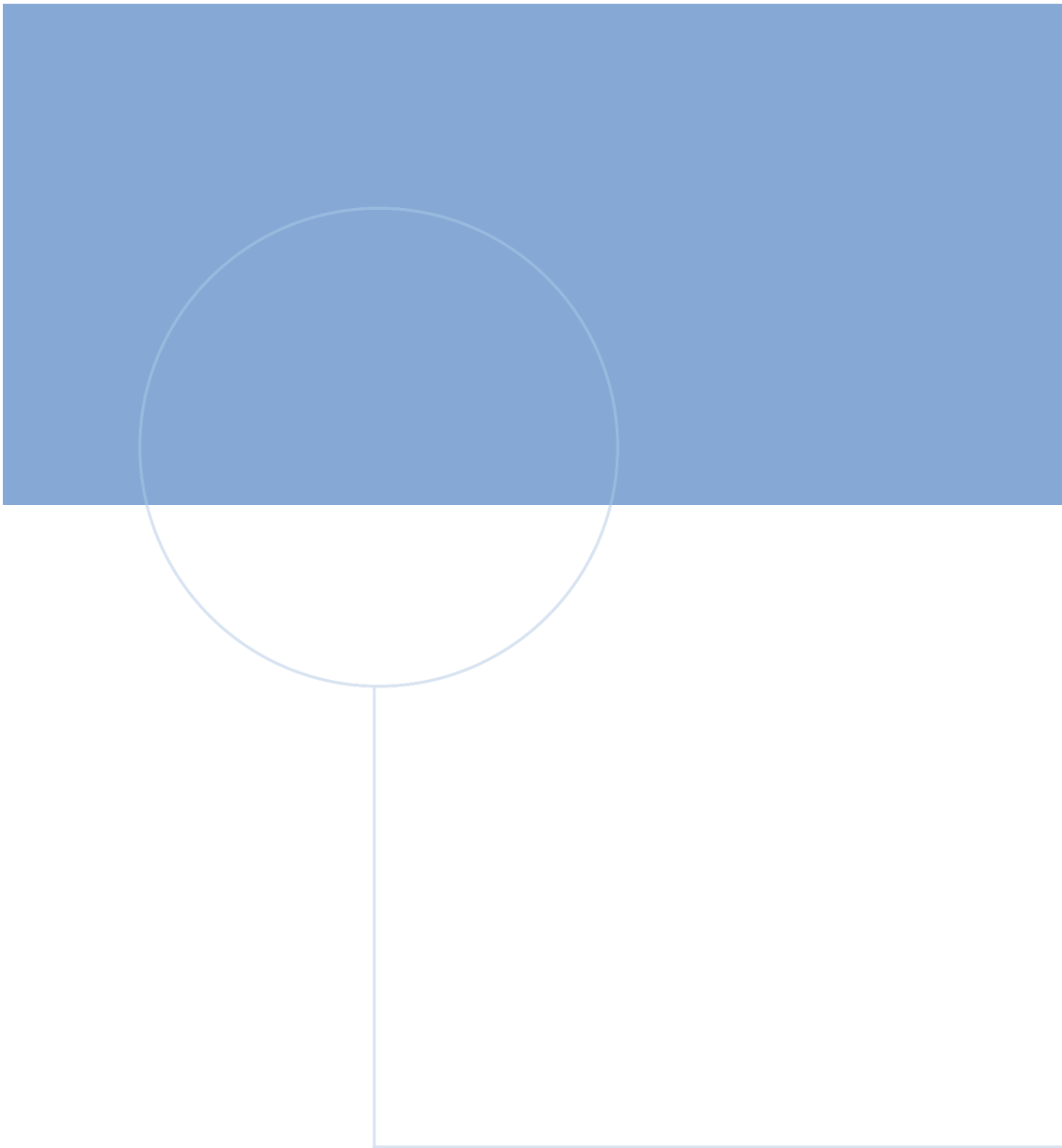
[51] *Welcome to the developer cloud*. URL: https://www.digitalocean.com (visited on 06/05/2019).

[52] *Withings API developer documentation*. URL: http://developer.withings.com/oauth2/ (visited on 05/08/2019).

[53] *Withings BPM Wireless Blood Pressure Monitor*. URL: https://www.apple.com/ie/shop/product/HMJR2ZM/A/withings-bpm-wireless-blood-pressure-monitor (visited on 06/11/2019).

[54] *Withings Thermo*. URL: https://www.withings.com/eu/en/blood-pressure-monitor (visited on 05/08/2019).

[55] *Xcode 10*. URL: https://developer.apple.com/xcode/ (visited on 05/08/2019).



NTNU
Norwegian University of
Science and Technology