



Semesterarbeit

Fitness Data Platform

Department of Computer Science
OST - University of Applied Sciences
Campus Rapperswil-Jona

Author(s) Joel Suter & Lucas von Niederhäusern

Advisor Frank Koch
Project Partner Michael Güntensperger
External Co-Examiner -
Internal Co-Examiner -

Abstract

Initial Situation: Today's world and day-to-day life are increasingly digitalized, with gadgets and devices that help with all kinds of tasks. This also includes the world of sports and health. The demand for trackers like smartwatches that monitor a person's vitals, fitness routine, and more shot up in the last few years. We envision massive potential in services and software in this domain. One of these potential services is a platform where users can synchronize multiple smartwatches or fitness trackers, view the data of these gadgets in a unified dashboard, and share their data with their fitness trainers, doctors, or friends.

Objective: With this project, we aim to build a basis for a platform of this type. The main goal is to create an extendable infrastructure and code that can be further expanded upon. That means we want to provide an initial platform that allows users to synchronize and save their data. The user should be able to view a dashboard where merged data of different trackers is displayed. He should be able to share it with the users he chooses. The platform must be structured to enable new features to be added easily in the future.

Conclusion: In this project, we created a prototype for a Fitness Data Platform by using TypeScript, Node.js, React, and MongoDB. Our prototype is a web application divided into frontend, backend, and a NonSQL database. This platform allows users to synchronize their Fitbit Connect and Garmin Connect accounts, view their weekly step count merged from multiple devices in a unified dashboard, and share these dashboards with other users by distributing their unique identification codes. Future implementations include additional support for other fitness tracking devices, more data on the dashboard, and being able to customize dashboards.



Figure 1: MongoDB



Figure 2: TypeScript

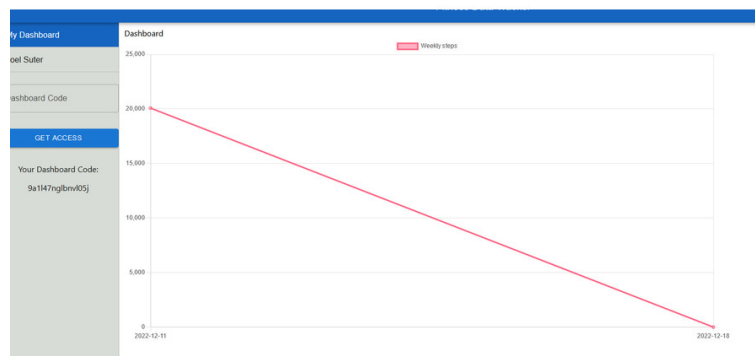


Figure 3: Own Presentment

Management Summary

With the advent of all kinds of fitness and health trackers, health and activity data has become more and more important, for private individuals who want to improve themselves or professional trainers or doctors trying to monitor their clients' progress. The problem with those metrics is that every manufacturer has an application or website to retrieve these data. Personal coaches would have to skim through multiple applications to collect all data from their clients. Furthermore, if the user has multiple tracking devices, it also becomes a hassle for the clients to keep up with their progress. The idea of this project is to collect all data from these fitness tracking devices in one place and display them in a dashboard. Clients can view their data in one place and share these dashboards with their trainers, other professionals, or friends to show off or have a friendly competition. Our platform currently supports devices from Fitbit and Garmin. It has excellent potential for further development as it is constructed modularly so that additional device supports can be implemented easily. Many metrics from the tracking devices are already stored in the database which have to be normalized and then are ready to be implemented on the frontend.

Glossary

backend The part of a computer system or application that is not directly accessed by the user, typically responsible for storing and manipulating data. 7, 14, 41

base64url Base64URL is a modification of the main Base64 standard, the purpose of which is the ability to use the encoding result as filename or URL address. 27

frontend The frontend includes all software or hardware that is part of a user interface. 14, 41

NoSQL A non-relational database technology. 14, 24

OAuth 2.0 OAuth 2.0 is the industry-standard protocol for authorization. 27

SHA-256 SHA-256 stands for Secure Hash Algorithm 256-bit and it's used for cryptographic security. 27

TypeScript TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale. 37

Acronyms

API Application Programming Interface. 7, 9, 17, 27, 36

CRUD Create, Read, Update, Delete. 17

DoD Definition of Done. 58

DoR Definition of Ready. 58

IDE Integrated Development Environment. 37

NFR Non-Functional Requirements. 24

Contents

1	Starting Position	6
1.1	Motivation	6
1.2	Project Description	6
2	Conceptual Formulation	7
3	Framework Conditions	8
4	System Context	9
5	Requirements	10
5.1	Functional Requirements	10
5.1.1	Additional information	11
5.2	Non-Functional Requirements	11
5.3	Optional Requirements	13
5.4	Verification	13
6	Design and Architecture	14
6.1	Container Diagram	14
6.2	Component Diagram	16
6.3	Backend architecture	17
6.4	Frontend architecture	18
6.4.1	UI Mockup	19
6.5	Scaling	21
6.5.1	Scaling up	21
6.5.2	Scaling out	22
6.6	Deployment	22
6.7	Domain Model	23
6.8	Database	24
6.8.1	Redundancy	24
6.8.2	Transactions and Rollback	25
6.9	Security	25
6.9.1	HTTPS and Domains	25
7	Implementation	27
7.1	Application Programming Interface	27
7.2	Dashboard Sharing	30
7.2.1	Share Dashboard	30
7.2.2	Switch Dashboard	32
7.2.3	Revoke Dashboard Access	33
7.3	Data Normalization	34
7.3.1	Weighting Algorithm	36
7.4	Logging	36
7.5	Code Documentation	37
7.6	Database	37

7.7	Displaying of Data	39
7.8	CI/CD and Hosting	41
8	Result	47
9	Conclusion	49
9.1	Needs to be addressed	49
9.2	Future Vision	49
10	Project and Time Management	50
10.1	Project Plan	50
10.1.1	Deviations	51
10.2	Risk Management	54
10.3	Test Concept	56
10.3.1	Additional	56
10.4	Definition of Ready / Definition of Done	58
11	Operational Notes	63
12	Meeting Minutes	64
13	Personal Reports	78
13.0.1	Joel Suter	78
14	Screenshots	79
15	Task	83
15.1	FitnessDataPlatform	83
15.1.1	Problembeschrieb	83
15.1.2	Aufgabenstellung	83
15.1.3	Technische Umgebung	83
15.1.4	Funktionale Anforderungen	83
15.1.5	Optionale Anforderungen	84
15.1.6	Nicht-Funktionale Anforderungen	84
15.1.7	Zur Durchführung	85
15.2	Additional Documents	85

1 Starting Position

1.1 Motivation

We had and still have a great interest in this project. When we first browsed through all the different project ideas that were available for the term paper, this one stood out the most. We are both sports and gym enthusiasts and have also worked with similar technologies in the past, which were required for this project. This project combined both our hobbies into one, so naturally, this was the project we hoped we could work on. Furthermore, there is a real need for a project of this type that combines tracker information into a single dashboard where the evaluation of these data can be done. Currently, it is also not possible to share this information easily without the need for exports or screenshots.

1.2 Project Description

Manufacturers usually have their own dashboards and graphs with which they can visualize their data from their clients. There are no solutions in the market where multiple brands and manufacturers are supported on the same dashboards, allowing users to share this data with fitness centers and personal coaches or friends. This paper is dedicated to creating such a solution from the ground up.

2 Conceptual Formulation

This project aims to create a web application that supports two fitness tracker manufacturers API's which are connected to a backend. Data delivered by the API will be evaluated and visualized on the dashboard, and these dashboards can be shared between users. A personal trainer or friend can see all his clients' or friends' data on the dashboards. In order to use this project in the future, the project must be built modularly to support more fitness tracker manufacturers in the near future.

3 Framework Conditions

This project is part of the "Semesterarbeit" which is a term paper required for the eligibility of the bachelor thesis. The planned time budget for this project is 240 hours and equals to 8 ECTS.

4 System Context

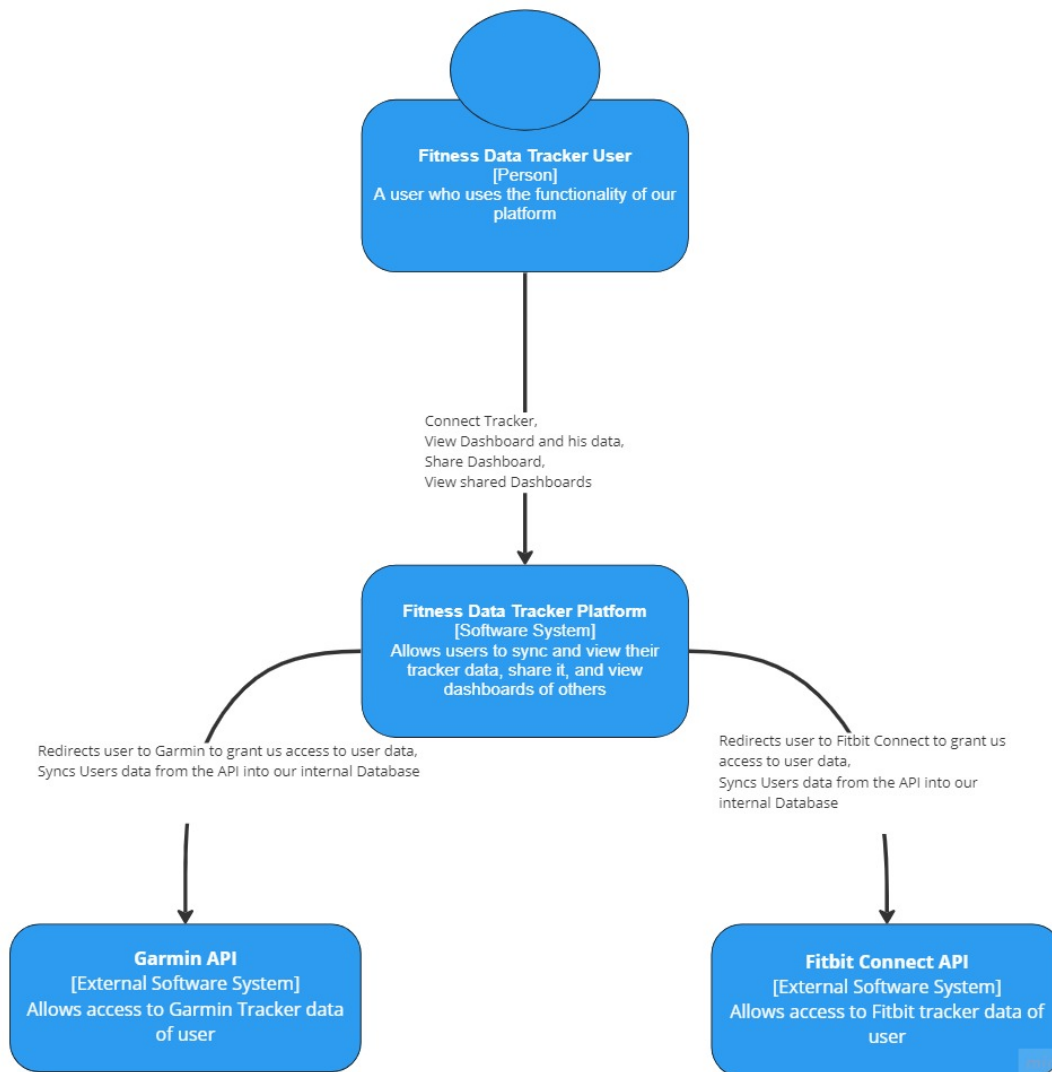


Figure 4.1: System Context

The above graphic shows the System Context of our application. We have the person "Fitness Data Tracker User" who represents our software system user. The User can be both a regular user and a professional. The distinction is only made for future development purposes, but currently, these users don't differentiate in functionality. Our Software System, "Fitness Data Tracker Platform" depends on two external Software Systems. Both are API's to Fitness Tracker devices. They allow our System to authorize, access, and sync user data to our internal Software System, which processes the data further to be displayed in the dashboard. Infrastructure hosting site [DigitalOcean](#) was provided by our industry partner, but dedicated servers and deployment had still to be set up.

5 Requirements

We use Agile Methodologies to keep track of our functional requirements. These are tracked as User-Stories or issues in Jira. We prioritize the User-Stories as and issues in our Jira-Backlog.

5.1 Functional Requirements

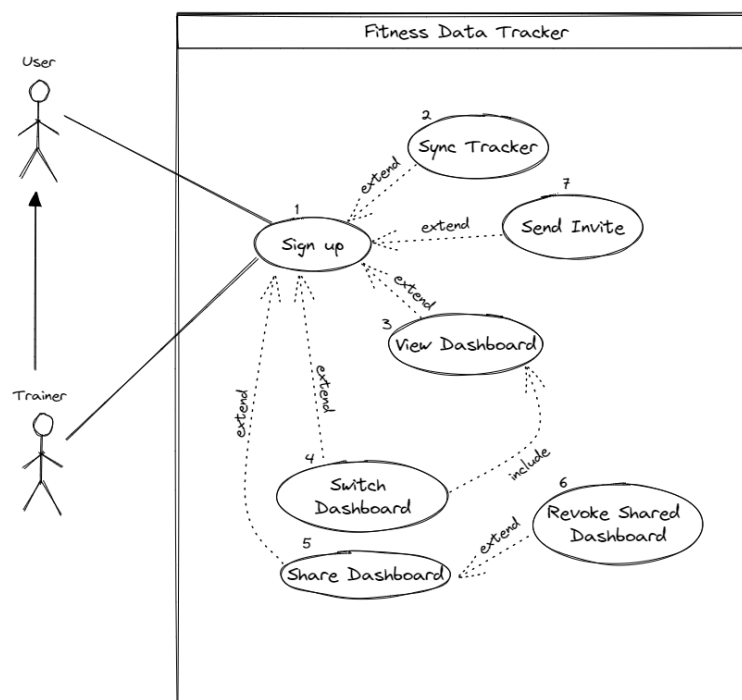


Figure 5.1: Testing Pyramid

We have two types of users in our system.

1. User: Standard user that uses this platform to track share and view the metrics of others.
2. Trainer: A User that uses the platform to track the metrics of his customers

Nr	UseCase	Description
1	Sign Up	User expresses the wish to be registered on the platform.
2	Sync Tracker	User expresses the wish to sync one of his fitness trackers.
3	View Dashboard	User expresses the wish to view a Dashboard with the tracked metrics.
4	Switch Dashboard	User expresses the wish to switch to another dashboard of another user.
5	Share Dashboard	User expresses the wish to share his dashboard with other users.
6	Revoke Shared Dashboard	User expresses the wish to revoke access of his dashboard from other users.
7	Send Invite	User expresses the wish to invite other users to use the platform.

Table 5.1: Use Cases

5.1.1 Additional information

The functional requirements represent the requirements provided in the initial task formulation. In the duration of the project one of the requirements changed. The changes done to them and the reason for that will be listed here:

3. View Dashboard: This functional requirement was limited after discussion with our supervisors to the weekly user steps a user takes. Reasons for that was the resources that would have to be reallocated to this requirement from other requirements like tracker syncing. So it made sense to limit this to focus on the core of the application.

Non-user stories Functional Requirements

- **Fitness Trackers:** The system should be able to sync two trackers from two distinct tracker manufacturers
- **Data Collection:** The application should be able to collect data from synced trackers and establish a connection/syncing with the chosen tracker manufacturers. As much data as possible should be collected.
- **Weighting Algorithm:** A user that has multiple device manufacturers synced should be able to view his merged data. That means our system should weigh data from both manufacturers and merge the data.
- **Application Deployment:** The application should be deployed and hosted.

5.2 Non-Functional Requirements

We express our Non-functional requirements as defined in FURPS+. While this project progresses, this table will be updated with new NFR's according to the knowledge of the domain and its technologies. All NFRs listed here are a must unless explicitly stated otherwise. Below is the List of NFRs in our project:

Type	Description	Acceptance Criteria
Functionality	All exceptions are handled and logged.	All Exceptions occurring in the tests are handled and logged. Furthermore, all possible exceptions should be tested.
Usability	The WebApp must run error free on Firefox, Chrome and Safari	Manually checked after each frontend change
Usability	The Domain must be able to be reached via the internet	Check while setting up this domain
Usability	At least three test users should rate the Application on a tablet with a grade of 8 from 10	Check at the end of the project
Reliability	Application data is backed up once a day	A backup is generated each day and checked manually.
Reliability	Errors do not cause Systemefailures, but generate an error message and reset the system to a previous state	Automated testing
Performance	The Backend (Fitnessdata) must be able to handle 1000 Requests per minute	Tested with Performance Testing Framework
Performance	Each page must not take longer than 200ms to load	Tested with Automated UI tests
Performance	The database must be able to handle at least 100 000 datasets	Test with SQL script
Supportability	The Business logic must be built in a modular way so that it can be easily extended	Business Reviews
Supportability	Backend-API must be tested with an API-Testing tool	Api Testing tool
Others (Security)	Data entered in input fields must be validated before it is passed to the backend	SQL Injection tests
Others (Security)	User passwords are not save in plain text	Review Database records
Others (Security)	The data of a user should not be visible to others unless he explicitly shared it	Testing
Others	Implemented functionality (Backend, Database, Frontend) must be deployed	CI/CD

Table 5.3: Non-Functional Requirements

Disregarded NFRs In the course of the project we decided to disregard some of the NFR, for various reasons which will be explained here.

- **The Backend (Fitnessdata) must be able to handle 1000 Requests per minute:** It didn't make much sense to stress test the backend since we currently host them on minimal required server ressource to keep costs down. This NFR should be considered before a go live.

- **Backend-API must be tested with an API-Testing tool:** We didn't consider this requirement since we didn't think the extra effort was worth it considering that backend functionality is fully tested with unit tests.

5.3 Optional Requirements

This section describes the optional requirements of our project. We didn't implement them since we focussed on making the core of our application robust and solid and didn't think it would make sense to take resources away from testing, refactoring and refining our core application. They will still be listed 5.5 here to keep track of them for future development purposes.

Type	Description
Non-Functional	User data is encrypted so that even the admin can't view them.
Functional	User is able to set goals/milestones in the dashboard
Functional	User is able to edit and customize the dashboard to display data he deems relevant
Functional	Personal Trainers are able to chat with client and upload videos that the user can view
Functional	Users can create communities for friends, company etc. Users are able to create challenges, chats and integrate these communities with Teams / Slack
Non-Functional	Monetizing of Professional-Accounts
Functional	User is able to manually enter data that gets considered in the whole evaluation
Functional	User is able to configure which data from which device will be used to create the dashboard

Table 5.5: Optional Requirements

5.4 Verification

If possible the NFR's will be checked automatically via automated tests. If that is not a possibility we will check each NFR after each sprint, and if not all are met the Sprint does not meet the DoD and has to be fixed as soon as possible.

6 Design and Architecture

6.1 Container Diagram

The container diagram 6.1 shows what technologies we chose to implement our application and how the code is split up. The frontend is implemented as a web application. Therefore, it allows the user to use our application without needing to download anything. We implemented our Fitness Data Tracker platform with a dedicated frontend and backend application. Two forces guided the technology decisions. First, the client suggested technologies and our knowledge and familiarity with them. The decisions were also guided by the current "state of the art". We chose MongoDB a NoSQL Database as our database to store the user information. Reasons for that decision are described in chapter 7.6. Our frontend uses HTTP calls to our backend API to make requests for data and more. Our backend reads and writes the data to the MongoDB using MongoDB, a NodeJS package. Our backend also communicates with two external systems, one for Garmin and one for Fitbit. With these API's we are able to read the user's data and save it to our database.

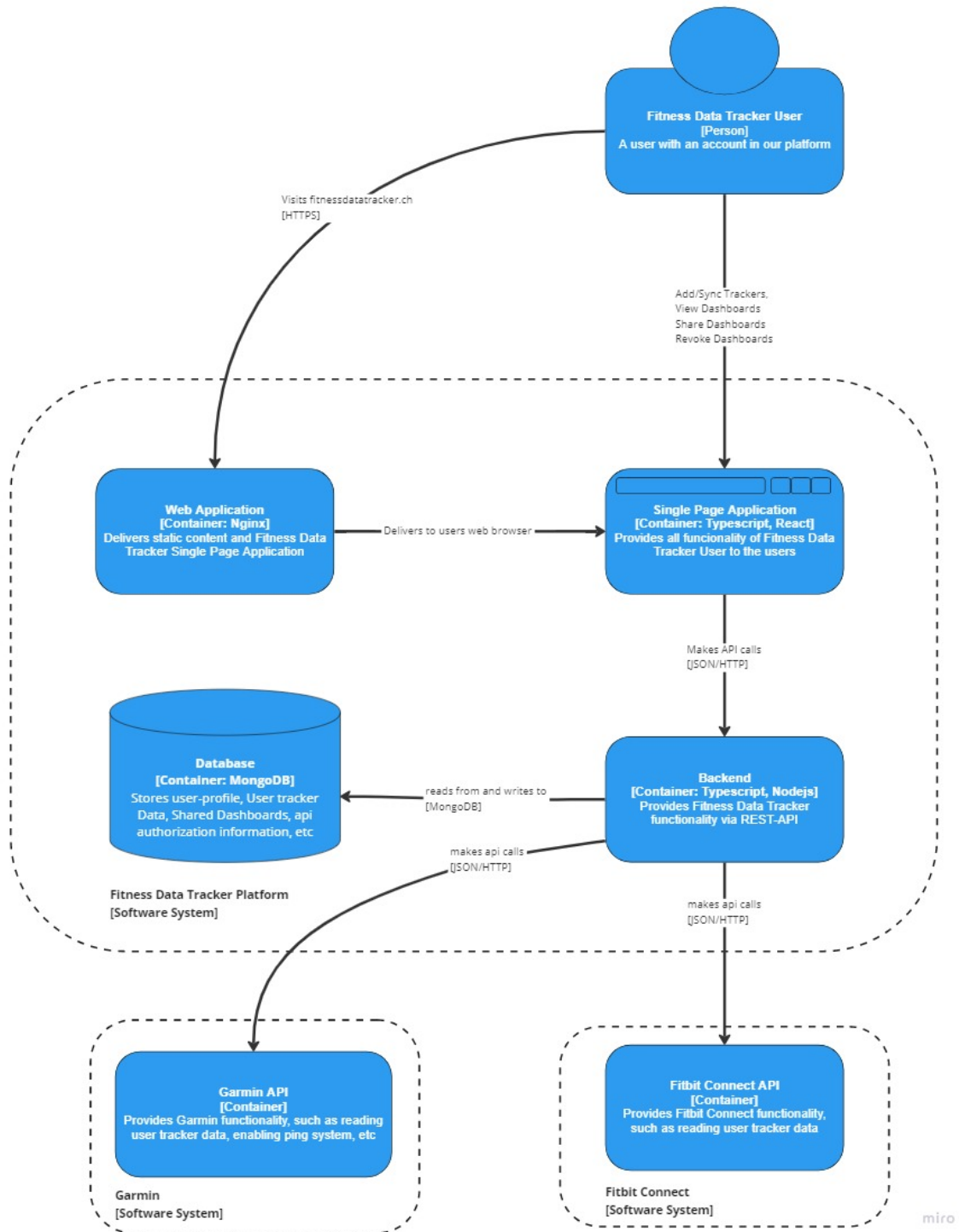


Figure 6.1: Container Diagram

6.2 Component Diagram

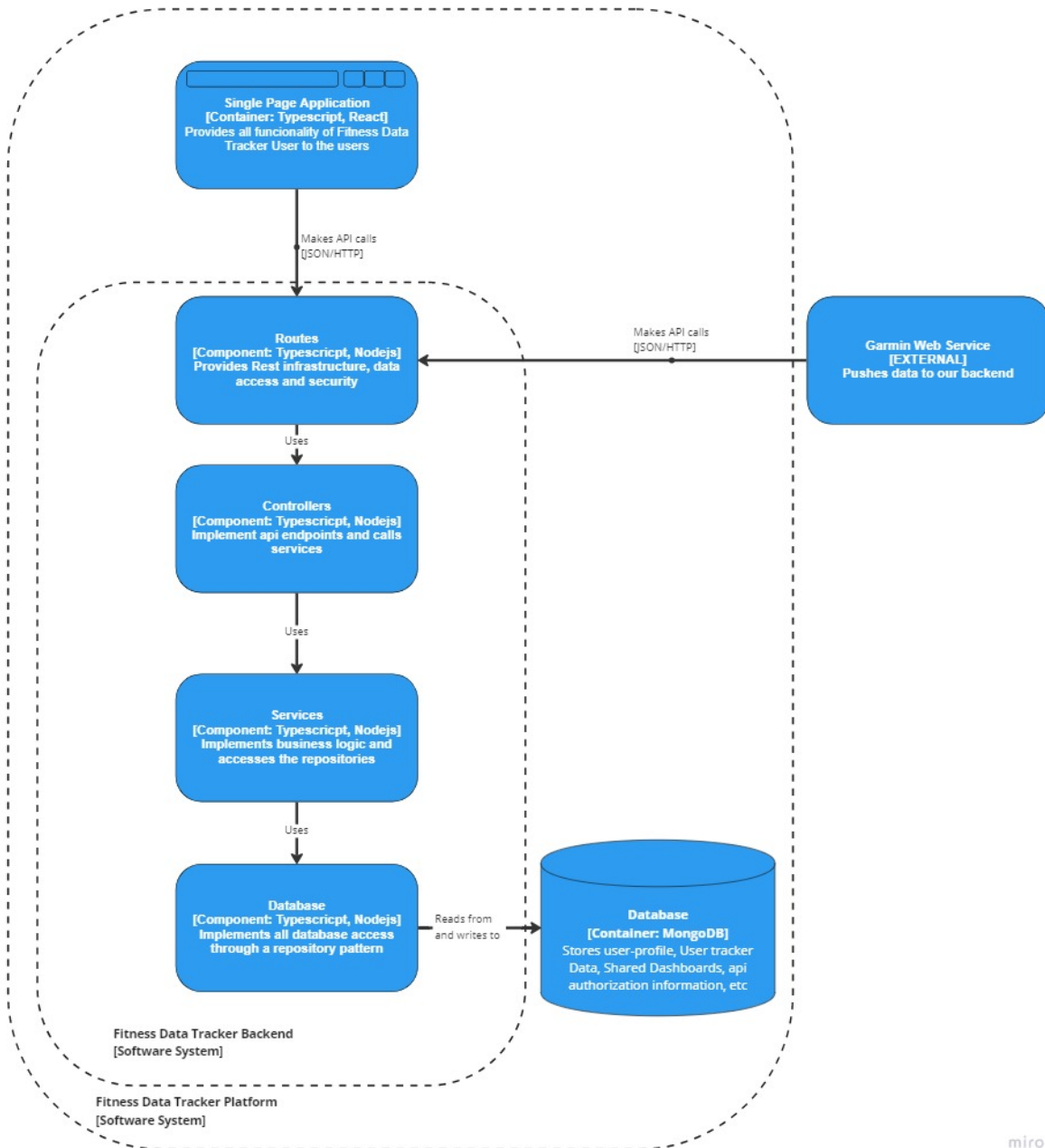


Figure 6.2: Component Diagram

Figure 6.2 represents the structure of our code. It is a generalization, and there might be deviations in the actual source code, which should be adjusted. We designed our application in a way that abstracts layers below and that components only communicate with the layer below them. For example, the user-controller.ts only uses the user-service.ts, which abstracts access to the repositories (database). This allows a distinction between API code which our frontend depends upon, and business logic. It allows us to change business logic without changing endpoint logic. For example, sometimes, a service function might only call one function from a repository. This is by design and allows changing of the service/repository logic without it affecting the controller-

s/routes directly. As visible, the only external service that uses our API is the Garmin web service. It pushes user data to our configured endpoint. Fitbit does not do that, but it should be extended to do so in the future.

6.3 Backend architecture

For our backend, we will use TypeScript with TSNode. Node.js, because it is a requirement by our client, we decided on TypeScript since we have experience with it. TypeScript allows for strongly typed programming, unlike JavaScript. The backend will be structured according to 6.3. This structure is commonly seen among Node.js projects, and we have previously applied similar structuring to our code.

- The config files contain configuration variables for connections and database-related things.
- Database folder contains all code related to connection and CRUD operations to our MongoDB database. It contains the connection client itself and all repository classes. It represents the Database component in the component diagram 6.2
- In the model's folder, we will create all objects we can model for this project. Excluded are data entries we receive from Garmin and Fitbit.
- Routes and Controller files will handle requests and responses from the backend to the frontend. Routes will configure the actual endpoints that can be reached, and the controllers will handle these requests. It represents the Routes component in the component diagram 6.2
- Services will implement the bulk of our business logic. They are responsible for establishing the authentication of our users with the external API's, normalizing the user data, manipulating data, and then calling the repositories to save the data to our database. It represents the Services component in the component diagram 6.2

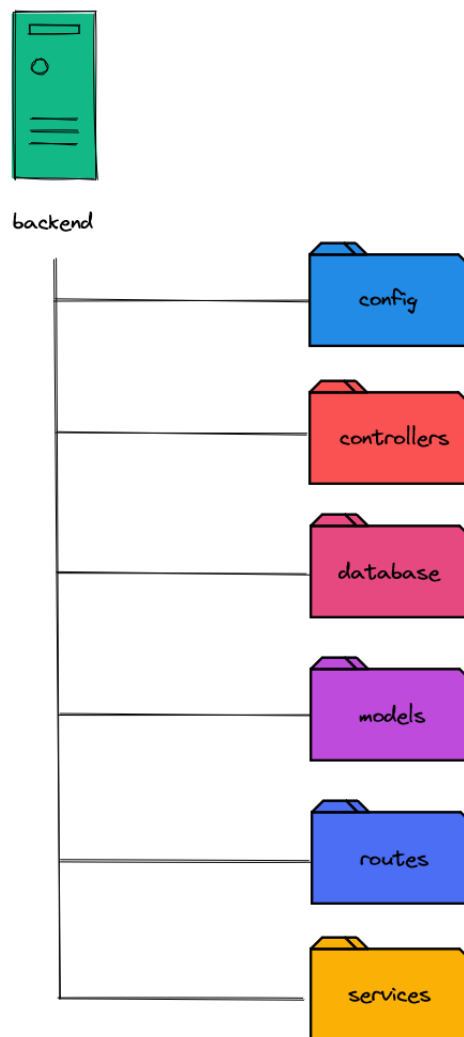


Figure 6.3: Backend folder structure

6.4 Frontend architecture

We use a lightweight, JavaScript-based frontend library with React. Therefore, a more extensive architecture is optional, but we must clarify certain internal architecture principles to keep the code consistent. React organizes itself into so-called components. Each component represents a logical part of the user interface. All components are considered equal, but there are a few worth mentioning:

- *frontend/src/index.tsx*: This is the root component and is the one that will always be present in our component structure. Once the Javascript is loaded, this component will be activated, and the remaining component tree will be initialized.
- *frontend/src/App.tsx*: This component is the only child of our root component. It contains important global configuration, and it has as its child the *RouteComponent.tsx*, which calls other components depending on the requested site. *App.tsx* is a "single page" that will be displayed to the user.
- *frontend/src/RouteComponent.tsx*: This component is responsible for deciding which components are displayed/called depending on the route.

- *frontend/src/common/apiClient.ts*: This file contains a very thin abstraction over the Axios library, which we use to do HTTP requests to our backend.

In previous versions, components were written as JavaScript classes. However, this requires much boilerplate code. So, for our project, we will use function-based React components. Those are much more compact and readable. React does not require implementing a specific folder structure, but for consistency, we will apply the following one:

- For each logical page, we create a folder in *frontend/src* and group all relevant components in there.
- Layout-related components like header and footer are stored in *frontend/src/components*.
- Things like our HTTP client to talk to the backend which are shared across multiple components are stored in *frontend/src/common*.

The diagram 6.4 visualizes how a request is handled internally, based on the example of when a user wants to see his own dashboard.

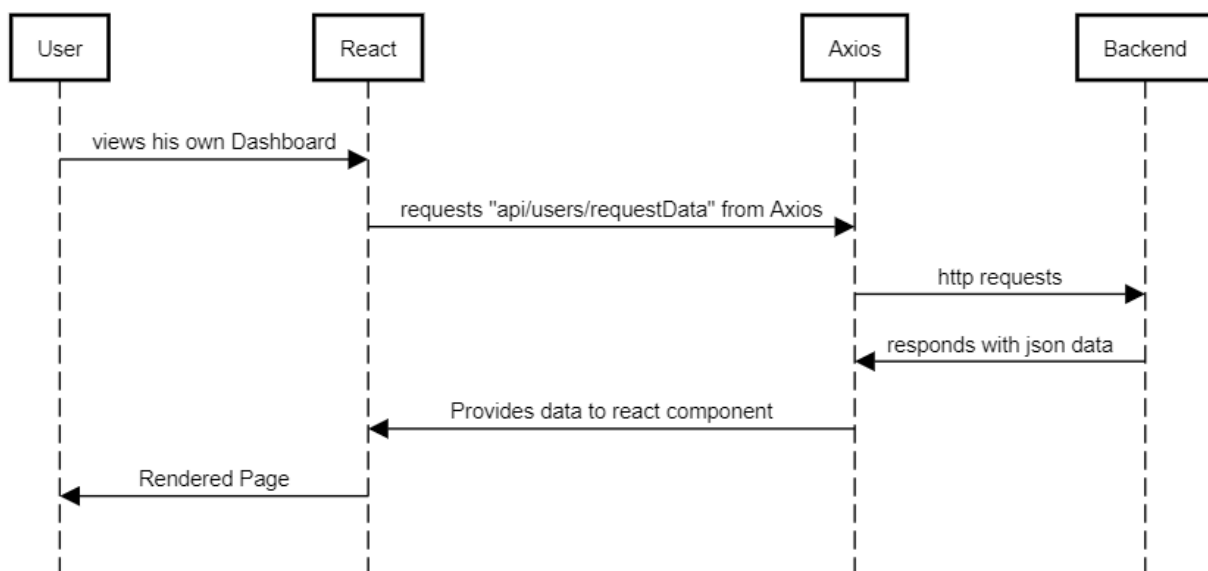


Figure 6.4: Request diagram frontend

6.4.1 UI Mockup

This section will show the initial draft at the beginning of the project. The UI design has since changed, but we will still include it to get a feeling of the evolution our design took over the course of the project. A general guide and inspiration for the UI was the WhatsApp Desktop app, so we tried a similar approach in designing our UI. This approach was suggested to us by our industrie partner and we tried for that advice to guide our design of the UI.

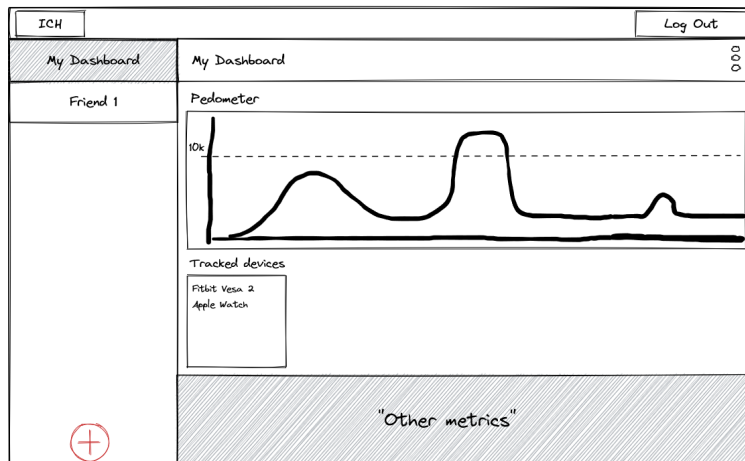


Figure 6.5: Dashboard main page Mockup 1

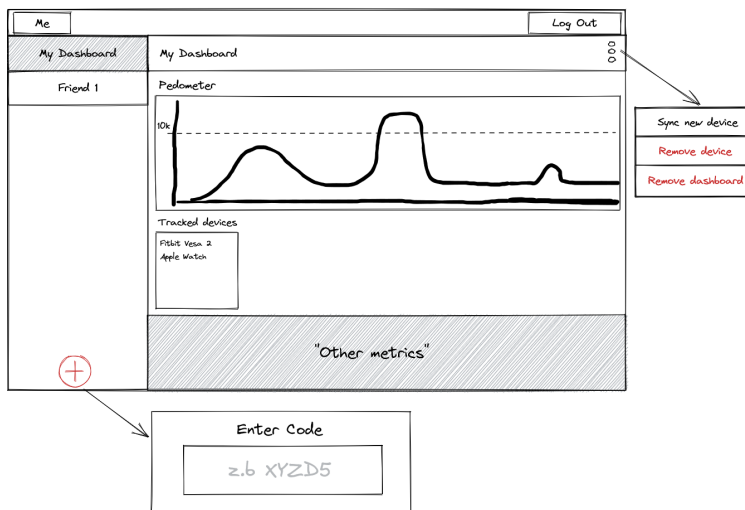


Figure 6.6: Dashboard Main Page Mockup 2

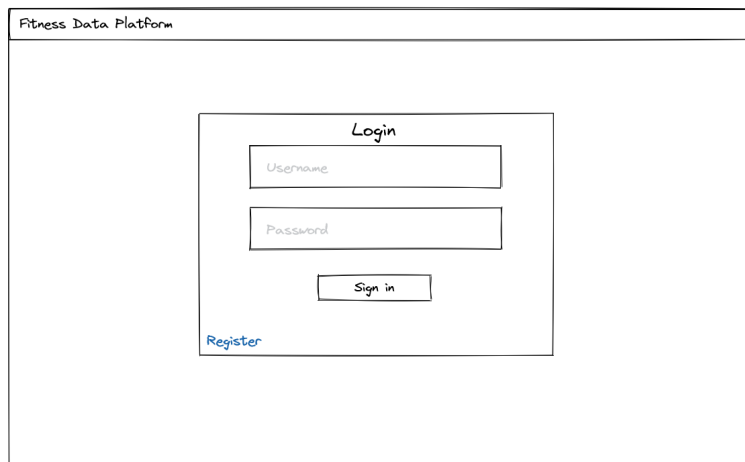


Figure 6.7: Login Page Mockup

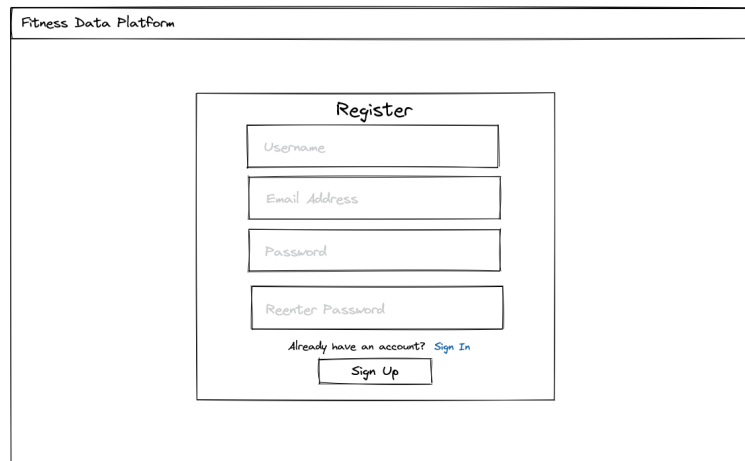


Figure 6.8: Register Page Mockup

6.5 Scaling

6.5.1 Scaling up

Since we use DigitalOcean Droplets to host our frontend, backend and database, scaling up is easy. We use Droplets which are servers to host both the backend and frontend and a managed Database cluster for our MongoDB. DigitalOcean allows scaling up the servers or database at any point in time. Just press resize and choose more compute power. We can change CPU power, memory, storage space, etc, on the fly with a click of a button. So upscaling the application is very manageable with DigitalOcean.

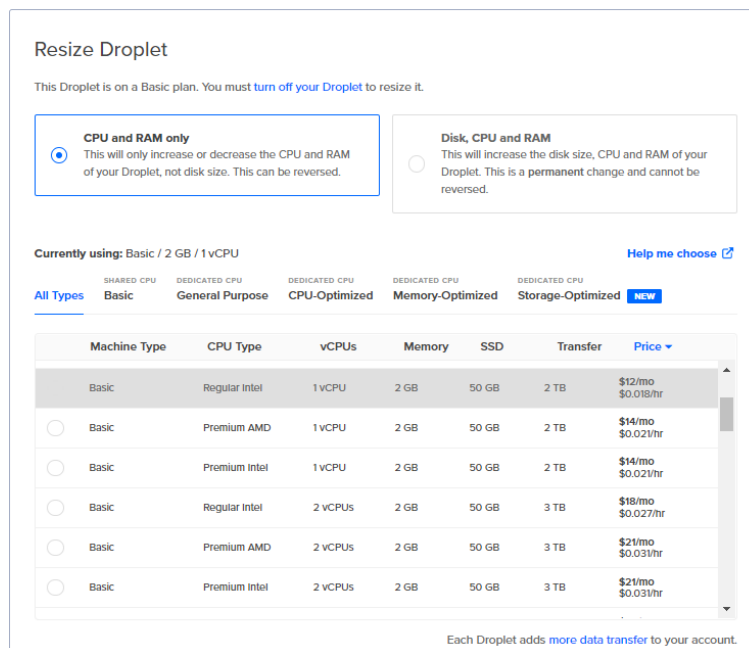


Figure 6.9: Register Page Mockup

6.5.2 Scaling out

We did not consider horizontal scaling, since this is currently a prototype and won't go into production in the duration of this project. It also wasn't mentioned as a requirement. Should the need for horizontal scaling arise, the following page should be examined [MongoDB scaling](#). The backend and frontend can easily be duplicated on different instances but data consistency has to be considered. For that please read the scaling article from MongoDB.

6.6 Deployment

We knew from the beginning that we were going to use [DigitalOcean](#) as our target for deployment since our client AdaptIT is already using it. We also chose [GitLab](#) to host our code. Now that we had the basics ready, we needed to choose how to best host and deploy our code from Gitlab to DigitalOcean. Again, we had a few options on how to host the code.

- We could create a kubernetes cluster and host our code there.
- We can dockerize our application and run the images on droplets..
- We clone the code onto a droplet, build it there and run the build files.

We looked into each option and even tried it first with a Kubernetes cluster, then a docker application on the droplet. We quickly realized that the most efficient way would be the third option. That way, there is no docker overhead, and DigitalOcean makes it very easy to use the third option. More detail about how exactly we deploy it will be in the CI/CD chapter 7.8. Since this product has yet to be in commercial use, we did not see the need for a staging/test environment, so we deploy to production every time. This is something that should be added before the go live.

6.7 Domain Model

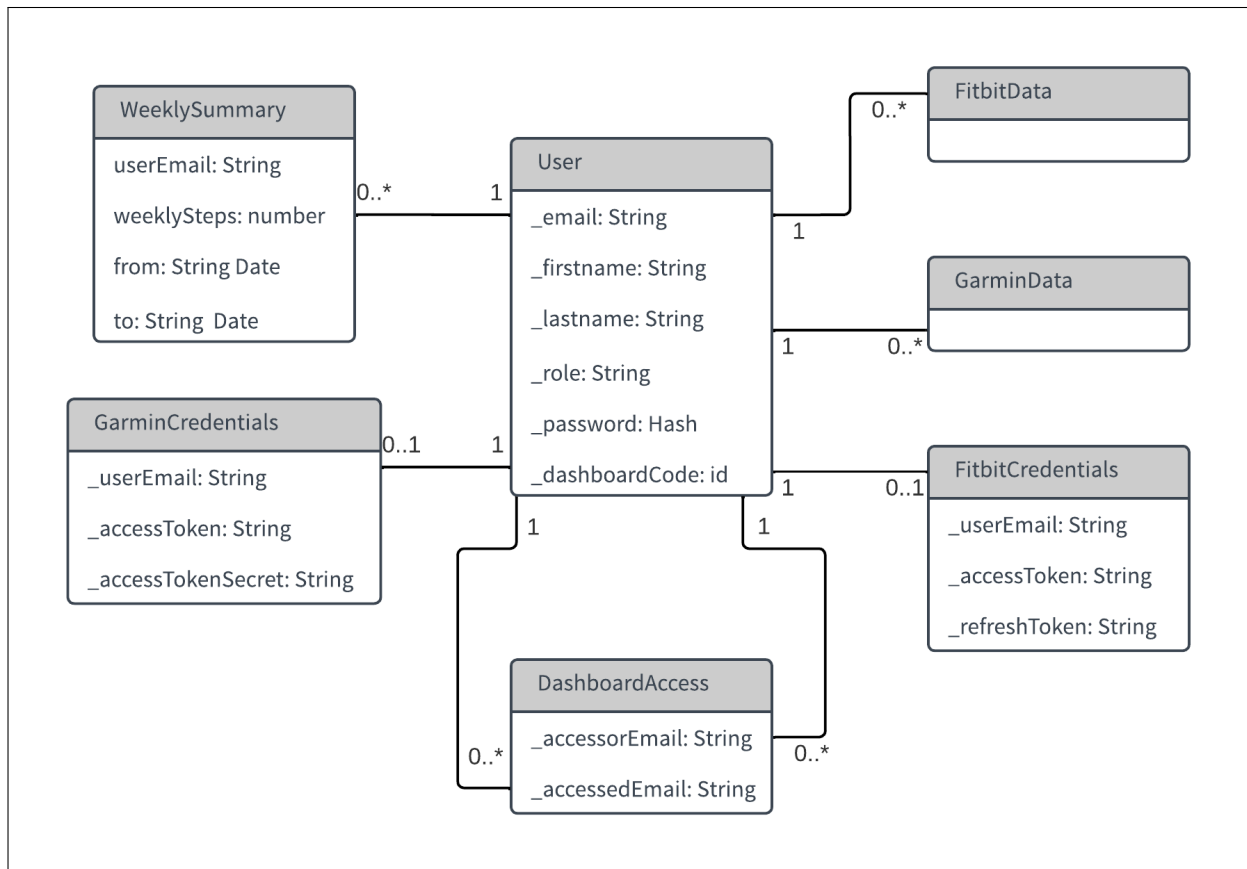


Figure 6.10: Domain Model

The graphic 6.10 both serves as a representation of the domain we are working in and also as a database model. For implementation details, please refer to chapter 7.6 At the center of it all is the **user**. **FitbitData** and **GarminData** are representations of the data we collect. They can (but do not have to) consist of multiple collections; for example, an activity collection of Garmin would be counted toward the GarminData modeled here. Modeling all the different collections, we will collect would clutter this diagram and not benefit the reader's understanding. Once the user allows us to sync his device, his credentials will be saved in one of the **credentials collections**. After that, his data will be regularly synced or pushed externally and saved into the various data collection. A significant decision was how the frontend would call and display the data. Since we first want to display a weekly summary of the steps, it would not be performant to calculate the weekly steps every time the user opens the dashboard. So we decided to "normalize" our data before we provided it to the frontend. This is represented here as the weekly summary, a normalized version of the steps contained in "FitbitData" and "GarminData". This way, data can be displayed without waiting on the backend calculations. Also, if we want any moment to display more data, we add a new entry in the summary or create a new summary, for example, a monthly one. Specifics about data normalization in 7.3. The model **DashboardAccess** is used to keep track of which user has access to other users' dashboard and intern their data.

6.8 Database

When we decided which database technology to use, we had to weigh two significant forces from the requirements. Performance and Flexibility. On the one hand, the technology we choose for our database should be as fast as possible to satisfy the NFR. However, on the other hand, it has to be able to store thousands of records and handle hundreds of queries simultaneously. We also needed the database to be flexible. It should store a multitude of unstructured data and be easily extended in the future. With these two forces, two Databases came to mind.

- [MongoDB](#)
- [PostgreSQL](#)

PostgreSQL is widely considered to be one of the fastest, if not the fastest, database out there. The only problem with PostgreSQL is that it is a relational database. That means you have to predefine the data structure you want to save. Since we have the goal to store as much data as possible from Garmin and Fitbit, and both come in different kinds of structures, there would have been a considerable overhead to define these structures. It would also increase the overhead cost for future extensions, for example, if we want to collect more data. The database is also at risk of breaking if the API endpoints of Garmin or Fitbit ever change in the future. For that reason, we decided that the performance boost of PostgreSQL was not worth the maintenance cost and initial implementation costs. We, therefore, decided on MongoDB. Even though it is slower, we can save the data from the API endpoints as they come since it is a NoSQL database and not worry about structure or modifying the data to fit our database. This reduces overhead considerably and eases future maintenance.

6.8.1 Redundancy

One of the NFRs was that the data from the database must be backed up at least once a day. The chosen hosting platform [DigitalOcean](#) makes this very easy. It allows restoring a database to a new cluster. It does not only make daily backups but allows to restore to pretty much any given time. DigitalOcean allows you to choose either the latest backup or an exact time, and DigitalOcean will create a new Database cluster and revert all transactions to the chosen time. If that is done, the previous cluster must be destroyed, and all connection details in the backend must be updated.

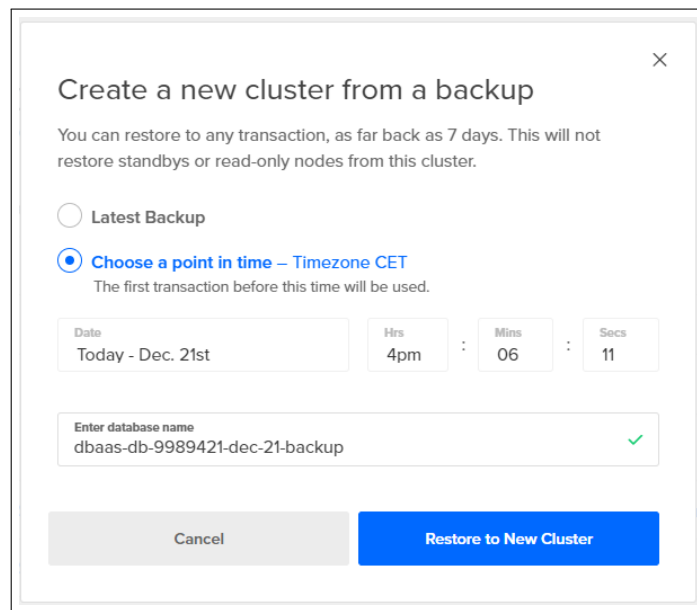


Figure 6.11: DigitalOcean Restore Dialog

6.8.2 Transactions and Rollback

One of our requirements was that when an error occurs, the system should be able to roll back to a previous functional state. This requirement was pretty much fully covered by [MongoDB](#) itself. As described [here](#), all instructions we use for MongoDB are atomic. Even "createMany()". This means we log the error when the instruction fails and our system is still functional. Furthermore, since we do not have consecutive inserts to the database that depend on each other, we just have to make sure that we catch and log the error. One exception to that rule was the requirement for a user to delete his account and all his data. We covered it by wrapping all actions in a transaction and rolling it back if they failed.

6.9 Security

Since the platform we are developing is not planned to go into production in the span of this project, security was not the most significant driving force. Still, we had a few considerations to make. One of these was how we save secrets and keys. For example, Garmin provides us with secrets and keys that we need to use to request their API. These should not be checked in the code. We decided to put these keys and secrets into a .env file that is located in our backend server. This file gets loaded when the application is deployed, and the keys and secrets get loaded into their respective "*.config.ts" files. We decided against putting the secrets into CI/CD variables and then dynamically loading them into the .env file every time the application is deployed. The reason for that is because of previous work experience where developers could adjust the pipeline but did not need access to secrets. This way, the secrets are only present in one place, on the servers themselves, which only authenticated personnel should have access to.

6.9.1 HTTPS and Domains

Any modern app should communicate with HTTPS. That is why we created a certificate with certbot for our frontend and backend. The frontend certificate was created for the "fit.adaptit.ch/"

domain and backend for "fitbackend.adaptit.ch/" domain. These domains were provided to us by our industry partner Michael Güntensperger from AdaptIT.

7 Implementation

This chapter will describe the most important implementation details and implementation aspects of our project.

7.1 Application Programming Interface

This section describes the implementation of the API's for Fitbit and Garmin.

Fitbit Web API

The Fitbit Web API was the first interface we implemented for our project. This API required multiple steps before we could request data. First, we had to create a developer account and go through a verification process. Secondly, we had to register our application to retrieve a client ID and secret, which we would need to authorize the user through our application. For the communication with the API a OAuth 2.0 is used. In order to keep the Fitbit user data secure, our application has to generate two specific values. The **code verifier** and a **code challenge**. The code verifier is a cryptographically random value between 43-128 characters. The code challenge is the hashed version of the code verifier with SHA-256 and a base64url encoding with padding omitted. The

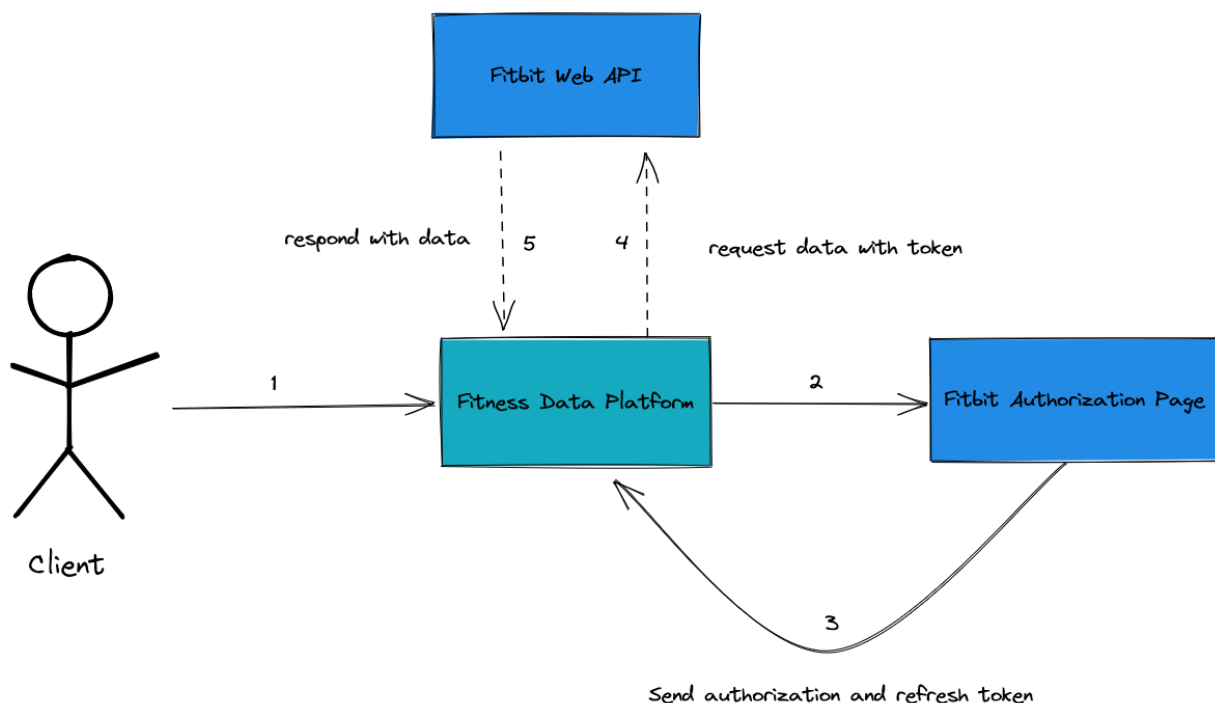


Figure 7.1: Fitbit Web API Workflow

figure 7.1 displays a simplified version of our workflow with the Fitbit Web API. After the client approves our application through the authorization page of Fitbit, access and refresh token will

be sent back as a response with which we can request the client's data from Fitbit. The refresh token is used for requesting a new access token after the current one expires. After authorizing our application through Fitbit, the only thing left was to request data about the user from Fitbit and persist those in our database. We do this by running a scheduled cronjob with [node-cron](#). The schedule for this cronjob can be adjusted in "src/config/scheduled.tasks.config.ts". Currently, it is set to run each day at 23:00. For simplicity and future purposes, we request that our application have access to all user data, even though we currently do not need all of it. This is done since we collect more data than we display and want to extend our application in the future. We registered our app for fitbit [here](#). All information regarding specifics for authorization or data collection can be viewed once an application is registered. Fitbit makes these documentation very accessible. For implementation details refer to the source code.

Garmin API

This section will describe how we decided to collect user data from their Garmin device. Garmin offered two distinct options that would cover our needs for this project, and are listed in the [Garmin developer page](#)

- Collecting FIT files through the [FIT SDK](#)
- Using the [Garmin Connect](#) api's to collect data

We decided to use the Garmin connect developer program and collect the data through the Garmin connect API. This has the advantage that in the case that a user has multiple Garmin devices, Garmin connect will automatically merge the data of the two, so there is no need on our end to merge or even know the amount of Garmin devices a user might have. Before we go into detail, it has to be mentioned that Garmin has a strict privacy policy, and most of its documentation is strictly confidential and cannot be shared. Therefore, we will only go into a little technical detail about their documentation and our implementation but will provide a general overview of our integration of Garmin Connect.

App Registration

Before we could receive any user data, we had to register our application in the Garmin connect developer program. Our Industry partner from AdaptIT did this. We could redirect our user to the Garmin Connect page with the provided authorization details. There the user has to log in and authorize our application. After that, the user gets redirected back to our frontend page, and we can read the user tokens from the URL and send them back to our backend.

Integration Options

Nevertheless, we had to choose a **integration option** for Garmin before we could collect any data. Next, we chose a **push configuration**. Garmin will send newly generated or updated data to predefined endpoints with this push configuration. We defined these endpoints in the Garmin endpoint configuration page for our application. Now we can receive each new entry or updated entry from our user.

Collected Data

We currently do not collect all possible data from Garmin, but limited it to two types of information:

- **Garmin Daily Health Summaries:** Those are daily summaries from the user that contain much information gathered from the device or devices the user has. From calories burned to steps taken that day, active seconds where the user actively walked around or did other activities, rest time, etc
- **Activity data:** Those are activities that the user either manually entered or manually started on his own. Those range from runs to hunting, skydiving, etc.

These two types of information we collect cover all of our collection requirements, with the focus being on the Garmin Daily Health Summaries since these contain all the information we need. We still decided to collect Activities, too, since they are a big part of Garmin and can provide interesting information from the user, which can be used in future extensions of our application. Now we can receive newly created or updated entries from the user. However, it would also be beneficial to collect data from the user before the user authorizes our app in Garmin connect. This requirement can be fulfilled by sending a backfill request for a datatype and a time range. The user data in this time range will be flagged as new or updated by Garmin and, therefore, will

be pushed to the same endpoint as newly created or updated user entries. With this configuration we described, we could receive and request the needed data from the user. In figure 7.2, you can see the workflow process we describe in detail above visually represented in a diagram

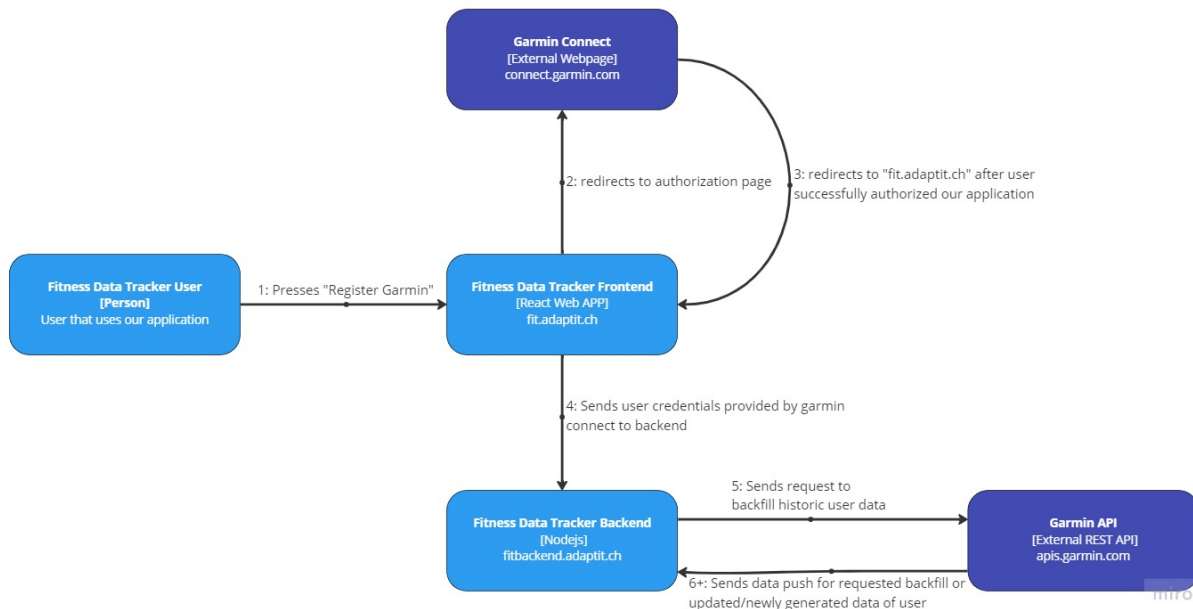


Figure 7.2: Garmin API Workflow

7.2 Dashboard Sharing

One of the initial requirements was for a user to be able to share his dashboard, and view dashboards that were shared with him. The user should also be able to revoke dashboard access. This section will go into some detail on how we decided to implement this requirement in our application. But for specific implementation details see the attached source code.

7.2.1 Share Dashboard

There were multiple options for a user to share his dashboard and give another user access. Some common methods are:

- **Access Code:** A user has an access code that is either generated every time the dashboard is shared or is a static code. This code can be shared with other users. The User that has an access code to another user can enter this code and gain access to the user's dashboard
- **Link:** Every User has a unique link. If another user visits that link, this User gains access to the other User's dashboard
- **User selects other users that should have access:** The User can choose/enter which users should have access to his dashboard. That means no action is needed on the user's end that gets a dashboard share with him.

The choice was between the "Access Code," and "Link" since the third option does not allow a user that gets shared a dashboard to ignore it. So we chose the "Access Code" option. We honestly

were torn between those two options but chose the "Access Code" for ease of implementation. Most of the time you see these two options combined. For example, in [Discord](#) you can share a server with a code or link.

Implementation Details

To allow the sharing of a dashboard with an access code, a few things had to be done. First, we had to associate a unique code to a user. Another user will use that to access the dashboard. We had to add a code to the User model:

```
export class User implements {_id: ObjectId} {
  ...
  private readonly _dashboardCode: number //new field for access code
  ...
  constructor(email: string, firstname: string, lastname: string, role: string, password:
    ...
    this._dashboardCode = uniqid() //gets uniquely generated when a user is initialized
  }
  ...
}
```

We used the [uniqid](#) package to generate a unique code. This code will always be unique since it generates the code based on the time, place, and machine on which it is generated. Now that we have a unique access code, we can use this code to give a user access to another user's dashboard.

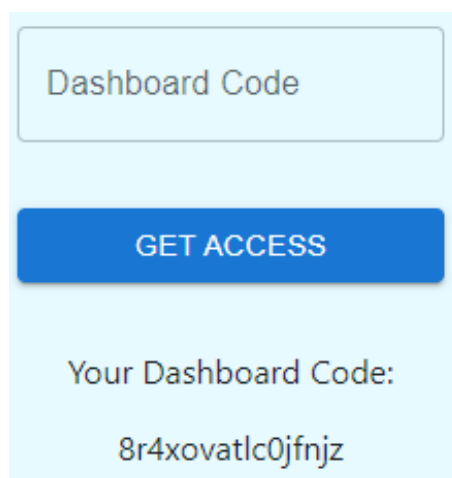


Figure 7.3: User Access Code Frontend Screenshot

These UI elements allow a user to view his code and enter a code from another user. Now let us go through what happens if a user enters another user dashboard code.

1. User enters code and presses "GET ACCESS." This submission does a post request to the backend with URL "<baseUrl>/api/users/requestDashboardAccess". We pass the current user jwt token to the backend and the entered user code in the body.
2. When the Backend receives this request, it creates a new entry in MongoDB's "dashboard-Access" collection. This entry has both the accessor's email and the accessed user's email. This entry will be used to determine if a user has access to a specific dashboard.

7.2.2 Switch Dashboard

This section describes how a user can switch between dashboards and view other users' dashboards.

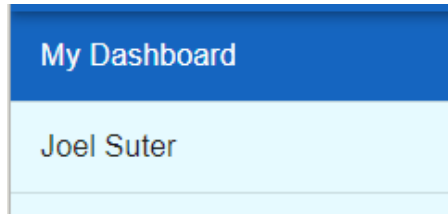


Figure 7.4: Accessed Dashboard Frontend Screenshot

In the above screenshot 7.4, an example is displayed where a user has two dashboards listed. "My Dashboard" represents his Dashboard and data. The "max muster" entry is a dashboard that this user can access. This list of accessed Dashboards gets returned by the backend when the user opens the home page of our application. The endpoint that returns this list is "<baseUrl>/api/dashboard/u". The code the backend runs is as follows:

```
exports.getAccessedUsers = async (req: express.Request, res: express.Response) => {
  ...
  try {
    await dashboardAccessRepository.getAccessedUsers(accessorEmail).then((emails) => {
      emails.forEach(async (email) => {
        await userRepository.findDashboardCodeByUserEmail(email).then((user) => {
          menuItems.push({
            route: "/users?accessCode=" + user.dashboardCode,
            owner: user.fullName
          })
        })
      })
      res.send(menuItems)
    })
  } catch (e) {
    log.error("Something went wrong while accessing database")
  }
}
```

The backend returns a list of "menuItems". These items have two properties.

- **route:** This is the route that will be used to access the data of this dashboard.
- **owner:** Both the name and last name of a user. Will be displayed in the frontend entry

The user can now select another dashboard he has access. In this example, he can switch to the dashboard of "max muster." When he does this, the URL changes to "<domain>/users?accessCode=<accessCode>". This triggers our route component:

```
<Route path="/users" element={<DashboardPage />} />
```

This route component will rerender our DashboardPage, which will try to read the accessCode from the URL.

```
function DashboardPage() {
  const { search } = useLocation();
  const accessCode = new URLSearchParams(search).get('accessCode'); //reads code

  return (
    <Layout>
      <DisplayGraph accessCode={accessCode} /> //passes it to LineGraph
    </Layout>
  );
}
...
```

If the LineGraph is instantiated, he will do one of the two queries to fetch data.

1. If no accessCode is present, LineGraph will fetch data from "<baseUrl>/api/users/requestData".
2. If an accessCode is present, LineGraph will fetch data from "<baseUrl>/api/users/requestData?accessCode=...".

As seen above, LineGraph will request data from the same endpoint with or without an accessCode. The backend will check if an access code is provided. If one was, he would return data for the user accessCode belongs to. If not, he will return data from the user that sent the request, which is read from the jwt token provided.

```
exports.requestData = async (req: express.Request, res: express.Response) => {
  if(!validator.isUserValidated(req, res)) {
    return
  }

  const userDashboardCode = req.query.accessCode?.toString()
  const jwtToken = req.body.token
  const userEmail: string = getJWTEmail(jwtToken)

  if(userDashboardCode) {
    const accessedEmail = await userRepository.findUserEmailByDashboardCode(userDashboardCode)
    const access = await dashboardAccessRepository.isAccessGranted(userEmail, accessedEmail)
    if(access) {
      getAllWeeklySummaries(accessedEmail).then((weeklySummaries) => {
        return res.send(weeklySummaries)
      })
    } else {
      return res.status(403).json({errors: "You don't have access to this repository"})
    }
  } else {
    getAllWeeklySummaries(userEmail).then((weeklySummaries) => {
      return res.send(weeklySummaries)
    })
  }
}
```

7.2.3 Revoke Dashboard Access

A user is able to see who has access to his dashboard and is able to revoke that access.

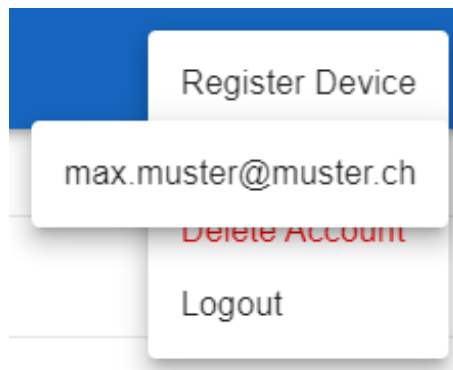


Figure 7.5: Revoke Dashboard Access

If the user revokes access to his dashboard to from a certain user, the entry in the "dashboardAccess" collection will be removed and the revoked user will no longer be able to view this users data.

7.3 Data Normalization

The topic of data Normalization was already addressed in the domain model section but will be highlighted here again. Since we collect data from Fitbit and Garmin, we decided to normalize this data before it is served to the dashboard. The reason for that is performance. Calculating weekly steps, going through the whole database, and weighing both Garmin and Fitbit data every time a user opens a dashboard would cause too much load on the backend and long loading times for the user. So we decided on data normalization, which allows the backend only to fetch a few records of data when a dashboard is opened instead of having to do calculations every time. Every time the backend is started, the "src/index.ts" script gets to run, which initializes the backend. This file also calls the "createScheduledWeeklySummaryJob()" from the "src/services/scheduled/scheduled-tasks-service.ts". This method initializes a cronjob which runs every day at 23:00. The time this job runs can be defined in the "src/config/scheduled-tasks.config.ts" file. Every time the cronjob is run "generateWeeklySummaries(email)" function from the "src/services/data/data.service.ts" gets to run for every user in our system asynchronously.

The method where this cronjob is initialized is implemented as follows:

```

/**
 * Creates a scheduled cronjob to generate Weekly summary for all users (Data Normalization)
 */
export async function createScheduledWeeklySummaryJob() {
  cron.schedule(scheduledTasksConfig.GENERATE_WEEKLY_SUMMARY_SCHEDULE, async () => {
    const userEmails: Array<string> = await userRepository.getEmailsOfAllUsers()
    userEmails.forEach((email) => {
      generateWeeklySummaries(email).then(() => {
        log.info("Completed generating of weekly summary for user: " + email)
      }).catch((err) => {
        log.warn("Error trying to generate weekly summary for user: " + email, err)
      })
    })
  })
}

```

We are able to do scheduled tasks with the [node-cron](#) package. Weekly summaries for a user are generated as follows

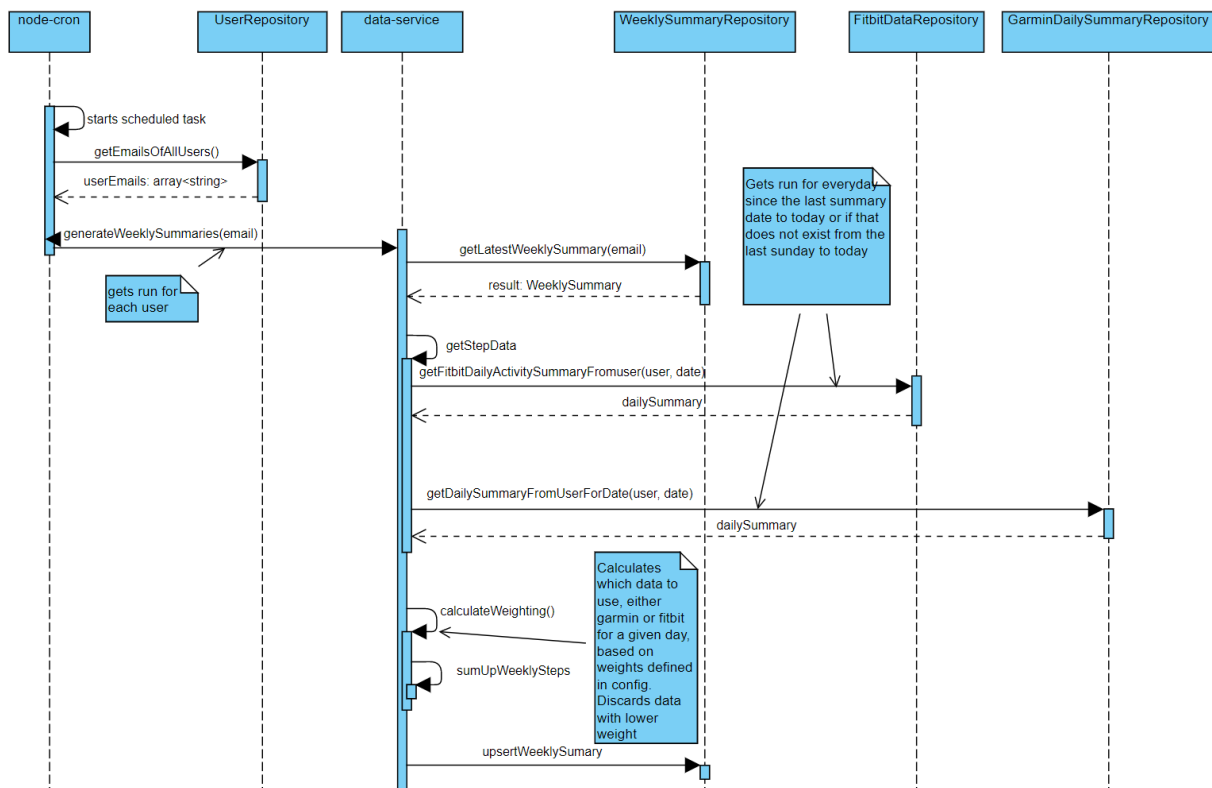


Figure 7.6: Generate Weekly Summary Sequence Diagram

Remember that this Sequence Diagram 7.6 is not an exact representation of our source code but a general overview of how weekly summaries are generated, and can be used to more easily interpret the code that implements this. As you can see in the sequence diagram, node-cron starts a scheduled job. First, it retrieves all our users, then loops through them and calls the method "generateWeeklySummaries" for every user. First, the latest summary data is retrieved. Depending on this date, the algorithm will run differently. There are three different cases:

- **No previous summary exists:** Summaries will be generated from the last Sunday to today
- **Latest summary lies in the current week:** The summary will be updated with new daily summaries from the current week.
- **The latest summary is one or multiple weeks old:** The latest summary will be updated, and gaps in weekly summaries will be filled. The current week will also be updated.

After the latest weekly summary is pulled, the algorithm will pull Daily summaries from both Garmin and Fitbit that are saved in our database. It will pull these for either the latest summary to today or if that does not exist, for the last Sunday to today. After that, we calculate which data we want to use for our weekly summary. This is done by weighing the Garmin data against the Fitbit data and choosing which data to use for that given day. Specifics on how we decided to weigh Fitbit steps and Garmin steps are described in 7.3.1.

- **Scope:** Currently we only summarize the step count, and sum up the data to create a weekly summary.

- **Extend Weekly Summary:** This algorithm can be easily extended to summarize more than just the weekly step count. It could also summarize the heartrate or other data that we currently collect. For that two things have to be done:
 1. **Scope:** Currently, we only summarize the step count and sum up the data to create a weekly summary.
- **Extend Weekly Summary:** This algorithm can be easily extended to summarize more than just the weekly step count. It could also summarize the heart rate or other data that we currently collect. For that, two things have to be done:
 1. Add weights for the new data that should be summarized
 2. Collect the wanted data (same as weekly steps), weigh them against each other, and write them into the weekly summary as a new field.
- **Daily/Monthly/Yearly Summary:** Data normalization could also be extended to generate daily, monthly, or even yearly summaries. For that, a similar algorithm has to be implemented as it already exists. In addition, the range of data written into the summary has to be adjusted, and a new collection has been added to the database where this summary will be written. But generating these summaries will not differ in logic too much, and this code can be used as a guide or refactored to extend to new summary types.

7.3.1 Weighting Algorithm

This chapter describes how the weighting algorithm works in our application. After implementing the Fitbit and Garmin API, we had to persist the data in our database and evaluate a weekly summary. For the weighting algorithm to work, we had to set a weight for every brand. In the paper [1], thorough research was done regarding the accuracy of different manufacturers. Through this paper, we found out that Fitbit watches are more accurate in step count than Garmin watches. Garmin watches, on the other hand are more accurate in measuring heartbeat. Because in this project, we only summarize the weekly step count, we made a simple weighting rank for Fitbit at 0.8 and Garmin at 0.6. These weights can be adjusted in "src/config/algorithm.config.ts". The algorithm requests all data from the current week and compares entries of the same date with each other. Any data with a higher weight will be prioritized, and if no data exists for this date, a step count of 0 will be added. Future implementation ideas include considering the actual step count and how long the watch was worn throughout the day. Depending on the worn length, the weight of the data would change.

7.4 Logging

For logging, we used a library called tslog. Very simple library with different levels for logging and extensive information. We primarily used info logs for high-level information and error logs for any faults in the database or API.

```
import { Logger } from "tslog";
const logger = new Logger({ name: "myLogger" });

logger.silly("I am a silly log.");
logger.trace("I am a trace log.");
logger.debug("I am a debug log.");
logger.info("I am an info log.");
```

```
logger.warn("I am a warn log with a json object:", { foo: "bar" });  
logger.error("I am an error log.");  
logger.fatal(new Error("I am a pretty Error with a stacktrace."));
```

7.5 Code Documentation

We have documented our TypeScript code for readability and future purposes. As a standard, we used TSDoc, a proposal for a standardization for TypeScript documentation. The documentation follows this principle:

```
export class Statistics {  
  /**  
   * Returns the average of two numbers.  
   *  
   * @remarks  
   * This method is part of the {@link core-library#Statistics | Statistics subsystem}.  
   *  
   * @param x - The first input number  
   * @param y - The second input number  
   * @returns The arithmetic mean of 'x' and 'y'  
   *  
   * @beta  
   */  
  public static getAverage(x: number, y: number): number {  
    return (x + y) / 2.0;  
  }  
}
```

In the backend, we have documented about 80% of our codes. The 20% left is mostly code snippets which are auto-generated by the IDE or code snippets for which documentation would be unnecessary.

7.6 Database

The topic Database was already addressed in a previous chapter 7.6. There it was described why we chose MongoDB and how we designed our database structure. This section will describe concrete implementation details and the connection to our backend. As mentioned, we chose MongoDB as our Database. We host this Database on [DigitalOcean](#). It runs on a separate cluster. While setting up the Database, we created a new user and password and secured trusted sources that can access the Database. For the moment, the only trusted sources are "fitbackend.adaptit.ch" where our backend runs, and our development computers, so that we can remotely connect to the Database. Local development is still done on a local instance of MongoDB, but remote access was still needed for testing and development purposes. The question now was how we used this database in our backend. The first option we found was with [mongoose](#). This is a prevalent option, but in our case, we found it does not apply. Mongoose requires you to define schemas for collections. One of the requirements we set ourselves, and why we chose mongodb, was that we do not have to model our collections. Since we want to collect data from Fitbit and Garmin without the need to model them and be able to save them unstructured, for

that reason, we chose another option which we found [here](#). We implemented a "generic repository pattern". We had to define a "BaseRepository" which implements functions that are used across all our repositories:

```
import {Db, Collection, ObjectId} from "mongodb"

export abstract class BaseRepository<T extends { _id: ObjectId }> implements IWrite<T>, IRead<T> {

  public readonly _collection: Collection

  constructor(db: Db, collectionName: string) {
    this._collection = db.collection(collectionName)
  }

  /**
   * Creates an item in the database.
   *
   * @param item - item to be persisted
   * @returns result - boolean if successful or not
   */
  async create(item: T): Promise<boolean> {
    const result = await this._collection.insertOne(item)
    return !!result
  }

  /**
   * Creates multiple items in the database.
   *
   * @param item - an array holding multiple items to be persisted
   * @returns result - boolean if successful or not
   */
  async createMany(item: Array<T>): Promise<boolean> {
    const result = await this._collection.insertMany(item)
    return !!result
  }

  ...
}
```

This base repository implements two interfaces itself, one responsible for write operations:

```
export interface IWrite<T> {
  create(item: T): Promise<boolean>;
  update(id: string, item: T): Promise<boolean>;
  delete(id: string): Promise<boolean>;
}
```

And the other responsible for read operations:

```
export interface IRead<T> {
  find(item: T): Promise<Array<T>>;
  findOne(id: string): Promise<T>;
}
```

This BaseRepository is then inherited from every collection we have. Each collection is only accessed by one repository which implements and or overrides method from the BaseRepository depending on the need. One of the implemented repositories will be shown here:

```
export class UserRepository extends BaseRepository<User>{

  async getEmailsOfAllUsers(): Promise<Array<string>> {
    const userEmails = []
    const cursor = await this._collection.find()
    await cursor.forEach((user) => {
      userEmails.push(user._email)
    })
    return userEmails
  }

  async findUserByEmail(userEmail: string): Promise<User> {
    const result = await this._collection.findOne(
      { _email: userEmail }
    )
    return result ? new User(result._email, result._firstname, result._lastname, result.
  }

  async findUserEmailByDashboardCode(dashboardCode: string): Promise<string> {
    const result = await this._collection.findOne(
      { _dashboardCode: dashboardCode }
    )
    return result ? result._email : undefined
  }

  async findDashboardCodeByUserEmail(userEmail: string): Promise<{ fullName: string, dashb
    const result = await this._collection.findOne(
      { _email: userEmail }
    )
    return result ? {fullName: result._firstname + " " + result._lastname, dashboardCode
  }
  ..
}
```

As seen above the repository implements the BaseRepository for the type user. In this case we do not overwrite the inherited methods since the default implementation suffices. But we extend the repository with numerous functions that our services need. The user Repository accesses the collection "user" from our mongoDB.

To see a model of the different collections please refer to the domain model 6.7. All collections are listed there except the garmin and fitbit data collection.

7.7 Displaying of Data

This section covers how the data is displayed in the frontend, and with data, we mean the weekly steps a user takes. It will not cover data collection 7.1, data normalization 7.3. Only how the data is pulled from the backend and displayed on the frontend. For the displaying of the weekly step count, we chose the package [chart.js](#). This package allows for displaying of data in various

forms and diagrams. Furthermore, it allows the dashboard and the current LineGraph to be easily extended in the future with further options like selecting the type of data that should be displayed, time range, etc. The function in our React app that gets called if the graph should be displayed is implemented as follows

```
...
const fetchData = (code: string | null) => {
  if (code) {
    authenticatedApiClient(jwt!).get<Array<any>>(`/users/requestData?accessCode=${code}`)
      // @ts-ignore
      .setData(response.data);
  }).catch((error) => {
    console.log(error);
  });
} else {
  authenticatedApiClient(jwt!).get<Array<any>>(`/users/requestData`)
    .then((response) =>
      // @ts-ignore
      .setData(response.data);
    ).catch((error) => {
      console.log(error);
    });
}
};
...

const stepData = {
  datasets: [
    {
      label: 'Weekly steps',
      data: data || [],
      borderColor: 'rgb(255, 99, 132)',
      backgroundColor: 'rgba(255, 99, 132, 0.5)',
    },
  ],
};
...

```

First, the "fetchData" function had to be implemented. This method makes a get request to our backend, either with an accessCode from another user's dashboard, if the data of this user should be displayed, or without one, with the data of the currently logged-in user should be displayed. Depending if this query parameter has been set, the backend will return different data. Lastly, the data gets set in a reacted state. If this state gets updated, the graph gets rerendered. This state is then used to build the "stepData", with some additional information on how the graph should look and what the title should be.

```
useEffect(() => {
  const { accessCode } = props;
  // @ts-ignore
  setParam(accessCode);
  fetchData(accessCode);
  // eslint-disable-next-line react/destructuring-assignment
}, [props.accessCode]);

```

If a user switches to another user's dashboard, the property "props.accessCode" will change. This will trigger this "useEffect" function, which will set the accessCode and rerun fetchData. All UI elements depending on these values will be rerendered with the new values.

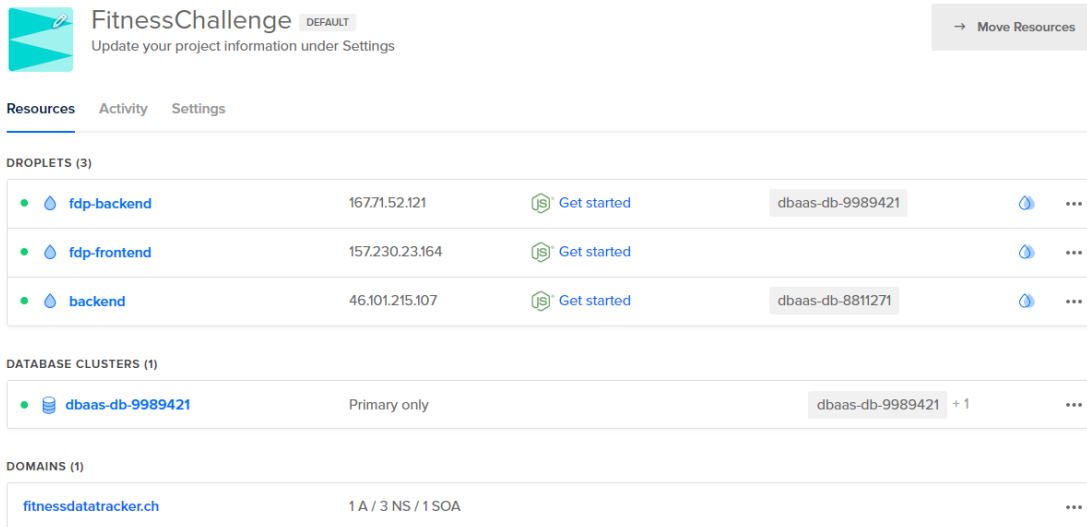
```
return (  
  // eslint-disable-next-line react/jsx-no-useless-fragment  
  <>  
    {data ? (  
      <>  
        <div id="chart-wrapper">  
          <Line  
            data={stepData}  
            options={{  
              parsing: {  
                xAxisKey: 'date',  
                yAxisKey: 'steps',  
              },  
              responsive: true,  
            }}  
          />  
        </div>  
        <p>{param}</p>  
      </>  
    ) : (  
      <div>Loading...</div>  
    )}  
  </>  
);
```

Lastly the function returns the UI elements and the chart-js graph to the component that called that tries to render this function.

7.8 CI/CD and Hosting

This chapter will go in depth on how we chose to implement CI/CD in our project. It will however not go into detail of our general infrastructure. We created two [Node.js Droplets](#) one for the frontend and one for the backend and a managed MongoDB database cluster that is accessible from these droplets. To be able to deploy our code to these droplets, we had to prepare a few things. First, we needed to manually clone the code to the droplets and checkout the main branch. This has only to be done once. The rest of the deployment will be automated. The Nginx file also needed reconfiguring to point all traffic to that droplet to our frontend and backend applications. Then we had to save a private key in our Gitlab and the public key on the droplets, so that we could ssh into them. And finally, we wrote to gitlab-ci.yaml. It consists of two stages, test and deployment. If the tests are successful, we deploy our code. For both backend and frontend the process looks like this: First, we pull the new changes into the droplets, then we build the code and finally, we serve the code with [PM2](#). Since they are both Node.js application, they are similar for deployment.

```
image: node:7.7.0
```



FitnessChallenge DEFAULT [→ Move Resources](#)

Update your project information under Settings

Resources | Activity | Settings

DROPLETS (3)

fdp-backend	167.71.52.121	Get started	dbaas-db-9989421	...
fdp-frontend	157.230.23.164	Get started		...
backend	46.101.215.107	Get started	dbaas-db-8811271	...

DATABASE CLUSTERS (1)

dbaas-db-9989421	Primary only	dbaas-db-9989421 + 1	...
------------------	--------------	----------------------	-----

DOMAINS (1)

fitnessdatatracker.ch	1 A / 3 NS / 1 SOA	...
-----------------------	--------------------	-----

Figure 7.7: DigitalOcean Setup

stages:

- test
- deploy

test-frontend:

```
image: alpine:latest
stage: test
script:
  - cd frontend
  - apk add --update nodejs npm
  - apk add --update npm
  - npm install
  - npm run test:ci
```

test-backend:

```
image: alpine:latest
stage: test
script:
  - cd backend
  - apk add --update nodejs npm
  - apk add --update npm
  - npm install
  - npm test
```

deploy-frontend:

```
image: alpine:latest
stage: deploy
dependencies:
  - test-frontend
  - test-backend
```

```

script:
  - chmod og= $ID_RSA
  - apk update && apk add openssh-client
  - ssh -i $ID_RSA -o StrictHostKeyChecking=no $SERVER_USER@$FRONTEND_API "cd /usr/src/
environment:
  name: production
only:
  - main
  
```

```

deploy-backend:
  image: alpine:latest
  stage: deploy
  dependencies:
    - test-frontend
    - test-backend
  script:
    - chmod og= $ID_RSA
    - apk update && apk add openssh-client
    - ssh -i $ID_RSA -o StrictHostKeyChecking=no $SERVER_USER@$BACKEND_IP "cd /usr/src/b
  environment:
    name: production
  only:
    - main
  
```

And this diagram 7.8 represents our CI/CD process.

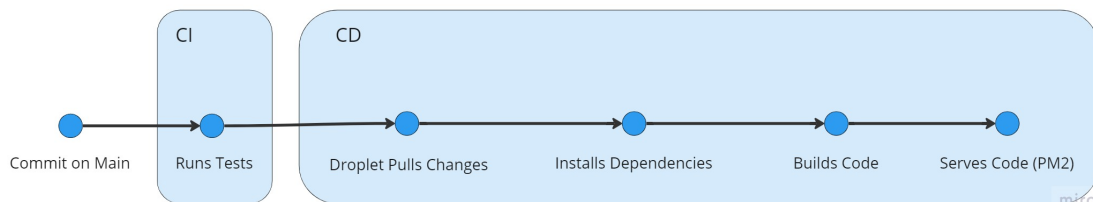


Figure 7.8: CICD Model

Technical Test Documentation

Backend Testing

Unit

Tests

We unit tested the Backend using [jest](#). Jest is a popular unit testing framework for Nodejs applications. Since we use Typescript for the Backend we had to use a [additional package](#), to allow us to use jest to test our typescript code. We aimed to achieve a 90% test coverage for the backend. We decided to exclude two directories from unit testing and coverage:

- /models
- /database

The /models directory only holds data representation classes and has almost no logic. For that reason, we do not unit-test them. The /database directory holds all of our repository classes responsible for database access. Since the methods in these repositories are very short, and most of the time, contain only one statement for access to the MongoDB database, it would make no sense to test them separately. Unit testing a method like this would require mocking the collection variable. This would make testing the method useless since we have to mock out almost all of its logic. An example of a function requiring all its logic to be mocked is listed below. Access to the collection must be mocked, and that makes this function essentially completely mocked and nothing is left to be tested.

```

async upsertDailyActivitySummary(today: FitbitDate, data: any): Promise<boolean> {
  const result = await this._collection.replaceOne(
    {"date": today.toString(), "dataType": "DailyActivitySummary"},
    {
      "activities": data.activities,
      "goals": data.goals,
      "summary": data.summary,
      "userEmail": data.userEmail,
      "dataType": data.dataType,
      "date": data.date
    },
    {upsert: true}
  )
  return !!result
}

```

The backend test coverage at the end of the project was:

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	80.98	62.5	74.46	80.86	
config	100	56.25	100	100	
algorithm.config.ts	100	100	100	100	
db.config.ts	100	75	100	100	5
db.datatypes.config.ts	100	100	100	100	
fitbit.config.ts	100	50	100	100	3-5
garmin.config.ts	100	50	100	100	5-6
jwt.config.ts	100	50	100	100	3
controllers/dashboard	93.54	60	100	93.54	
dashboard-controller.ts	93.54	60	100	93.54	18, 51, 80, 96
fitbit-service.ts	70.83	71.42	53.57	70.83	27-38, 50-75, 140-143, 213, 249, 262-282, 292-307
services/garmin	100	100	100	100	
garmin-activity-service.ts	100	100	100	100	
garmin-credentials-service.ts	100	100	100	100	
garmin-daily-summary-service.ts	100	100	100	100	
services/user	96.55	50	100	96.55	
user-service.ts	96.55	50	100	96.55	52
utils	100	100	100	100	
credentials-util.ts	100	100	100	100	
garmin-oauth-util.ts	100	100	100	100	
logger-util.ts	100	100	100	100	
Test Suites: 11 passed, 11 total Tests: 86 passed, 86 total Snapshots: 0 total Time: 13.334 s					

Figure 7.9: Backend Test Coverage

Frontend Testing

We decided to also unit test our react frontend with [jest](#) since we can use this library both for our backend and frontend. In the frontend we do not aim to test 90% of the components only render non-interactive elements. Testing these makes little sense since they have no edge cases or user interactions. Non-interactive components like these can be tested easily through manual testing. So we only test interactive components like the login page.

Manual Test Protocol

We also decided to setup a manual test protocol that gets run after each sprint or when desired. At first, it was more informal, but we decided to formalize it here. These will mainly test frontend components and integration with the backend but not test backend functionality fully. This Test protocol represents the final version of the test protocol. It had evolved throughout the project and was first created when the first tracker was added to the application, which was Fitbit. Since we currently have no active users and the application is not yet in production, we decided to do these tests on "fit.adaptit.ch" directly. Due to budget constraints, we decided not to create a new instance and test it directly there. The results of every time this Protokoll was run are not listed since it would clutter this document. However, the last run at the end of the project was successful.

Out of scope:

Scheduled tasks cannot be tested with a manual test protokoll. But they can be checked each day to see if they have been run.

Prerequisites:

1. Connect to database hosted on DigitalOcean with [MongoDBCompass](#).
2. Drop all collections.
3. Create a weeklySummary collection
4. Insert two entries for "max.muster@muster.ch" (date spanning from sunday two weeks ago to the last saturday, both entries with 20080 weekly steps)
5. Insert two entries for "jannet.muster@muster.ch" (date spanning from sunday two weeks ago to the last saturday, both entries with 10000 weekly steps)
6. Go to your garmin connect account and disconnect from "Fitness Data Tracker"
7. Navigate to "fit.adaptit.ch"

Step	Procedure	Expected Result
1	Create user with email "max.muster@muster.ch"	Should navigate to dashboard page. Also check db entry if it matches entered values
2	Check if data is displayed in the graph	Should display a line in the line graph right above 20000 steps.
3	Click the three dots in the right corner, press "Register Device" then press "Register Fitbit"	Should get redirected to Fitbit login page where authorization for our app is needed
4	Authorize app	Should get redirected back to the Dashboard. Expect collection "fitbitCredentials" and entry for "max.muster@muster.ch" to be made in db (check with MongoDBCompass).
5	Click three dots in right corner, press "Register Device" then press "Register Garmin"	Should get redirected to Garmin login page, where authorization for our app is needed
6	Authorize app	Should get redirected back to the Dashboard. Expect collection "garminCredentials" and entry for "max.muster@muster.ch" to be made in db (check with MongoDBCompass).
7	Go to the Garmin data Generator tool (link excluded due to confidentiality). Generate daily summaries, and activities for user access token present in db entry.	Wait 1m and check if collection "garminDailySummary" and "garminActivity" have been made, both with entries that match the generated data.
8	Go back to the Dashboard and copy the dashboard code present there and then logout	Expect to be returned to the login page
9	Create a new user with the email "janet.muster@muster.ch"	Should navigate to dashboard page. Also check db entry if it matches entered values
10	Check if data is displayed in the graph	Should display a line in the line graph at 10000 steps.
11	Enter the copied dashboard code in the textfield present in the drawer menu, then press "GET ACCESS"	Should add a new entry below "My Dashboard" with the name "max muster".
12	Press the new entry	The Graph should now display data from "max.muster@muster.ch". Check if graph line is at 20000 steps.
13	Press "My Dashboard" entry in drawer	Graph should display a line at 10000 steps
14	Press the top right menu, then press "Delete Account"	Expect to be redirected back to login page. Check database to see if all entries for that user have been deleted

8 Result

In this project, we created a basis for a new kind of platform where tracker data from two manufacturers is displayed in a single unified dashboard. As a result, we were able to complete all functional requirements.

The user can create an account on our application. He can sync two devices both, Garmin and Fitbit. After he syncs these devices, he can view his Data on a unified dashboard. This requirement was changed in the project's duration with our supervisors' approval only to include the user's weekly steps. The user can share his dashboard with others, view other dashboards and revoke access to his dashboard from others. Data from synced trackers gets merged with a weighting algorithm. The application can be deployed via CI/CD and is hosted on a publicly available domain.

We were able to complete most of the NON-functional requirements. The completed ones are listed below.

- Features were prioritized with the client
- The Web app is able to run on Firefox, Chrome, and Safari.
- The Web app is reachable through a domain provided by the client publicly.
- We tested the application with three end-user tests. The rating of these can be viewed 10.3.1
- No system errors occur, but they are logged.
- Each error is logged.
- Communication between the backend and frontend is HTTPS (SSL) encrypted.
- User passwords get saved as a Hash in the database.
- User can only view data that he has access to.
- Business logic is implemented modularly to allow further development.
- Implemented functionalities are deployed on DigitalOcean.

Non-functional requirements not achieved will be listed here. Reasons for that are described in 5.2.

- Backend must be able to handle 1000 requests per minute. This was never tested
- Backend-API must be tested with an API-Testing tool. Such API endpoint tests have not been done.

The optional features were tracked, but we did not implement them.

The resulting application covers all the functional requirements and most non-functional requirements. Where we deviate from these is described in this document.

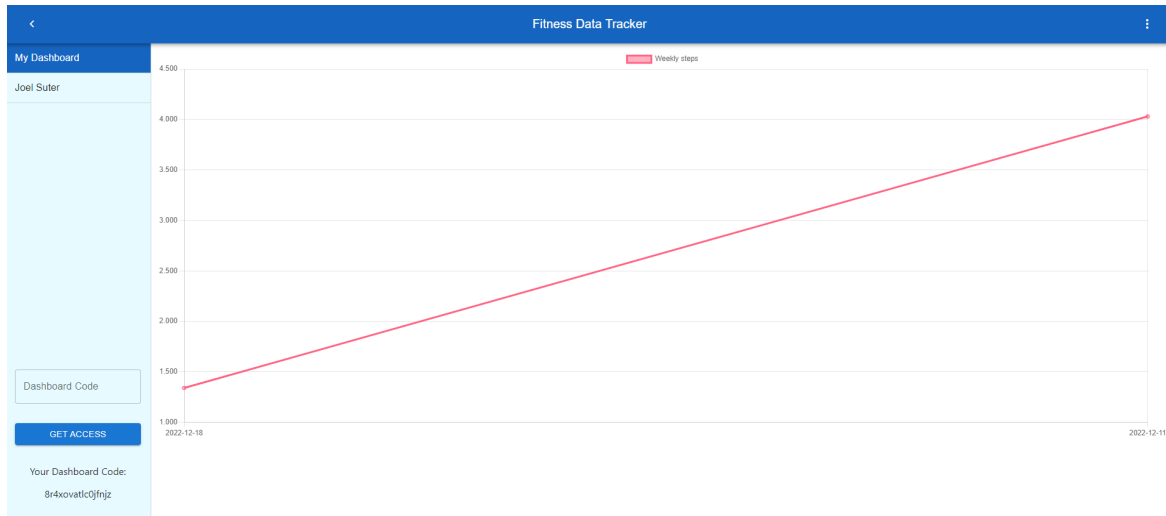


Figure 8.1: Dashboard Main Page End Result

9 Conclusion

In this project, we created a basis for a platform that does not exist as such yet. A platform where a user can keep track of multiple devices, share his data and view the data of others.

We were able to fulfill all the functional requirements and are happy with the result we were able to achieve.

The shortcomings of this project are mainly testing. We could not fulfill all non-functional requirements regarding the testing of the application since we needed more time, budget constraints in hosting, and had to set priorities. We mentioned these in previous chapters.

9.1 Needs to be addressed

Since the basis for a unified Fitness Tracker Platform stands, before any new features get implemented, testing of the application should be extended. This includes new testing hosting environment, stress tests, and API tests. We are happy with what we achieved but are not satisfied with the number of tests we were able to achieve.

Before any new features are implemented, all bugs or errors that we did not yet detect and might be detected in the new tests must be addressed.

9.2 Future Vision

In our opinion, this project has many potentials and needs in the current digitalized world. More and more people wear tracker devices.

The optional requirements mentioned in the initial task we received are perfect extensions for new features and should be considered for future work.

We of course, have some additional input in the forms of questions that can guide further development:

- Should more tracker companies be added to the supported trackers?
- What data is next to be displayed in the frontend?
- Should horizontal scaling be considered before a go live?
- If dashboards can be customized in the future, should a shared dashboard be displayed as the user has configured it, or should the one that has access be able to customize it?
- Should a user be able to choose what data we collect?
- Should a user be able to choose what data he wants to share with another user?
- Since we are working with PHI, what degree of security should be implemented before a go live.

10 Project and Time Management

10.1 Project Plan

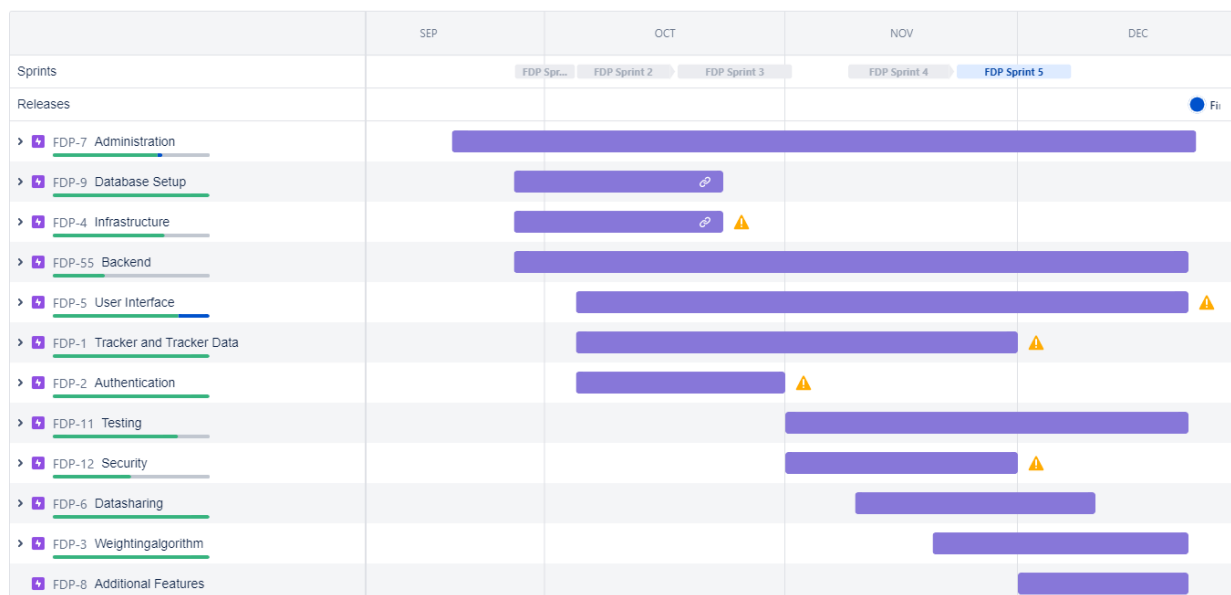


Figure 10.1: Initial Project Plan

The figure 10.1 shows our initial project plan at the beginning of the project. In the end, it deviated due to some unknown factors at the beginning of the project. These will be mentioned later. Note that the completion bars of the milestones can be ignored due to the screenshot being taken in the middle of the project. We will quickly explain what these milestones mean and what work was done in them:

- **Administration:** This was work on the documentation, meetings and general work that couldn't directly be associated with an issue.
- **Database Setup:** This milestone was work regarding the database and its setup. It also includes choosing our database technology and setting up the connection to the code
- **Infrastructure:** Everything to do with hosting and CI/CD
- **Backend:** All the work needed to be done in the backend. Often overlaps with other milestones(epics). Since this is the biggest part of our project, it is no surprise that it runs for almost the whole duration.
- **User Interface:** All work done to the frontend. Sometimes overlapped with "backend" and "Tracker and Tracker Data".
- **Authentication:** Work done regarding user authentication.

- **Testing:** Everything regarding test setup, from unit tests to manual test protocol, to test the concept
- **Security:** In this milestone, we made sure that our application has an acceptable degree of security. That means password hashing to keep secrets safe and HTTPS.
- **Datasharing:** Is regarding the dashboard sharing, revoking, and switching requirements
- **Weightingalgorithm:** This milestone represents work done for Data Normalization, weighing tracker data etc.

10.1.1 Deviations

As in all projects, we ended up deviating from the project plan in some aspects. They will be listed here

- **Database Setup:** This Milestone took longer, longer then the end of October. Reason for that is some technical issues with hosting the application.
- **Infrastructure:** This was extended the same amount as the "Database Setup" milestone for the same reasons. Issues with hosting on DigitalOcean.
- **Tracker and Tracker Data:** This one probably had the biggest deviation. We began it earlier than intended since we had to clear up some unknown variables regarding data collection from Garmin and Fitbit, and it took longer, up to the middle of December, since issues with the Garmin documentation arose.

We were able to complete the rest as seen in the project plan with a few deviations not worth mentioning here, for example, minor changes in User Authentication, which happened after the Milestone was completed.

Project organization

We have organized ourselves according to SCRUM with minor changes as we are a small team of two developers. The long-term plan has been organized with RUP. The minor changes to SCRUM are listed below:

- Our sprints last for two weeks except for the first sprint, which we decided on for one week as it only consisted of administrative and setup issues.
- Our review meetings will be held every week. Inconsistencies can occur due to advisors and clients who have different work schedules.
- We scratched the Daily SCRUM Meeting due to overhead and effectiveness for a team consisting only of two people.

Roles

Usually, SCRUM based projects divide their participants into roles. Now because we are only a team of two developers, we decided to leave the role distribution out. Our team members divide their work depending on who has time and more experience in a given issue. As both team members bring similar experiences in the field, there is no need to define specific roles for individuals. The other team member will review any code or major decisions someone makes.

Issue management

For issue management, we decided to use Jira Cloud Solution. One of our team members has prior experience with it, and our client suggested using this solution. We have divided our requirements into epics which will be further divided into more minor issues. The only release we captured is the final submission, so it is easier for us to estimate which epics have to be done in a time frame.

Time tracking

This section will list the time spent on each issue divided by each month. We tracked the time by logging it to the respective issue or logging it directly to an epic if the work couldn't be associated with a specific task. We will not list the time spent per person since both members worked together and any time differential will be caused by a overlogging or underlogging time.

Key	Issue	Priority	Time Spent
✓ FDP-14	FDP-7 / DoD and DoR	=	0.25h
✓ FDP-16	FDP-7 / Vision	=	0.5h
✓ FDP-22	FDP-7 / Document NFR	=	1.5h
✓ FDP-31	FDP-4 / Build up Digital Ocean Knowledge	=	2h
✓ FDP-18	FDP-5 / User interface draft	=	2h
✓ FDP-26	FDP-7 / Create Initial Project Plan	=	3h
✓ FDP-15	FDP-7 / Risk Management	=	3h
✓ FDP-13	FDP-7 / Initial Backlog	=	6h
✗ FDP-5	User Interface	=	10h
✗ FDP-7	Administration	=	11h
✗ FDP-55	Backend	=	16h
Total (11 issues)			55.25h

Figure 10.2: Time Spent September

Key	Issue	Priority	Time Spent
✓ FDP-38	FDP-2 / Decide on Authentication Framework	=	0.42h
✓ FDP-42	FDP-7 / Create Agenda for weekly meetings	=	0.5h
✓ FDP-62	FDP-7 / Add actors to the use case diagram	=	1h
✓ FDP-45	FDP-5 / Analyse and decide for frontend dashboard framework	=	1h
✓ FDP-29	FDP-9 / Pick Database Technology	=	1.33h
✓ FDP-23	FDP-7 / Domain analysis	=	1.5h
✓ FDP-44	FDP-1 / Garmin Initial Setup	=	2h
✓ FDP-28	FDP-7 / Document Project Organization	=	2h
✓ FDP-28	FDP-7 / Define Use Cases	=	2h
✓ FDP-17	FDP-7 / Management Summary	=	2.5h
✓ FDP-48	FDP-9 / Use sequelize in node.js in connection to postgres (initial db setup) with rest api	=	3.17h
✓ FDP-34	FDP-1 / Decide and obtain trackers	=	3.5h
✓ FDP-32	FDP-4 / Connect Digital Ocean (Setup CICD)	=	4h
✓ FDP-15	FDP-7 / Risk Management	=	4.25h
✓ FDP-38	FDP-9 / Setup Initial Database	=	5h
✓ FDP-65	FDP-5 / Dashboard Initial view (display only)	=	6.75h
✓ FDP-47	Setup Rest Api	=	7h
✗ FDP-4	Infrastructure	=	7.75h
✓ FDP-61	Change backend to use MongoDB	=	8h
✗ FDP-39	FDP-2 / User login	=	9.5h
✗ FDP-55	Backend	=	12h
✗ FDP-7	Administration	=	24h
✓ FDP-43	FDP-1 / Collect Initial Fitbit Tracker Data	=	27.25h
Total (23 issues)			136.42h

Figure 10.3: Time Spent October

Key	Issue	Priority	Time Spent
✓ FDP-77	FDP-7 / User interface draft in doc	=	0.5h
✓ FDP-60	FDP-7 / Document DB discussion/decisions	=	0.5h
✓ FDP-76	FDP-7 / Domain model description	=	0.75h
✓ FDP-97	FDP-11 / Write frontend tests for existing components	=	1h
✓ FDP-93	FDP-4 / Use secure connection (HTTPS)	=	1h
✓ FDP-48	FDP-2 / Admin login	=	1.5h
✓ FDP-90	FDP-7 / Document CICD in doc	=	2h
✓ FDP-87	FDP-4 / Migrate from Ost Gitlab to AdaptIT Gitlab	=	2h
✓ FDP-86	FDP-7 / Create latex documents from meeting protocols	=	2h
✓ FDP-58	FDP-11 / Backend Unit Tests	=	2h
✓ FDP-24	FDP-7 / Define Software architecture	=	2h
✓ FDP-21	FDP-11 / Test Concept (Doc)	=	2.33h
✓ FDP-106	FDP-5 / Display graph with steps on the dashboard	=	4h
✓ FDP-101	FDP-1 / Data preparation for weekly summary of fitbit	=	5h
✓ FDP-104	FDP-7 / Transfer documentation to new environment with better table of contents	=	5.5h
✓ FDP-88	FDP-4 / Configure CICD in new Gitlab	=	6h
✓ FDP-32	FDP-4 / Connect Digital Ocean (Setup CICD)	=	6h
✓ FDP-9	Database Setup	=	6h
✓ FDP-44	FDP-1 / Garmin Initial Setup	=	7h
✓ FDP-43	FDP-1 / Collect Initial Fitbit Tracker Data	=	7h
✓ FDP-1	Tracker and Tracker Data	=	8h
✓ FDP-102	FDP-3 / Initial setup for weighting algorithm	=	8.5h
✓ FDP-4	Infrastructure	=	10h
✓ FDP-55	Backend	=	12h
✓ FDP-7	Administration	=	14h
Total (25 issues)			116.58h

Figure 10.4: Time Spent November

Key	Issue	Priority	Time Spent
✓ FDP-108	FDP-11 / Frontend technical test documentation	=	0.67h
✓ FDP-114	FDP-12 / fix multiple user with same email	=	1h
✓ FDP-97	FDP-11 / Write frontend tests for existing components	=	1h
✓ FDP-96	FDP-11 / Technical Backend Test documentation	=	1h
✓ FDP-78	FDP-1 / Create job for data synchronisation	=	1h
✓ FDP-117	FDP-1 / Garmin daily summary data normalization	=	2h
✓ FDP-93	FDP-4 / Use secure connection (HTTPS)	=	2h
✓ FDP-51	FDP-1 / User is able to sync register device in frontend	=	2h
✓ FDP-107	FDP-11 / Add test stage to cicd	=	2.25h
✓ FDP-123	FDP-5 / Fix some frontend design problems	=	3h
✓ FDP-58	FDP-6 / User is able to revoke Dashboard access	=	5h
✓ FDP-115	FDP-1 / garmin collect daily summaries	=	5.67h
✓ FDP-57	FDP-11 / Frontend Tests	=	6.5h
✓ FDP-2	Authentication	=	7h
✓ FDP-44	FDP-1 / Garmin Initial Setup	=	7.75h
✓ FDP-106	FDP-5 / Display graph with steps on the dashboard	=	8h
✓ FDP-122	FDP-7 / User is able to delete his account	=	9h
✓ FDP-58	FDP-11 / Backend Unit Tests	=	9h
✓ FDP-49	FDP-6 / User is able to share Dashboard through UI	=	9h
✓ FDP-52	FDP-6 / User is able to switch dashboard through UI	=	9.5h
✓ FDP-111	FDP-11 / Refactor unit tests	=	11.5h
✓ FDP-101	FDP-1 / Data preparation for weekly summary of fitbit	=	12.25h
✓ FDP-85	FDP-1 / Garmin initial data collection	=	14.17h
✓ FDP-121	FDP-55 / Refactor code in backend	=	19h
✓ FDP-7	Administration	=	27h
Total (25 issues)			176.25h

Figure 10.5: Time Spent December

Division of Tasks

We mostly worked together on most tasks, but still there are some divisions on how the overlying requirements were divided under each other. The rough divisions of tasks will be listed below. Some tasks both worked on equally and that will be mentioned too.

Task	Lead
Fitbit Data	Lucas v. Niderhäusern
Garmin Data	Joel Suter
Dashboard Sharing	Lucas v. Niderhäusern
User deletion	Lucas v. Niderhäusern
Documentation	both
Data Normalization	Lucas v. Niderhäusern
CI/CD and Hosting	Joel Suter
Testing	Both
Frontend setup	Joel Suter
Database setup	Both
Database setup	Both
Scheduled Tasks	Joel Suter

Table 10.1: Divisions of tasks

10.2 Risk Management

This section will describe the risks to our project. If they occur, they are listed in 10.2 with their respective probability and severity to the project. The section will be constantly updated for any new risks that arise during the project and the realized risks that happened. Any countermeasures and their effectiveness will be documented as well.

Risk ID	Risk	Countermeasure	Severity	Probability
1	Team member is unavailable	Communicate through channels to distribute tasks between other team members	Very High	Likely
2	Scope Creep	Any changes to scopes have to be estimated again	High	Very Likely
3	Technical inexperience	Assign issues to the person best suited. Eliminate any inexperience as early as possible	High	Likely
4	Infrastructure breakdown	Create backups and ensure high availability of the infrastructure	Very High	Not likely
5	Poor project planning	Review project plan after every sprint	High	Not likely

Table 10.2: Project Risks



Figure 10.6: Risk acceptance graph

Realized risks

Team member is unavailable

One team member was unavailable for 3 days. This absence was known in before and workload was distributed respectively.

Severity: Small

Countermeasure: Successful

Team member is unavailable

Another team member was unavailable for 3 days during a different time period. This absence was known in before and workload was distributed respectively.

Severity: Small

Countermeasure: Successful

Technical inexperience:

Due to our technical inexperience in MongoDB, we have realized that setting up the MongoDB took more time than initially anticipated. We have assigned the task to both us to speed up the process and read through the documentation of MongoDB.

Severity: Medium

Countermeasure: Successful

Technical inexperience

We were aware from the beginning that DigitalOcean would be needed as a cloud solution for our project. As only one of our team members has worked with DigitalOcean before, he was assigned

for the initial setup. We encountered some time consuming problems regarding the CI/CD for our project because of the private instance of our Gitlab repository. The issue was resolved after we migrated our repository to the clients private Gitlab instance.

Severity: Medium

Countermeasure: Successful

Insufficient Documentation

One of the risks we did not expect at the beginning was insufficient documentation for api's we need to use. We had problems integrating the Garmin api to our project because of somethings that were not documented. We were able to solve this by contacting the Garmin support.

Severity: Severe

Countermeasure: Successful

10.3 Test Concept

In general we will conform to the testing Pyramid and design our test concept accordingly, as it is considered to be a good guideline for software testing. When we deviate from it, we will document the changes and give reasons for that. We aim to test 80% of our code. Higher than that usually involves writing tests that don't add real value to the code base and are more meant to get to a higher percentage. From our experience, 80% is realistic to achieve with good tests.

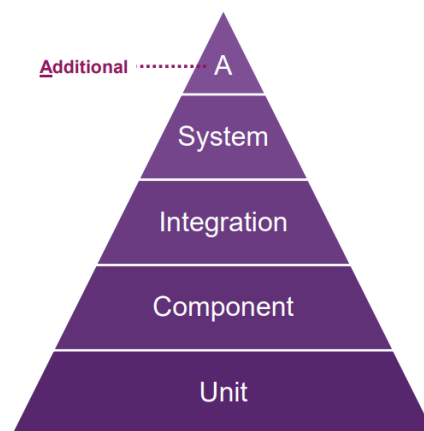


Figure 10.7: Testing Pyramid

10.3.1 Additional

As of now we don't see the need for additional test techniques since the scope of our project is overseeable. We rather concentrate on the following testing techniques and implement them with a high degree of quality.

System

We will test the system manually using a test protocol. This protocol will evolve during the duration of the project. If one of the tests fails in the protocol the sprint cannot be finished, it does

not meet the definition of done. The final iteration of the protocol will be listed in the technical test documentation, and their results. When to run: After Sprint

Integration

We cover these with our unit test and manual testing. So we decided against Integration testing.

Component

Since we have very small modules/components it doesn't make much sense to test them separately. For that reason we won't test the modules separately but will cover these tests through unit and manual testing.

Unit

We will implement most of our testing through unit tests. With these we want to make sure all the code units run as intended. When to run: With Pipeline

End User Tests and Rating

One of the requirements was that three test users would rate our application eight or higher out of ten. Since this is a very objective requirement and highly depends on the user testing, it is hard to say for sure if it was fulfilled. It also has to be mentioned that the focus in this project was in the backend, collecting tracker data and providing a basis that can be extended. The UI was not the main focus of this project.

For this testing/rating we asked three of our friends and colleagues to do the following unguided:

1. Create a user
2. Connect their fitbit account (which we asked them to create beforehand)
3. Logout
4. Create a second user
5. Access the dashboard of the first user
6. Delete the second account.

The results were as follows:

1. Rating of 6
2. Rating of 7
3. Rating of 9

In the context of our project we consider these ratings as a success, but they show us that there is still work to be done in our UI. Aggregated feedback worth mentioning are:

- The graph is not customizable (can't choose date range etc)
- Color Scheme could be more pleasant
- The Dashboard code should be hidden and only visible when pressed

- The menu and submenu at the top right could be colorcoded and or contain logos of Fitbit and Garmin.

These ratings were done at the very end of the project. The Feedback received from the users should be considered if this application received further development.

10.4 Definition of Ready / Definition of Done

This section will discuss and list the different DoDs and DoRs. These lists will serve as checklists either before we do something (DoR) or when we want to check if something is finished (DoD). In addition, it will ensure that quality standards are met with a clearly defined process.

Definition of Ready

If any of the criteria in the checklist are violated, the concerned item is not ready to be worked. It will have to be corrected until it meets them. It is ready only if all criteria are met.

- **User Stories**
 - Acceptance Criteria defined
 - Story definition accepted by Team
 - Story well-defined
 - No ambiguities
 - Story estimated
 - Story in current Sprint
 - Story assigned
- **General Ticket**
 - Acceptance Criteria defined
 - Ticket in current sprint
 - Ticket estimated
- **Sprint**
 - Tickets with highest priority in sprint
 - Sprint backlog prioritized
 - No hidden work, everything in tickets
 - Sprint planned according to capacity
 - All Stories meet definition of Ready

Definition of Done

If any of the criteria in the checklist are violated, the concerned item is considered unfinished. It will not be allowed to be moved to done or regarded as finished. It is done only if all are met.

- **User Stories**
 - Assumptions of User Story met (justified if not)
 - Project builds without errors

- Tests written according to test concept
- Story tested against acceptance criteria
- Configuration, Architecture or Build changes documented
- Code Review passed
- Code written according to defined guidelines
- NFR stay unviolated
- **General Ticket**
 - Assumptions of Ticket met (justified if not)
 - Changes to Documentation reviewed
 - Acceptance Criteria met
 - No Contradictions in Documentation
 - Decisions accepted by Team
- **Sprint**
 - DoD of each Ticket included in Sprint is met
 - Sprint scope completed
 - All tests pass
 - Backlog refined and updated
 - Application deployed
 - Upcoming Sprint planned

Bibliography

- [1] Fuller D et al. "Reliability and Validity of Commercially Available Wearable Devices for Measuring Steps, Energy Expenditure, and Heart Rate: Systematic Review." PhD thesis. JMIR Mhealth Uhealth, 2020.

List of Figures

1	MongoDB	1
2	TypeScript	1
3	Own Presentment	1
4.1	System Context	9
5.1	Testing Pyramid	10
6.1	Container Diagram	15
6.2	Component Diagram	16
6.3	Backend folder structure	18
6.4	Request diagram frontend	19
6.5	Dashboard main page Mockup 1	20
6.6	Dashboard Main Page Mockup 2	20
6.7	Login Page Mockup	20
6.8	Register Page Mockup	21
6.9	Register Page Mockup	21
6.10	Domain Model	23
6.11	DigitalOcean Restore Dialog	25
7.1	Fitbit Web API Workflow	27
7.2	Garmin API Workflow	30
7.3	User Access Code Frontend Screenshot	31
7.4	Accessed Dashboard Frontend Screenshot	32
7.5	Revoke Dashboard Access	34
7.6	Generate Weekly Summary Sequence Diagram	35
7.7	DigitalOcean Setup	42
7.8	CICD Model	43
7.9	Backend Test Coverage	44
8.1	Dashboard Main Page End Result	48
10.1	Initial Project Plan	50
10.2	Time Spent September	52
10.3	Time Spent October	52
10.4	Time Spent November	53
10.5	Time Spent December	53
10.6	Risk acceptance graph	55
10.7	Testing Pyramid	56
14.1	Login Page of Application	79
14.2	Sign In Page	80
14.3	Sign Up Page	81
14.4	Dashboard	82

List of Tables

5.1	Use Cases	11
5.3	Non-Functional Requirements	12
5.5	Optional Requirements	13
10.1	Divisions of tasks	54
10.2	Project Risks	54

11 Operational Notes

Vor allem bei Softwareprojekten: Gehen Sie auf folgende Punkte ein (bei grösserer Dokumentation verweisen Sie auf den Anhang).

1. Verwendete SDK, IDE und Werkzeuge
2. Hinweise zu CI/CD
3. Installationsanleitung / Bedienungsanleitung
4. Test-Logs
5. Bei Systemen mit User Interfaces: Dokumentation der Usability Tests

12 Meeting Minutes

Protocol 21.09 17:00

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- None

Topics / Questions

- Frontend style: Preferred libraries, stylesheet
- Backend technology: Do we need to use Node.js?
- Database technology: Can we use PostgreSQL?
- Question to "Gewichtungsalgorithmus"
- "Project Management Tool" choice (Jira)
- Choose date for issue prioritization
- Formal questions regarding shape and structure of the documentation

Decisions

- We are free to design the frontend how we want
- The backend has to be in Javascript -> Because JS developers are cheaper than other technologies
- We have to elaborate

Open Tasks / Topics for next week

- None

Protocol 27.09 17:00

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- None

Topics / Questions

- Do we organise business meetings with client?
- Is there a maximum hours we are allowed to work on this project?
- What is the difference in the UI for admins and normal clients?
- What information do we have to collect from the tracking devices?
- Does Michael have a example for a dashboard?
- When do we plan the meetings for the issue prioritization?

Decisions

- Business review will be held within the normal weekly meetings
- The dashboard should only include the steps, but has to be easily extendable
- We have to collect as much data as possible from the tracking devices into the database
- Most important features: Connection of tracking device, sharable dashboards, collection of data
- (There might be open source solutions for the dashboard)

Open Tasks / Topics for next week

- Setup meeting for issue prioritization
- Michael won't be attending next meeting

Protocol 04.10 17:00

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch

Review

- None

Topics / Questions

- Feedback to our documentation structure

Decisions

- Document end results. Did we achieve our (non-)functional requirements
- Document our software (how is it implemented, testing results)
- Document our deployment (especially workflow with DigitalOcean)

Open Tasks / Topics for next week

- Meeting with Michael regarding technical issues
- Are devices actors in the use cases?

Protocol 14.10 19:00

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- None

Topics / Questions

- Quick briefing for Frank so he is up to date with which decisions were made during last meeting

Decisions

- None

Open Tasks / Topics for next week

- None

Protocol 20.10 19:00

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- First version of UI - Feedback

Topics / Questions

- Domain Model - Database solution
- organize a technical / code review
- Data preparation (solution)

Decisions

- Sidebar open by default
- Userprofile on the dashboard
- Livedata to show that synchronization works
- Write email regarding garmin credentials (consumer secret)
- We can dump all data into mongodb collections
- Show live data / show weekly summarization
- Document difficulties regarding data preparation (if one device battery went dead, weighing)
- Optional point: Multiple graphs, one for fitbit, one for garmin

Open Tasks / Topics for next week

- None

Protocol 27.10 19:00

Present

Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- None

Topics / Questions

- Subscription API from Fitbit

Decisions

- Use subscription API as a separate service for the application

Open Tasks / Topics for next week

- None

Protocol 03.11 19:00

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- None

Topics / Questions

- Update of current events
- Does Michael have a wish for domain name
- Is cost of droplet okay?

Decisions

- Try using less performance on droplet to see how much it uses
- Send IP-Address to Michael so he can configure domain name and DNS

Open Tasks / Topics for next week

- None

Protocol 19:00 10.11.2022

Present

Joel Suter, Frank Koch, Michael Güntensperger

Review

- None

Topics / Questions

- Show deployment

Decisions

- None

Open Tasks / Topics for next week

- Frank will be absent on the first week of december
- Brainstorming ideas for bachelor thesis

Protocol 19:00 17.11.2022

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- Brief Lucas on what happened last week

Topics / Questions

- Show progress of project

Decisions

- None

Open Tasks / Topics for next week

- None

Protocol 19:00 24.11.2022

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- None

Topics / Questions

- Almost finished with testing for front- and backend
- Garmin issues

Decisions

- None

Open Tasks / Topics for next week

- None

Protocol 19:00 01.12.2022

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- None

Topics / Questions

- Showed progress of our project, nothing else

Decisions

- None

Open Tasks / Topics for next week

- None

Protocol 19:00 08.12.2022

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- None

Topics / Questions

- None

Decisions

- Reduce droplet costs to a minimum to see if any performance issues arise

Open Tasks / Topics for next week

- None

Protocol 19:00 15.12.2022

Present

Joel Suter, Frank Koch, Michael Güntensperger

Review

- None

Topics / Questions

- We reached maximum minutes on our pipelines
- AVT Tool is now open for registration

Decisions

- None

Open Tasks / Topics for next week

- None

Protocol 19:00 22.12.2022

Present

Joel Suter, Lucas von Niederhäusern, Frank Koch, Michael Güntensperger

Review

- Informal presentation of our project

Topics / Questions

- What to do with Garmin documentation due to it being confidential

Decisions

- We wished each other merry christmas and happy new year

Open Tasks / Topics for next week

- None

13 Personal Reports

Lucas von Niederhäusern

This project combined a lot of knowledge and experience I gathered during my studies at OST. Starting this project everything felt a bit bumpy because of my inexperience in working with API's. I needed a lot of ground work to get accustomed to how API's work and how I can use them efficiently in this project. Otherwise, especially in the backend, most things worked out pretty great because of past projects I built with similar technologies. It was also a great experience to work with React technology in the frontend and how efficient these libraries work once I got the hang of it. I am happy with the results of this project even though we could not implement every feature we intended to and hope that this project will be a good foundation on what Adapt IT wants to implement in the future.

13.0.1 Joel Suter

In this project I tested what I have learned in my years as a DevOps engineer and my time here at OST. I sharpened my skills in Typescript, hosting, CI/CD, and more. The project is a success. We were able to achieve what we were given. Of course, as in all projects, some times could have been better. I learned a lot, especially in project planning. We often estimated tasks to be shorter than they were since they often had new APIs we had never worked with. I will calculate more buffers for unknown variables like external APIs in future projects. Nevertheless, I enjoyed my time working on this project, even when not everything worked out as planned. I am happy with what a team of two people achieved, and I am happy with our result. This basis we built will bring great benefit to our client. I learned a lot in this project which I will certainly apply in the next one.

14 Screenshots

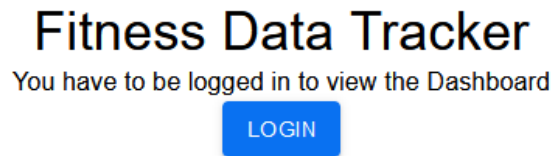


Figure 14.1: Login Page of Application

Sign in

SIGN IN

[Don't have an account? Sign Up](#)

Figure 14.2: Sign In Page

Register

email

First Name

Last Name

Role

Client

password

password

SIGN IN

[Already have an account? Sign In](#)

Figure 14.3: Sign Up Page

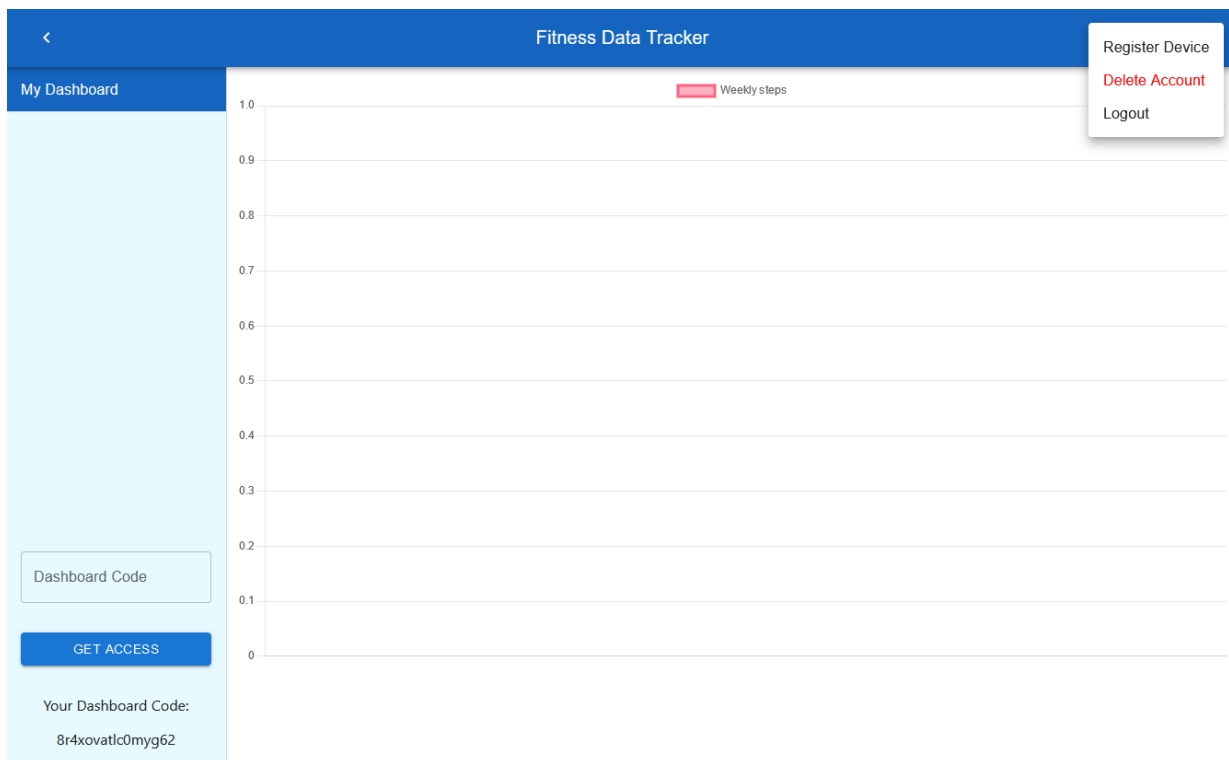


Figure 14.4: Dashboard

15 Task

15.1 FitnessDataPlatform

Beteiligte Personen

- Studierende: Lucas von Niederhäusern und Joel Suter
- Industriepartner: AdaptIT GmbH, Michael Güntensperger
- Betreuer: Frank Koch

15.1.1 Problembeschrieb

Fitness-Tracker-Daten werden im Bereich Personal Training immer wichtiger. Im Moment haben Trainer nur die Möglichkeit, anhand separater Plattformen der Fitness-Tracker-Hersteller die Bewegung der Kunden zu tracken und auszuwerten. Dies wird schnell sehr aufwendig, wenn ein Trainer mehrere Kunden mit unterschiedlichen Trackern hat (z.B. Samsung, Google, Apple, Fitbit, Garmin, ...). Damit dies in Zukunft einfacher wird, soll ein Tool zum Anbinden von Fitness-Trackern und der Freigabe der Daten z.B. an einen Personal Trainer entwickelt werden.

15.1.2 Aufgabenstellung

Mit diesem Projekt soll die Schnittstelle eines Fitnessstracker-Anbieters (z.B. Garmin) an ein Backend angebunden werden und die Daten auf einem Dashboard ausgewertet werden. Dashboards sollen mit anderen Usern geteilt werden können (Personal Trainer kann die Dashboards der Kunden einsehen). Die Applikation soll modular aufgebaut sein, damit in Zukunft weitere Fitnessstracker-Anbieter angebunden und das Dashboard erweitert werden kann.

15.1.3 Technische Umgebung

Für die Umsetzung wird mit Web-Technologien gearbeitet.

- Frontend: Angular / React
- Backend: Node.js
- Datenbank: z.B. MongoDB

15.1.4 Funktionale Anforderungen

- Anbindung von zwei Trackern (z.B. zwei unterschiedliche Fitbit-Geräte).
- Evaluieren was für Datenbanktechnologie eingesetzt werden soll, um diese Datenmenge darzustellen und die Umsetzung damit.
- Benutzer

- Erstellen und Löschen von Accounts.
- Darstellen der Daten auf einem Dashboard.
- Teilen von Dashboards mit anderen Usern (so kann Personal Trainer z.B. 20 Dashboards einsehen).
- Berechtigungen an Externe wieder entziehen können.
- Gewichtungsalgorithmus, um zu entscheiden, welche Daten für die Auswertung relevant sind, wenn ein Benutzer zwei Tracker bei sich hat (z.B. Smartwatch und Handy). Z.B. Genauigkeit von Fitbit Armband beim Tracken von Schritten genauer als das Samsung Handy -> Für die Auswertung die Schrittzahl von Fitbit nutzen.
- Deployen der Applikation.
- Einladung zum Erstellen eines Accounts versenden können (z.B. per Mail oder WhatsApp).

15.1.5 Optionale Anforderungen

- Verschlüsselung der Daten, dass auch der Admin diese nicht einsehen kann.
- Setzen von Zielen im Dashboard (Bsp. Schritte).
- Benutzer können das Dashboard selbst bearbeiten und für sie relevante Informationen anzeigen.
- Personal Trainer können via Chat mit Kunden kommunizieren und Videos für diese hochladen.
- Erstellen von Communities (Kollegen, Verein, Firma, ...).
- Starten von Challenges (z.B. wer Schafft am meisten Schritte diese Woche).
- Chat-Funktionalität.
- Integration Slack / Teams (um über Stand der Challenge zu informieren).
- Monetarisierung von Professional-Accounts.
- Erstellen einer App mit minimaler Auswertungs- und Chat-Funktionalität.
- Manuelle Erfassung von anderen Daten z.B Tägliches Wohlbefinden, Stresslevel, Blutuntersuchungen, etc.
- Manuelle Einstellung, welcher Tracker für welche Daten eingesetzt werden soll.

15.1.6 Nicht-Funktionale Anforderungen

- Das Entwicklerteam implementiert die Features gemäss der abgesprochenen Priorität mit dem Kunden.
- Das Backend (Fitnessdaten) sollte 1000 Requests pro Minute handeln können (Abhängig von der Implementation des Daten-Uploads).
- Jede Seite sollte nicht länger als 200ms für das Laden benötigen.
- Die Web-Applikation sollte auf Firefox, Chrome und Safari laufen.
- Via Internet sollte auf eine vom Kunden zur Verfügung gestellte Domain zugegriffen werden können.

- Drei von vier Testusern sollten das UI (Kategorien: layout, responsiveness, colour, content) der Applikation mit einem Tablet mit einer Note von mindestens 8 von 10 bewerten, wobei 10 das Beste ist.
- Die Datenbank soll bis zu 100'000 Datensätze managen können.
- Errors sollen keine Systemfehler erzeugen, aber eine Error Nachricht Zeigen und das System auf den vorherigen Zustand zurücksetzen.
- Jeder Error soll im System geloggt werden.
- Jede Kommunikation zwischen Front- und Backend soll mit einem SSL-Zertifikat verschlüsselt werden.
- Daten welche in Eingabefelder abgefüllt werden, sollen zuerst validiert werden, bevor diese durch das System verarbeitet werden. SQL Injection test der Eingabefelder sollte keine Verletzlichkeiten zeigen.
- User-Passwörter werden nicht in plain-text in der Datenbank gespeichert.
- Wenn sich ein User in die Web-Applikation einloggt, werden ihm auch nur seine Daten / auf Daten die er Zugriff haben soll, angezeigt.
- Businesslogik im Backend soll modular aufgebaut werden, so dass sie erweitert werden kann.
- Die Backend-API soll durch API-testing Tools getestet werden.
- Implementierte Funktionalität (Datenbank, Backend, Frontend,...) sollen deployed werden.

15.1.7 Zur Durchführung

Mit dem Betreuer finden Besprechungen gemäss Absprache statt. Die Besprechungen sind von den Studierenden mit einer Traktandenliste vorzubereiten und die Ergebnisse in einem Protokoll zu dokumentieren, das dem Betreuer per E-Mail zugestellt wird. Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. An Meilensteinen gemäss Projektplan sind einzelne Arbeitsergebnisse in vorläufigen Versionen abzugeben. pdfpages

15.2 Additional Documents

[pages=-]template/resources/pdfs/eigenstaendigkeit.pdf [pages=-]template/resources/pdfs/einverstaendnis.pdf
[pages=-]template/resources/pdfs/urheber.pdf