Software Design Document

- for Diabetter, a Diabetes data dashboard

Team members

Alexandra Nikolova	- 1339311
Rik Litjens	- 1317059
Vasil Shteriyanov	- 1307282
Gabriela Zanova	- 1307509
Konstantin Velkov	- 1307436
Roel Koopman	- 1299743
Rinse Vlaswinkel	- 1312529
Nikaels Balasovs	- 1250221
Jeroen Gijsbers	- 1305832
Kevin Dirksen	- 1302191

Client & Platform Owner
M. Chaudron & P. Van Gorp

Project Managers
A. Pintea & C. Olteanu

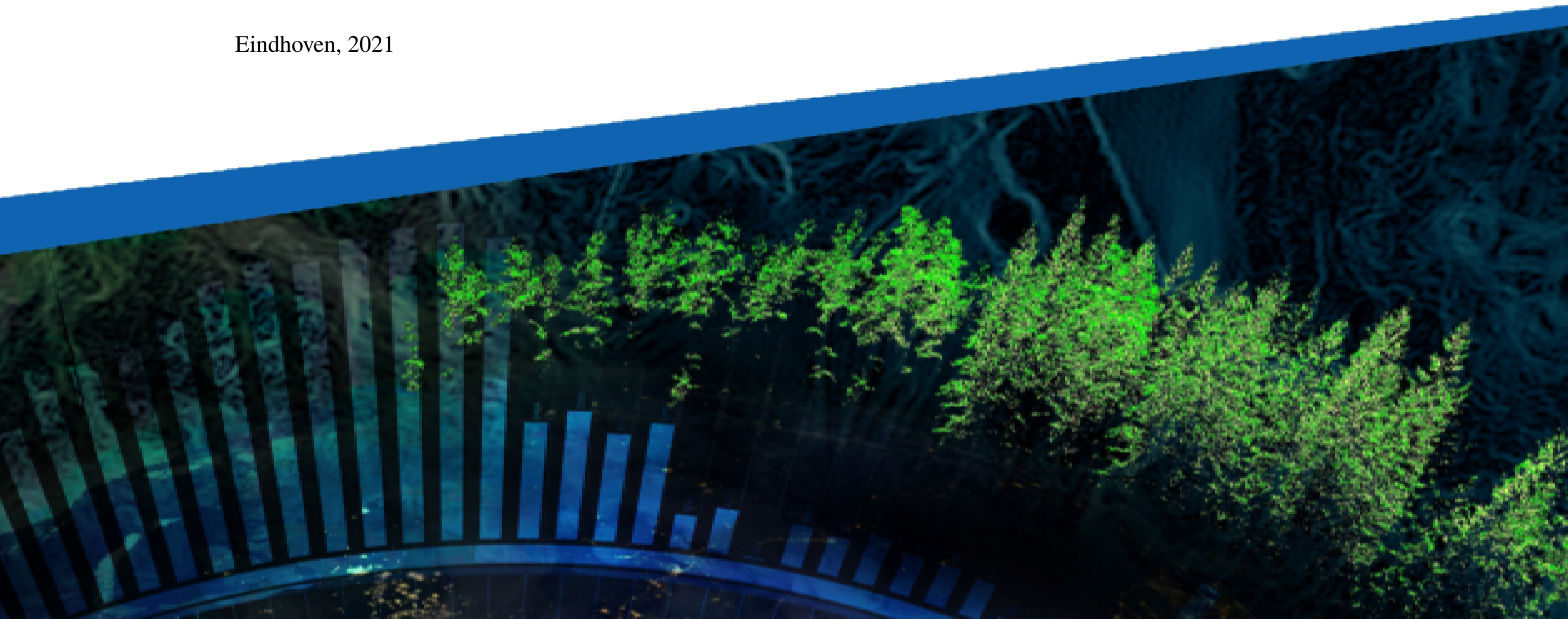
Supervisor
G. Kahraman

Eindhoven University of Technology

Department of Mathematics and Computer Science

Diabetter

Eindhoven, 2021



Abstract

This document is a Software Design Document (SDD) for Diabetter, a web-based data dashboard for people living with Diabetes. The design decisions presented in this document comply with the requirements as stated in the User Requirements Document (URD). Furthermore, this document is in line with the ESA software standards.

Contents

1 Introduction 5

1.1 Purpose . . . 5

1.2 Scope . . . 5

1.3 List of definitions and abbreviations . . . 5

1.3.1 Definitions . . . 5

1.3.2 Abbreviations . . . 6

1.4 List of references . . . 6

1.5 Overview . . . 7

2 System overview 8

2.1 Relation to current projects . . . 8

2.2 Relation to predecessor and successor projects . . . 8

2.3 Function and purpose . . . 8

2.4 Environment . . . 9

2.5 Relation to other systems . . . 9

3 System architecture 11

3.1 Architectural design . . . 11

3.1.1 Front-end . . . 11

3.1.2 Back-end . . . 12

3.2 Logical model description . . . 16

3.2.1 Front-end . . . 16

3.2.2 Back-end . . . 33

3.2.3 API endpoints . . . 77

3.3 State dynamics . . . 90

3.4 Data model . . . 107

3.4.1 Local database . . . 107

3.5 External interface definitions . . . 108
3.5.1 GameBus . . . 108
3.5.2 OneDrive . . . 109
3.5.3 NightScout . . . 110
3.6 Design rationale . . . 111
3.6.1 Separation of front-end and back-end . . . 111
3.6.2 Front-end language and framework . . . 111
3.6.3 Plain HTML / CSS . . . 111
3.6.4 React.js . . . 111
3.6.5 Vue.js . . . 111
3.6.6 Back-end language and framework . . . 112
3.6.7 Back-end libraries . . . 112
3.6.8 Data storage . . . 113
3.6.9 Continuous Integration and Deployment . . . 113

4 Feasibility and resource estimates **114**
4.1 Minimum requirements . . . 114
4.1.1 Development machine minimum requirements . . . 114
4.1.2 Client machine requirements . . . 114
4.1.3 Server machine requirements . . . 114
4.2 Performance . . . 114
4.2.1 Test setup . . . 115
4.2.2 Results . . . 116

Document status sheet and change records

Document title	Software design document
Document identifier	Diabetter.SDD/1.0
Authors	Nikaels Balasovs
	Roel Koopman
	Jeroen Gijsbers
	Konstantin Velkov
	Kevin Dirksen
Document status	Final version

Document history

Version	**Date**	**Authors**	**Description**
0.0	14-05-2021	Rik, Nikaels, Roel	Initial version
0.1	20-05-2021	Roel, Nikaels	Start on chapter 1 and 2
0.2	01-06-2021	Roel, Konstantin, Nikaels	Work on chapter 2
0.3	14-06-2021	Roel, Konstantin, Jeroen, Kevin	Work on chapter 3
0.4	21-06-2021	All	Work on chapters 3
0.5	24-06-2021	Roel	Finished chapter 4
0.6	28-06-2021	All	Finishing chapter 1, 2, 3
1.0	01-07-2021	Roel, Nikaels, Jeroen	Proof read and styling.

Change records

Version	**Changes**
0.0	Added document template, Abstract, Section 2 and 3
0.1	Finished on sub chapters 1.1, 1.2, 2.1, 2.2, 2.3
0.2	Finished chapter 2
0.3	Work on sub chapters 3.1.1 and 3.1.2
0.4	Work on chapter 3
0.5	Finished chapter 4
0.6	Finished chapters 1, 2, 3
1.0	Proof read all the chapters

1 Introduction

1.1 Purpose

The purpose of the Software Design Document (SDD) is to provide an overview of the functions, purpose, environment, relation to other systems, as well as, the relation to predecessor and successor projects. Moreover, this document provides high-level architecture overview of Diabetter which is subdivided into multiple components in order to increase readability. Lastly, both design decisions and estimation for the resources needed to run the application are given.

1.2 Scope

The Diabetter initiative will be designed and developed by a group of Computer Science and Engineering students from the Eindhoven University of Technology, in cooperation with the client and the platform owner. The aim of the project is to create a dashboard that integrates data of different important aspects in the life of people with Diabetes type 1 involving glucose levels, activity information, insulin intake, food intake, and emotional status. It will become a part of the GameBus [1] platform, a platform that promotes a mentally and physically healthy lifestyle in a playful manner.

The dashboard shall be able to visualize data from the separate important aspects in a simple and intuitive way, but showing the correlations and interactions between these aspects will also be an important goal of the Diabetter dashboard. The users need to get insight into their glucose levels, but also into how food, insulin, activity, and emotion may influence these levels. Besides this, the integration with GameBus may enable more ways to encourage the users to live a healthier life with Diabetes.

1.3 List of definitions and abbreviations

Some of the used terms, phrases, and abbreviations might be ambiguous. Therefore we include all relevant definitions in Table 1 and 2.

1.3.1 Definitions

A1C estimate	Average blood glucose level over the past 3 months.
Endpoint	Entry point to a service.
Module	Independent unit which is part of a more complex structure.
Credentials	Information used for authentication.
Framework	An abstraction which provides generic functionality.
Ecosystem	A network of interconnected systems.

Table 1: Definitions of terms that are used in this document

1.3.2 Abbreviations

CGM	Continuous glucose monitor
TU/e	Technical university of Eindhoven
API	Application programming interface
WHO	World Health Organisation
JWT	JSON Web Token
SQL	Structured Query Language
MSAL	Microsoft Authentication Library
CSV	Comma-Separated Values (file format)
SEP	Software Engineering Project
XML	Extensible Markup Language
TK	Translation Key

Table 2: Definitions of abbreviations that are used in this document

1.4 List of references

References

[1] Gamebus. https://blog.gamebus.eu/. Accessed: 04-05-2021.

[2] Glooko. https://www.glooko.com/about/. Accessed: 16-05-2021.

[3] American Diabetes Association. Good to know: Factors affecting blood glucose. *Clin Diabetes*, 36,2(202):1, 2018.

[4] Glucome. http://www.glucome.com/about-us/overview. Accessed: 24-05-2021.

[5] Nightscout. http://www.nightscout.info/. Accessed: 28-06-2021.

[6] World health organisation. https://www.who.int/about. Accessed: 24-05-2021.

[7] Onedrive. https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage. Accessed: 29-06-2021.

[8] Microsoft graph api. https://docs.microsoft.com/en-us/graph/use-the-api. Accessed: 25-06-2021.

[9] Oauth2. http://oauth.net/2/. Accessed: 25-06-2021.

[10] Excel workbooks and charts api. https://docs.microsoft.com/en-us/graph/excel-concept-overview. Accessed: 25-06-2021.

[11] Nightscout setup. https://nightscout.github.io/nightscout/new_user/. Accessed: 28-06-2021.

1.5 Overview

This document consists of four chapters. Chapter two describes relation of the Diabetter to the current, past and future projects, as well as, the relation to the other projects. Moreover, a general overview of the function and purpose of the product is given. Additionally, overview of the environment is given. Chapter three describes modular and logical program structure for both front-end and back-end. Furthermore, description of the persistent data is provided together with relationships with external interfaces and design rationale. Finally, Chapter four provides performance measurements and a summary of computer resources needed to build, operate and maintain the software.

2 System overview

2.1 Relation to current projects

Diabetter is an application that visualizes different factors of interests for diabetes type-1 patients. Diabetter can be closely associated with Glooko [2], which is a free closed source web/phone application that provides personalized insights to improve diabetes management. Diabetter, in turn, aims at providing personalized insights to an average user with a basic understanding of diabetes in order to enhance awareness of how body responses to a certain daily decisions. Furthermore, Diabetter takes into consideration emotional status of the user which according to the American Diabetes Association [3] is one of the key factors that affects glucose level. Lastly, Diabetter focuses only on diabetes type-1 patients instead of focusing on all other group types at the same time. Therefore, a more precise analysis of data is possible.

Another comparable application is GlucoMe [4] which is also a free closed source diabetes management platform that uses mobile application to provide insights to diabetes patients. One of the main differences between GlucoMe and Diabetter is that GlucoMe uses artificial intelligence algorithms to provide personalized treatment recommendations. Diabetter, in turn, focuses on providing useful insights based on the data rather than recommendations. In addition, GlucoMe uses it's own blood glucose monitor which needs to be purchased to start using the application. Diabetter, on the other hand, integrates various CGM via Nightscout[5] API, as well as, allows users to import data manually using the dashboard.

2.2 Relation to predecessor and successor projects

Diabetter will be implemented as part of the TU/e GameBus platform, which provides a personalized gaming experience to promote a mentally and physically healthy lifestyle in a playful manner.
There are currently no plans for successor projects. However, since Diabetter code will be available to the GameBus developers they will have the ability to extend or modify source code and therefore possible successor projects depends on the decision made by GameBus.

2.3 Function and purpose

According to the WHO [6] the number of people with diabetes rose from 108 million in 1980 to 422 million in 2014. That indicates that the number of people with diabetes continuously increases every year. As the result, need for a diabetes management solutions emerged. However, currently there is no platform that incorporates together glucose data and trends, activity data, food consumption, and emotional status.

Diabetter will incorporate all the above-mentioned factors of interests in the user-friendly interactive dashboard to provide better insights for diabetes management. Following functionalities will be incorporated:

- **Providing of personalized diabetes insights:** Diabetter supports various visualization types that in combination with statistics provide personalized insights for a better diabetes management.
- **Integrating different factors of interest:** Diabetter supports integration of multiple factors of interest for diabetes patients.
- **Importing data:** Diabetter provides several import options to facilitate accumulation of data.

2.4 Environment

Diabetter will be a web-application that supports both English and Dutch localization. It will run on Google Chrome version 90 and later. There are three types of users: general user, super-user and developers

- **General users:** General users are users that directly interacts with the dashboard. They provide data to the dashboard by connecting their CGM to the dashboard via Nightscout or by uploading it manually using OneDrive client[7] or using the dashboard directly.
- **Supervisors:** Supervisors are subset of general users that have rights to view or modify general user's data. Moreover, supervisors can use all features of the dashboard's to explore their own data.
- **Developers:** Developers are mainly people from GameBus platform. They have partial access to some of the user's personal data. A developer has the ability to add or modify Diabetter functionality.

2.5 Relation to other systems

Diabetter is in its essence a tool for visualising data of Diabetes patients and lets them have a thorough insight into multiple variables generated in their daily lives. Hence, it consumes data from a plethora of sources and combines it into rigorous depictions. It is to a large extent a natural addition to the Gamebus platform, heavily depending on it.

Gamebus As a platform it provides a connected experience for its users and promotes healthy living by competing for achievements with you friends and family. It does that through interfacing with many applications that are heavily invested in harvesting health related data. It distinguishes between three categories of activity - physical, social and cognitive. Diabetter interacts with the open API of this platform and extract all personal physical data for processing and visualization to the users. Applications from which monitored data might come into the platform are:

- Samsung smart watch
- Medisana
- Polar
- Withings
- Fitbit
- Moves
- Wahoo fitness
- Ghostrace
- Endomondo
- Runkeeper

- Strava
- Smashrun
- Garmin
- Selfcarecan

Gamebus is used by Diabetter as its main storage. Every piece of information that is made available by the user to Diabetter is then sent to Gamebus after processing.

Eetmeter This is a platform for entering food consumption. Each meal is translated into nutrients. This data at the moment of writing is not available through an open API, hence Diabetter accepts an upload from an XML file that is exported from Eetmeter.

OneDrive Microsoft OneDrive is a file hosting service, capable of hosting many different types of files and sharing these files with other users. Hosted files can also be reached by third party applications through the Microsoft Graph API. Third parties can register their application for free to use this API.

Excel With access granted from the user, Diabetter is able to extract specific files related to food intake that the user creates and logs manually on their own. This file has to adhere to a certain standard such that it can be successfully parsed and translated.

NightScout It is an open source project which allows real time access to a CGM data via personal website, smartwatch viewers, or apps and widgets available for smartphones. It provides an integrable API with which Diabetter interfaces. Hence, it allows users to automatically transfer their data to the the developed system rather than manually filling in their glucose levels.

3 System architecture

3.1 Architectural design

The design of Diabetter consists of two major components: the front-end, responsible for handling the user interface and visualizing the data, and the back-end, responsible for parsing, storing and retrieving the data. These components are interconnected through an API exposed by the back-end to the front-end.

3.1.1 Front-end

The front-end part of Diabetter was built using the Vue.js JavaScript framework, a 'progressive framework for building user interfaces'. The application consists of a navigation bar and views that are loaded based on the user's active state. Furthermore, each view is composed of several components that fill up available space. Overview of these modules is depicted in the Figure 1. These modules will be discussed further in this section.

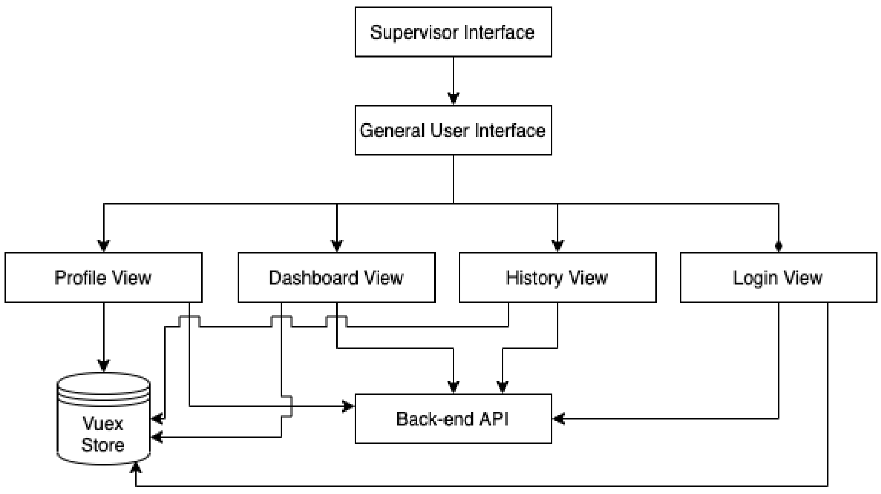


Figure 1: The composition of front-end modules

- **General User Interface:** The General User Interface provides all the necessary functionality to provide personalized diabetes insights to a user. This interfaces uses Dashboard and History views to display all respective content once the user gets authorized in the system.
- **Supervisor Interface:** The Supervisor Interface is an extension of General User Interface as it provides similar functionality to a user. However, the main difference between these interfaces is that the Supervisor Interface provides the ability to view or/and modify data of a user that is being supervised. Following interface makes use of Dashboard and History views.
- **Dashboard View:** The Dashboard View displays overview visualization and various statistics based on the user provided data.

- **Profile view:** The Profile View provides the ability to view, modify, and delete data from the GameBus database using the back-end API.
- **History view:** While the Dashboard view provides insights through charts and aggregates, users can view their data through tables when they access the history view. This provides the user with easier filtering and insights which would not be possible in a chart.
- **Back-end API:** The Back-end API module provides communication between Gamebus API, server and client.
- **Vuex store:** Vuex store is state management pattern and library that serves as the centralized store for all of the components in the application.

3.1.2 Back-end

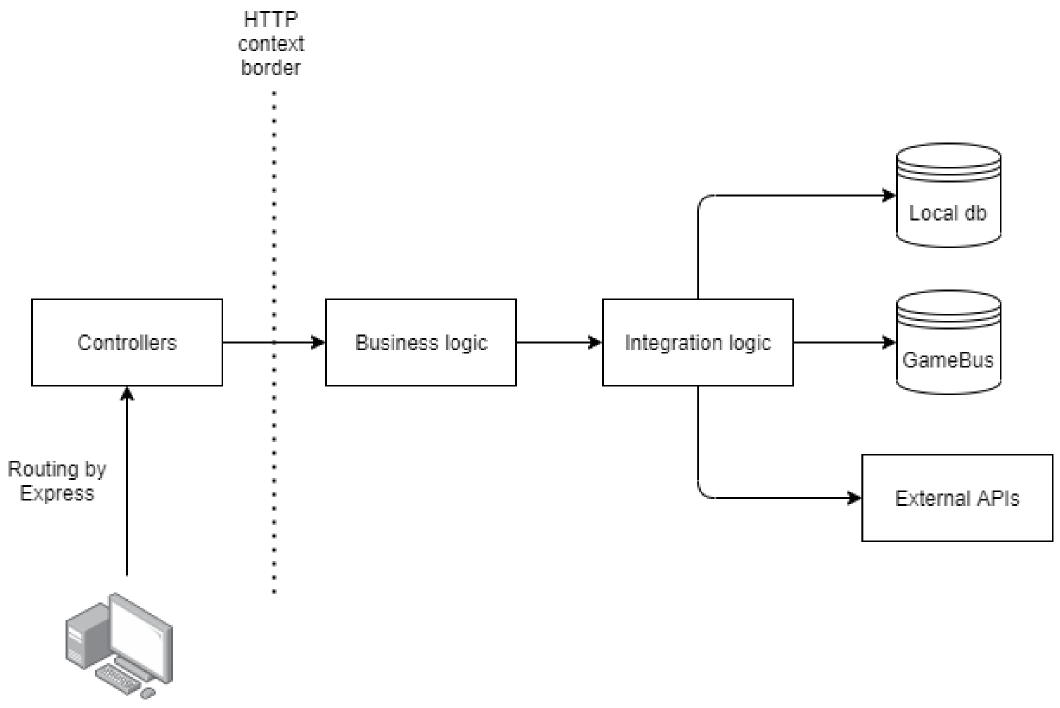


Figure 2: Back-end 3-layer design

The back-end handles all incoming API requests. A 3-layer design is used to decouple HTTP context, internal business logic and external services. The first layer consists of the controllers which handle all incoming HTTP requests and makes use of the Express objects to retrieve the required data from the HTTP calls. Passed credentials are, if required, verified. The data is then checked, passed to the second layer and based on the returned result a response is sent back to the user. The second layer only accepts well-defined data structures (i.e. primitive types like strings or TypeScript interfaces). In this layer the business logic is performed. The third layer is the layer between the internal services and the external services (not being the front-end), like databases and external APIs. It is implemented in such a way that integration details like for example the structure of external APIs does not need to be taken into account in the internal business logic.

3.1.2.1 Controllers

The controller gets the required data from the HTTP request, processes it and returns a response to the user. Every controller is connected to a specific route and is accessed through this route from the outside world. Before the controller itself is called, middleware may do some preprocessing on the request data and deny the request if necessary.

Input validation

For requests to be processed properly, all relevant data must be included and formatted such that it can be understood by the server. The controller must verify if this condition is met before executing the request.

Authentication

For some requests it is required that the user is authenticated, not only to protect local resources, but also to have the right credentials to access the GameBus API. Every request requiring authentication must come with a Json Web Token (JWT) included. This token contains an access token, a refresh token and a player id, which give access to the GameBus user account linked to the Diabetter profile for which the requests are made. The content of the token cannot be modified by the user and can only be issued by our server. The controller verifies the integrity of the token and decodes the credentials.

3.1.2.2 Business logic

The business logic is responsible for tasks which do not directly rely on HTTP context details and also do not directly rely on external services. One of the main tasks of the business logic is transforming data such that it can be retrieved, stored and updated, but without focusing on low-level details.

File parsing

Files uploaded by the user must be transformed such that the way the content is formatted no longer matters. In this way, comparable content formatted in a different way can be handled by the same modules of the business logic. This increases modularity, since it gets easier to add new file formats. Diabetter supports CSV, Excel and XML files.

Data parsing

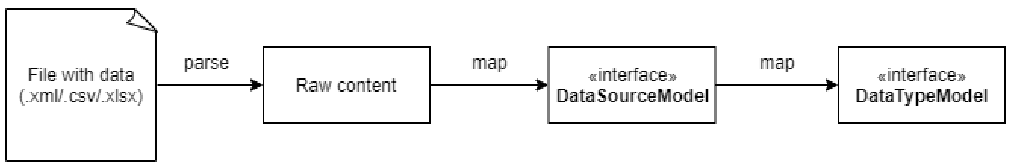


Figure 3: Data parsing steps, note that the first step is file parsing which is described earlier.

The data must be parsed such that it can be stored in GameBus and be visualized by our front-end. After the file parser removed file specific formatting, the data parser maps the raw data to the interfaces of the data sources we support. These sources are: Abbott, Eetmeter, Food diary and Nightscout. If for some reason data is missing or invalid, an error will be thrown.
In the next step, the source-mapped data is mapped again, but this time to the interfaces used for storage and the visualizations. These interfaces are different from the interfaces of the sources, as we do not need all the provided data. We support 5 specific types of data: food, glucose, mood, insulin and exercise. For all of these we have an interface available. Since the source mapping step already filtered most deviations out, less checks are required. The final mapping will be used by all the other components of the back-end.

Login logic

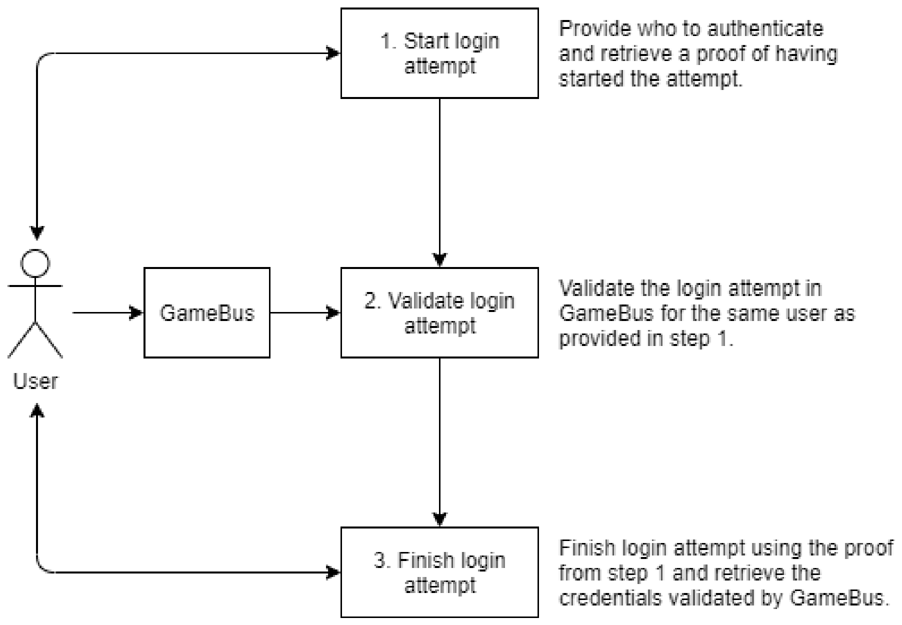

1. Start login attempt
Provide who to authenticate and retrieve a proof of having started the attempt.
User
GameBus
2. Validate login attempt
Validate the login attempt in GameBus for the same user as provided in step 1.
3. Finish login attempt
Finish login attempt using the proof from step 1 and retrieve the credentials validated by GameBus.

Figure 4: The three steps required to make a successful login attempt. In step 1, the user retrieves a login token after entering the email address to login with. In step 2, the user logs into the GameBus account associated to the email address in step 1 and connects the Diabetter data provider. In step 3, the user exchanges the login token retrieved in step 1 for the GameBus credentials.

The login procedure starts with generating and issuing a login token after an email-address is entered by a user. Then, a call from GameBus made on command of the same user must be retrieved by Diabetter to get the required GameBus credentials. The email-address of the GameBus account used to make the call must match the email-address entered earlier by the user. The last step is to show the earlier issued login token again and exchanging it for GameBus credentials encapsulated in a JWT.
The task of the business logic is to coordinate this authentication process. The controller can obviously do very little of the authentication before the user has logged in, so possible intruders must be stopped too by the business logic.

3.1.2.3 Integration

The integration handles the communication with all external APIs and databases. It mainly focuses on low-level details.

GameBus

GameBus requires to retrieve calls formatted in a very specific way. The integration layer maps the data of the business logic to interfaces which can directly be pushed to GameBus. The same happens the other way around, data from GameBus is mapped to the interfaces used internally in Diabetter. Headers, query parameters and URLs are also generated such that these meet what is required by GameBus.

OneDrive

The OneDrive API, part of the Microsoft Graph API [8], uses the OAuth 2.0 standard [9] for authentication. Diabetter has been registered in the Microsoft ecosystem such that it can be authorized by users to access resources managed by Microsoft. An access token is used to access these resources. This token can be retrieved in two ways.
If the user has used the OneDrive API in Diabetter before and so already has a refresh token, it can be exchanged for an access token and a new refresh token. This can be repeated until the user decides to revoke the authorization for Diabetter to use the OneDrive API.
If it is the first time the user uses the OneDrive API in Diabetter, meaning no refresh token is available, an authorization code is required to get both an initial access token and refresh token. This code can be retrieved by logging into a Microsoft-account and authorizing Diabetter.
Using the access token, files stored on OneDrive can be accessed. We use the Excel workbooks and charts API [10] to access Excel files in a straightforward way.

Nightscout

Nightscout[5] is a third party service to which users of Continuous Glucose Monitors (CGMs) can connect, such that their data is stored in the Nightscout database. To connect to this service, the users have to setup their own Nightscout account[11] which can be accessed through a specific web address (for example: https://nightscout-sep.herokuapp.com). If this Nightscout web address is given to Diabetter by the user, no further authentication is needed to access the Nightscout API for this account. With this API, both Nightscout glucose entries can be retrieved through the /entries endpoint, which contain glucose levels for a certain timestamp, as well as Nightscout treatments, that can be retrieved through the /treatments endpoint and can contain either insulin or food data for a certain timestamp. Documentation for the API can be found under <nightscout web address>/api-docs/.

Local database

Almost all data is stored in GameBus. Only a small amount of mostly temporary data is stored in a local database, which uses the SQLite engine. The database resides in a locally stored database file. At each server startup, it is validated if a database file with the correct tables is already on the disk. If not, a new database file, including the tables, will automatically be created and saved.

3.2 Logical model description

3.2.1 Front-end

This section describes front-end logical model by means of UML class diagram. An overview of used UI components and repositories is given. To improve readability of the UML diagram we decided to leave out all methods and variables. These will be described in the following sub sections.

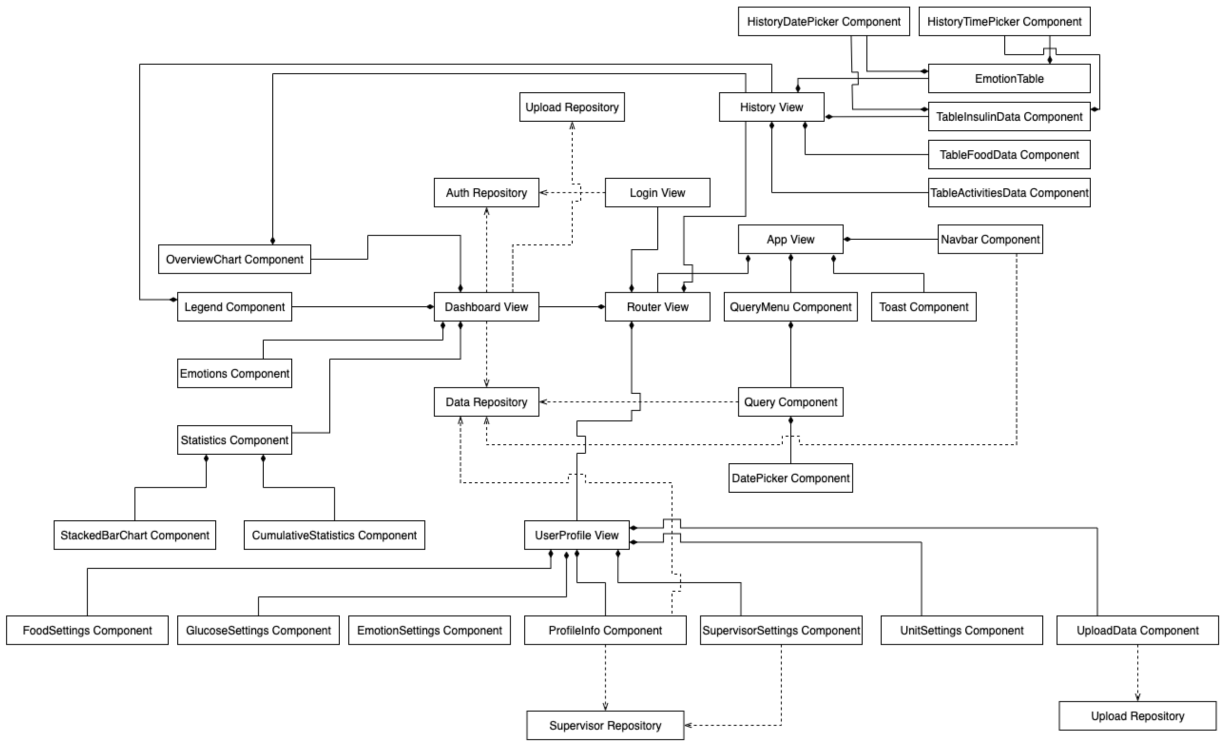


Figure 5: UML class diagram describing front-end modules

3.2.1.1 Repositories For the frontend-backend connections, we used a Client - Repository structure. In the Client.js file, a basic axios object is created with some default values, namely the base URL and the headers. Multiple repositories are made for specific parts of the project, like Authentication and Data fetching.

Auth The authentication repository features the following methods.

1. login(payload : any). This method is used for both requesting a login redirect to GameBus, but also for confirming after successfully linking GameBus with Diabetter.
2. uploadToken(payload : any). This method is used for uploading the user's token to the database on the backend. This token is then used to allow supervisors to fetch the data of the user(s) they

supervise.

3. getProfile(token : string). This method is used to fetch the user's profile data, like their first and last name. This data is then used to populate the information in the profile-view and the navigation bar.
4. updateUser(payload : any, token : string). This method is used to update the user's profile data when they edit it in the profile page.

Data The data repository contains methods for adding, modifying, removing and fetching user's data from the back-end:

1. fetch(payload : any, token : string). This method is used for all calls to the back-end relating to the fetching of data.
2. deleteItem(payload : any, token : string). This method us used to remove specified entry from the back-end.
3. postItem(payload : any, token : string, endpoint : string). This method is used to add/modify entry in back-end.

Supervisor The supervisor repository features the following methods.

1. request(payload : any). This method is used for requesting to be supervised by the specified supervisor.
2. getChildren(payload : any). This method is used to fetch all the users that a supervisor supervises, in order to show them in the dropdown featured in the navigation bar.
3. getSupervisors(payload : any) This method is used to fetch all the supervisors for a child **that were not approved yet**.
4. deleteSupervisor(payload : any). This method is used to remove the permission to be supervised by the specified supervisor.
5. getApproved(payload : any). This method is used to fetch all **approved** supervisors for the logged-in user.
6. getRole(payload : any). This method checks if the currently logged in user is a supervisor or not, in order to showcase the right role in the Navigation bar / profile view ("User" or "Supervisor").
7. getToken(payload : any). This method fetches the token of the user that a supervisor wants to see the data of.

Upload The upload repository is used for everything regarding data-uploading (including the OneDrive connection, the NightScout connection and the actual uploading of data). The repository features the following methods.

1. upload(data : any, token : string). This method is used for uploading a Food Diary, Abbott or Eetmeter file to the backend.

2. fetchOneDriveToken(payload : any). This method is used to refresh the user's OneDrive-token using their HomeAccount-ID.
3. uploadOnedrive(payload : any, token : string). This method is used for uploading the OneDrive file to the backend.
4. connectNightscout(payload : any). This method is used for uploading the nightscout data to the backend.

3.2.1.2 Views

App App view is an entry point of application that initializes all the necessary views and components. It does not have any methods nor variables.

Dashboard Dashboard provides an overview of the user data.

Methods

- `async created() : void`
  Checks reminder options, connections to onedrive and nightscout, also fetches data from back-end on view render.
- `updateMinMax(parameters : any) : void`
  Updates `minMax` value which is passed to statistics component.
- `updateProportions(parameters : any)`
  Update `proportions` value which is going to be passed to statistics component.
- `updateEmotions(parameters : any)`
  Update `emotions` value which is going to be passed to emotions component

Variables

+ `data : any`
  Data object that contains fetched data.

- `rendered : boolean`
  Boolean variable that represents whether fetched data was assigned to the `data` object.

+ `proportions : Array<number>`
  Array that contains selected proportion in data slider in overview visualization.
+ `minMax : Array<number | null>`
  Minimum and maximum timestamp of the fetched data.

History History view provides a detailed overview of the user's data.

Methods

- `async created() : void`
  Checks whether data object was passed to the `history` view.

Variables

- `+ data : any`
  Data object that contains fetched data.
- `- tab : null | number`
  Variable that keeps track of the currently displayed tab.

Login Login view is used to provide authorization methods in diabetter application

Methods

- `- async loginClicked() : void`
  Redirects user to the Gamebus platform in case such user exists in Gamebus.
- `- confirmLogin() : void`
  Links diabetter account with Gamebus.
- `- cancel() : void`
  Cancel authorization.

Variables

- `- show : boolean`
  Boolean variable that represents view state of the 'toast'.
- `- email : string`
  Entered email address of the user.
- `- enteredEmail : boolean`
  Boolean variable that checks whether email was entered in 'email' field.

UserProfile The parent view of all the profile components. This view contains the ProfileInfo, SupervisorSettings, GlucoseSettings, UploadData, EmotionSettings, FoodSettings and UnitsSettings components. They get displayed in a nice fashion using vue-bootstrap's rows and columns. *Methods*

- `- created() : void`
  When this view is created (by the user navigating to the profile section of Diabetter), this function retrieves the profile for the logged-in user.

3.2.1.3 Components

Router View Default Vue.js router that is used to render views of the application.

Toast Toast is used to display messages/notifications on the right bottom side of the screen.

Methods

- `- created(): void`
  Updates values in the component upon vuex state change.

Variables

- `show : boolean`
  Boolean variable that represents view state of the 'toast'.

- `messsage : string`
  Text of the displayed message.

- `color : string`
  Color of the displayed message.

- `timeout : number`
  Number of milliseconds for which messages is displayed.

Navbar Navbar component is used to display navigation bar of the application.

Methods

- `created() : void`
  Refresh current user settings and fetch user role by calling `refreshUser()` and `fetchUserRole()` methods.

- `checkData() : void`
  Check if data object is not empty.

- `historyClicked() : void`
  Check whether history view link was clicked. Prevent redirect in case there is no data available.

- `logout() : void`
  Logout from the diabetter account.

- `refreshUser() : void`
  Refresh user data.

- `async fetchUserRole() : boolean | void`
  Fetch user role. For example, supervisor or user.

Variables

+ `arousalIcon : string`
  Class of the icon tag representing arousal.

+ `valenceIcon : string`
  Class of the icon tag representing valence.

+ `data : any`
  Data object that contains fetched data.

- `profileData : any`
  Object that contains name and profile image of the user.

- `supervisor : boolean`
  Boolean that represents whether user status is set to supervisor or not.

- `children : Array<string>`
  Array that contains emails of the users that are being supervised by the current user.
- `childToSupervise : any`
  Object that contains settings for the user that are being supervise by the current user.
- `showing : boolean`
  Boolean variable that keeps track of dropdown that allows to switch between current user and users that are being supervised.

QueryMenu Query menu is implemented as dialog component which is used for filtering visualizations. Menu includes fields for filtering date, glucose, insulin, activity and emotional status.

Methods

- `async applyFiltering(): void`
  Applies selected filtering options to fetched data.
- `resetSelection() : void`
  Reset selection selected in a query menu.
- `updateParameters() : void`
  Update `parameters` object used to store selected query fields.
- `cancelFiltering() : void`
  Dismiss query menu dialog and reset all selected fields by calling `resetSelection()` method.
- `formatLabel(str : string) : string`
  Format attribute name for displaying selected field in a query menu.
- `setupParameters(parameters : any)`
  Setup parameters object before sending fetch request.
- `checkSelection(paramters : any)`
  Check parameters object for selected filtering options.

Variables

+ `filter : any`
  Object to keep track of the filter show/hide state.
+ `data : any`
  Data object that contains fetched data.
+ `filteredData : any`
  Data object that contains filtered data.
+ `date : any`
  Date object that contains start and end date the fetched data interval.

- `parameters : any`
  Object that contains state of all filtering fields.

+ `properties : any`
  Properties object that contains settings for filtering fields.

+ `reload : boolean`
  Boolean variable that indicated whether to update field label text.

Query Query component is used to represent type of the field, for example, whether it is a date field, numeric field or a string field

Methods

- `update(event : string, index : number, value : any) : void`
  Update or reset query menu fields.

Variables

- `model : Array<number>`
  Array that contains models used by fields to track changes.

DatePicker DatePicker is a special type of field that used vuetify date picker component

Methods

- `update() : void`
  Update value of the selected date field.

- `clear() : void`
  Clear selected input in date picker component.

Variables

- `dateMenu : boolean`
  Boolean variable that tracks state of the date picker pop up.

- `dateRange : any`
  Object that stores start and end date of the selected field.

- `attrs : Array<any>`
  Object that store attribute values for visualizing date picker component.

CumulativeStatistics CumulativeStatistics component contains respective statistics of the user data.

Methods

- `[min/max/average]Glucose()`
  Calculate the respective statistics for Glucose over the selected time frame.

- `total[Carbs/Calories/Insulin/BurntCalories]()`
  Calculate the total value of the specified kind over the selected time frame.

StackedBarChart StackedBarChart component contains respective chart visualization displaying time distribution of glucose data.

Methods

- `computeTimeDistribution(data : any) : Array<Array<number»`
  Computes time distribution of glucose data.
- `options(data : any) : any`
  Creates options object which is used to generate eCharts object.

Variables

+ `data : any`
  Data object that contains fetched data.
+ `filteredData : any`
  Data object that contains filtered data.

Emotions Emotions component used to gather emotional status data from user

Methods

- `async checkEmotionInput() : void`
  Checks user input and converts emotional status values from emoticon to numbers before making a request to the back-end.

Variables

- happinessOption : string
  Happiness emotion value collected from user.
- excitementOption : string
  Excitement emotion value collected from user.
- parameters : any
  Payload of the post requests to be sent to the back-end.

Legend Legend component contains overview chart legend items.

Methods

- `toggleLegendItem(type : string, key : string) : void`
  Toggle visibility of the selected item in an overview chart visualization.

Variables

+ `data : any`
  Data object that contains fetched data.
+ `filteredData : any`
  Data object that contains filtered data.

- `legend : any`
  Legend options for displaying legend markers and names.

- `chartInstance : eChart instance`
  Instance of the eCharts object.

OverviewChart OverviewChart component provides stacked visualizations of user data.

Methods

- `scaleValue(value : number, from : Array<number>, to : Array<number>) : number`
  Scales value to a specified range.

- `prepareData(data : any, model : string, properties : Array<any>) : Array<Array<any»`
  Set up provided data to eCharts format

- `createTooltipBody(marker : string, name : string, value : number) : string`
  Creates tooltip item.

- `prepareTooltip(params : any) : string`
  Prepares tooltip used in overview chart visualization.

- `renderInterval(params : any, api : any) : any`
  Creates custom shape for displaying duration of the activity shown in an overview chart visualization.

- `alignGluconeData(glucose : any, data : Array<any>) : any`
  Creates dummy points in the glucose data to align axis pointer in the overview chart visualization.

- `parseRangeString(str : string) : number`
  converts number in string format into number.

- `addUnits(data : any, idx : number, units : string) : string`
  Creates tooltip entry from data value and append provided units.

- `findMinMax(data : any) : Array<number>`
  Finds minimum and maximum timestamps in the data object.

- `options(data : any) : any`
  Creates options object which is used to generate eCharts object.

Variables

+ `data : any`
  Data object that contains fetched data.

+ `filteredData : any`
  Data object that contains filtered data.

+ `newTimeInterval : any`
  Time interval object which is used to filter visualization.

Statistics Statistics component provides overall statistics of the user data about glucose, carbs, exercises and insulin.

Variables

- `tab : any`
  Variable that keeps track of the currently displayed tab.

EmotionSettings The emotions component features a toggleable slider to switch on/off the emotions reminder.

Variables

- `boolAsk : boolean`
  Variable that keeps track of whether or not the emotions reminder should be displayed.

GlucoseSettings The GlucoseSettings component features sliders to edit the glucose-range settings and inputs for the thresholds.

Variables

+ `healthSettings : dictionary`
  Dictionary containing the values for the given keys.

SupervisorSettings The SupervisorSettings component provides the user with the means to manage their supervisor(s).

Methods

- `getRequested()`
  Gets the list of supervisors that have requested to supervise the logged-in user.

- `getApproved()`
  Gets the list of supervisors that have been approved by the logged-in user.

- `request(childEmail: string)`
  Request to supervise the provided child.

- `approve()`
  Approves the clicked supervisor-request.

- `reject()`
  Rejects the clicked supervisor-request.

- `remove()`
  Removes the permission to supervise for an already approved supervisor.

Variables

+ `supervisors : object`
  Object that contains all the *approved* supervisors for the logged-in user.

+ `requestedSupervisors : object`
  Object that contains all the supervisors that have not been approved yet.

ProfileInfo The ProfileInfo component provides the user with the means to manage their user settings, like age and weight.

Methods

- `- refreshUser()`
  Fetches the user data that is stored in the Vuex store

Variables

- `+ profileData : dictionary`
  Dictionary that contains all data regarding the logged-in user.

FoodSettings The FoodSettings component provides the user with the means to manage their food goals.

UnitSettings The UnitSettings component provides the user with the means to manage their preferred unit of measure, like mmol/L or mg/dL.

UploadData The UploadData component provides the user with options to connect some data endpoints (like OneDrive) or upload a dataset themselves.

Methods

- `- fileUpload()`
  Handles the direct uploading of a dataset
- `- connectOnedrive()`
  Starts the authentication process for a OneDrive account.
- `- uploadOnedrive()`
  Upload the file specified in the path to the backend of Diabetter.

Variables

- `+ items : object`
  Array containing the three different types of data to upload: Eetmeter, Food diary or Abbott.
- `+ nightscoutUrl : object`
  URL to the user's Nightscout account.
- `+ onedriveFileUrl : object`
  Locally saved value containing the path to the desired file in the user's OneDrive client.

TableActivitiesData TableActivityData is used to display detailed activity information.

Methods

- `selectActivity(activity : any) : void`
  Handles row click action for activity table.

- `convertExercises(data : any) : Array<any>`
  Converts initial data object to a structure used in a activity table.

Variables

+ `data : any`
  Data object that contains fetched data.

+ `filteredData : any`
  Data object that contains filtered data.

- `items : Array<string>`
  Local filter operations for activity table.

- `headers : Array<any>`
  Activity table headers.

- `exercises : Array<any>`
  Array containing activity entries.

- `name : string`
  Name of the activity provided by a user.

- `type : string`
  Type of the activity provided by a user.

- `startTime : string`
  Start time of the activity provided by a user.

- `endTime : string`
  End time of the activity provided by a user.

- `startDate : string`
  Start date of the activity provided by a user.

- `endDate : string`
  End date of the activity provided by a user.

- `calories : string`
  Number of calories of the activity provided by a user.

- `startTimeFilter : string`
  Start time entry used to filter table.

- `endTimeFilter : string`
  End time entry used to filter table.

- `startDateFilter : string`
  Start date entry used to filter table.

- `endDateFilter : string`
  End date entry used to filter table.

- `caloriesFilter : string`
  Calories entry used to filter table.

TableFoodData TableFoodData is used to display detailed food information.

Methods

- `selectFood(food : any) : void`
  Handles row click action for food table.

- `convertFood(data : any) : Array<any>`
  Converts initial data object to a structure used in a food table.

Variables

+ `data : any`
  Data object that contains fetched data.

+ `filteredData : any`
  Data object that contains filtered data.

- `items : Array<string>`
  Local filter operations for food table.

- `headers : Array<any>`
  Food table headers.

- `food : Array<any>`
  Array containing food entries.

- `type : string`
  Type of the food provided by a user.

- `time : string`
  Time of the food provided by a user.

- `date : string`
  Date of the food provided by a user.

- `carbs : string`
  Number of carbs of the food intake provided by a user.

- `calories : string`
  Number of calories of the food provided by a user.

- `glycemicIndex : string`
  Glycemic index of the food provided by a user.

- `timeFilter : string`
  Time entry used to filter table.

- `dateFilter : string`
  Date entry used to filter table.

- `carbsFilter : string`
  Carbs entry used to filter table.

- `caloriesFilter : string`
  Calories entry used to filter table.

TableInsulinData TableInsulinData is used to display detailed insulin information.

Methods

- `selectInsulin(insulin : any) : void`
  Handles row click action for insulin table.

- `convertInsulin(data : any) : Array<any>`
  Converts initial data object to a structure used in a insulin table.

- `displayType(type : number) : string`
  Convert value of insulin type to a respective string.

- `async checkInsulinInput(editing : boolean) : void`
  Checks insulin fields in editing mode and post new settings upon change approval.

- `editItem(item : any) : void`
  Assigns field value to an object upon input.

- `close() : void`
  Closes editing pop up in insulin table.

- `save() : void`
  Save modified fields in insulin table.

- `showDeleteDialog() : void`
  Shows delete item pop up in insulin table.

- `deleteItemConfirm() : void`
  Confirms deletion of the item from table.

- `closeDelete() : void`
  Closes delete pop up in insulin table.

- `updateInsulinTable() : void`
  Updates values in insulin table.

+ `deleteItem(parameters : any) : void`
  Deletes item from insulin table.

Variables

- `items : Array<string>`
  Local filter operations for insulin table.

- `headers : Array<any>`
  Insulin table headers.
- `insulinData : Array<any>`
  Array containing insulin entries.
- `editing : boolean`
  Boolean that represents state of the entry. E.g. whether entry is being edited or not.
- `dialog : boolean`
  Boolean that represents state of the editing pop up.
- `time : string`
  Time of the insulin provided by a user.
- `date : string`
  Date of the insulin provided by a user.
- `dialogDelete : boolean`
  Boolean that represents state of the delete pop up.
- `editedItem : any`
  Object that contains modifications of specific item.
- `defaultItem : any`
  Object that contains current data of the modified item.
- `types : Array<string>`
  Array containing types of the insulin.
- `timeFilter : string`
  Time entry used to filter table.
- `dateFilter : string`
  Date entry used to filter table.
- `typeFilterItems : Array<string>`
  Options for filtering type of the insulin.
- `amount : string`
  Amount of the insulin provided by a user.
- `amountFilter : string`
  Amount entry used to filter table.
- `typeFilter : string`
  Type entry of insulin item used to filter table.

EmotionTable EmotionTable is used to display detailed emotion status information.

Methods

- `selectEmotion(emotion : any) : void`
  Handles row click action for emotion table.

- `convertEmotions(data : any) : Array<any>`
  Converts initial data object to a structure used in a emotion table.

- `async checkEmotionInput(editing : boolean) : void`
  Checks emotion fields in editing mode and post new settings upon change approval.

- `editItem(item : any) : void`
  Assigns field value to an object upon input.

- `close() : void`
  Closes editing pop up in emotion table.

- `save() : void`
  Save modified fields in emotion table.

- `showDeleteDialog() : void`
  Shows delete item pop up in emotion table.

- `deleteItemConfirm() : void`
  Confirms deletion of the item from table.

- `closeDelete() : void`
  Closes delete pop up in emotion table.

- `updateEmotionTable() : void`
  Updates values in emotion table.

+ `deleteItem(parameters : any) : void`
  Deletes item from emotion table.

Variables

- `items : Array<string>`
  Local filter operations for insulin table.

- `headers : Array<any>`
  Insulin table headers.

- `emotions : Array<any>`
  Array containing emotions entries.

- `editing : boolean`
  Boolean that represents state of the entry. E.g. whether entry is being edited or not.

- `dialog : boolean`
  Boolean that represents state of the editing pop up.

- `time : string`
  Time of the emotion provided by a user.

- `date : string`
  Date of the emotion provided by a user.

- `dialogDelete : boolean`
  Boolean that represents state of the delete pop up.

- `editedItem : any`
  Object that contains modifications of specific item.

- `defaultItem : any`
  Object that contains current data of the modified item.

- `emotionValues : Array<string>`
  Array containing different values for emotion types.

- `timeFilter : string`
  Time entry used to filter table.

- `dateFilter : string`
  Date entry used to filter table.

- `happinessFilter : Array<string>`
  Options for filtering happiness emotion.

- `excitementFilter : Array<string>`
  Options for filtering excitement emotion.

HistoryDatePicker HistoryDatePicker is used to pick dates using a table located in `History` view.

Variables

+ `date : any`
  Date object that stores current date if such exists.

- `dateMenu : boolean`
  Boolean that represents state of the date picker. E.g. shown/hidden.

- `insulinDate : any`
  Date object that stores date entered by user.

HistoryTimePicker HistoryTimePicker is used to pick time using a table located in `History` view.

Variables

+ `time : any`
  Time object that stores current time if such exists.

- `timeMenu : boolean`
  Boolean that represents state of the time picker. E.g. shown/hidden.

- `insulinTime : any`
  Date object that stores time entered by user.

3.2.2 Back-end

In this section, the components of the backend are explained. An overview is given to show the relation between the main modules of the program. For exchanging data, the modules make use of interfaces, types and enums besides the primitives already included in TypeScript. This makes it easier to know what data is expected to be provided to a method. Classes are used to couple related functionality into modules and keep the data valid inside these modules.

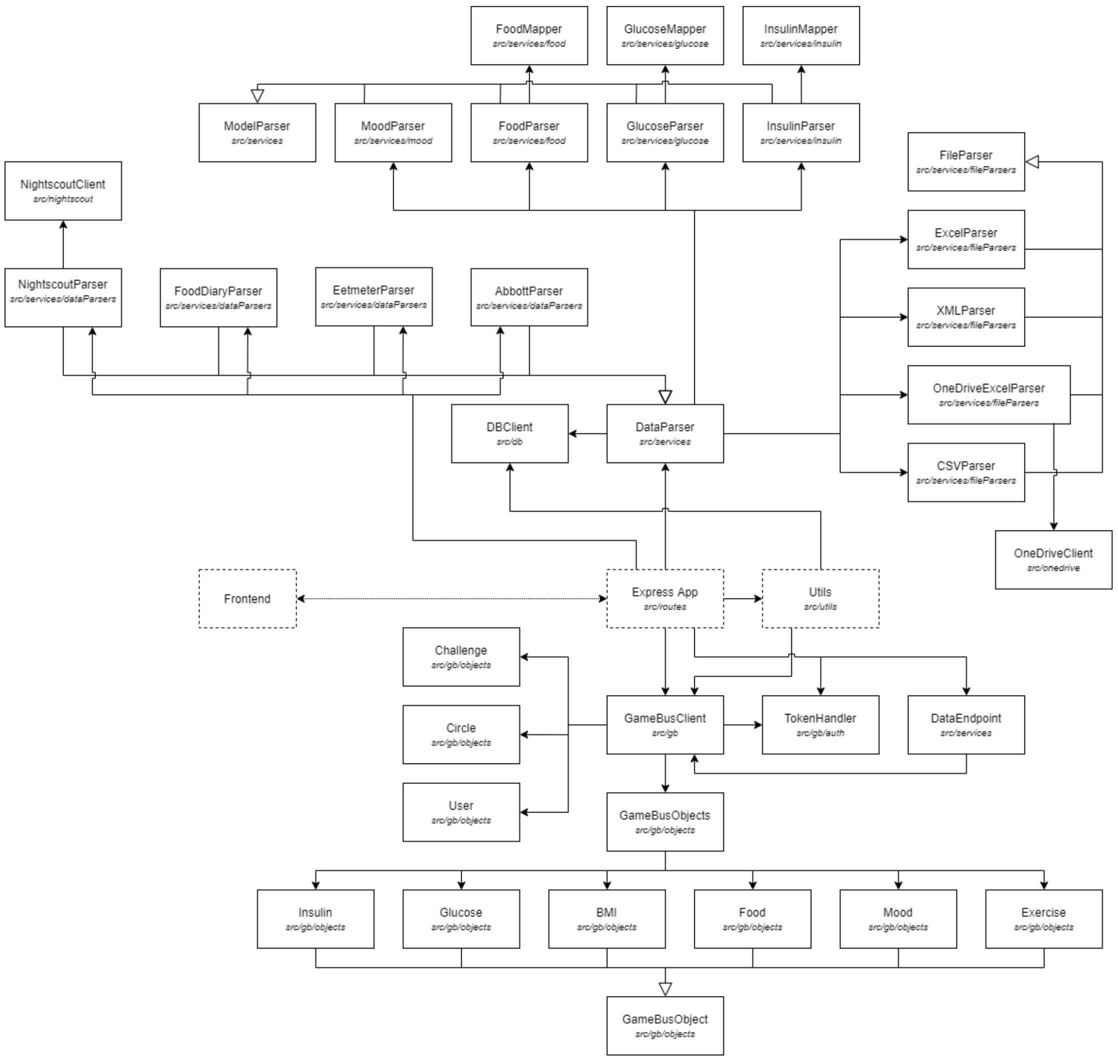

FoodMapper src/services/food
GlucoseMapper src/services/glucose
InsulinMapper src/services/insulin
ModelParser src/services
MoodParser src/services/mood
FoodParser src/services/food
GlucoseParser src/services/glucose
InsulinParser src/services/insulin
FileParser src/services/fileParsers
NightscoutClient src/nightscout
ExcelParser src/services/fileParsers
NightscoutParser src/services/dataParsers
FoodDiaryParser src/services/dataParsers
EetmeterParser src/services/dataParsers
AbbottParser src/services/dataParsers
XMLParser src/services/fileParsers
OneDriveExcelParser src/services/fileParsers
DBClient src/db
DataParser src/services
CSVParser src/services/fileParsers
OneDriveClient src/onedrive
Frontend
Express App src/routes
Utils src/utils
Challenge src/gb/objects
Circle src/gb/objects
User src/gb/objects
GameBusClient src/gb
TokenHandler src/gb/auth
DataEndpoint src/services
GameBusObjects src/gb/objects
Insulin src/gb/objects
Glucose src/gb/objects
BMI src/gb/objects
Food src/gb/objects
Mood src/gb/objects
Exercise src/gb/objects
GameBusObject src/gb/objects

Figure 6: UML class diagram describing the back-end. Only relevant classes are included.

3.2.2.1 Interfaces and types

ActivityModel
Interface which holds information about a generic GameBus activity.

- `activityId: number`
  Id of the GameBus activity
- `timestamp: number`
  The moment the activity was added as Unix timestamp
- `id: number`
  Id the property instance
- `translationKey: number`
  Translation key for the model
- `value: number | string`
  Value of the model
- `property: ActivityProperty`
  Property of the activity

ActivityProperty
Interface which holds information about an activity property.

- `id: number`
  Id of the property
- `translationKey: string`
  Translation key of the property
- `baseUnit: string`
  Base unit to use for the property
- `inputType: string`
  Input type to use for the property

BMIModel
Interface which holds information about BMI related information.

- `timestamp: number`
  The moment the BMI was retrieved as Unix timestamp
- `activityId: number | undefined`
  Id of GameBus activity
- `weight: number | null`
  Weight in kg, null if unknown
- `length: number | null`
  Height in cm, null if unknown

- `age: number | null`
  Age in years, null if unknown
- `gender: string | null | undefined`
  Either m, f or o, null if unknown
- `waistCircumference: number | undefined | null`
  In cm, null or undefined if unknown
- `bmi: number | null | undefined`
  In kg/m^2, null or undefined if unknown

ExerciseModel
Interface which holds information about an exercise.

- `timestamp: number`
  The moment the exercise was added as Unix timestamp
- `name: string`
  Sensible name of activity (regex of type)
- `type: string`
  Activity type
- `duration: number | null | undefined`
  If relevant, exercise duration in seconds
- `steps: number | null | undefined`
  If relevant, number of steps
- `distance: number | null | undefined`
  If relevant, distance in meters
- `calories: number | null | undefined`
  If relevant, calories burned in in kcal
- `groupSize: number | null | undefined`
  If relevant, group size
- `penalty: number | null | undefined`
  Only relevant for air hockey, between 0 and 100
- `score: number | null | undefined`
  Only relevant for air hockey
- `maxSpeed: number | null | undefined`
  If relevant, maximum speed in m/s
- `avgSpeed: number | null | undefined`
  If relevant, average speed in m/s
- `maxHeartrate: number | null | undefined`
  If relevant, maximum heartrate in bpm

- `avgHeartrate: number | null | undefined`
  If relevant, average heartrate in bpm
- `minHeartrate: number | null | undefined`
  If relevant, minimum heartrate in bpm
- `heartrate: number | null`
  If relevant, recorded heartrate in bpm
- `activityId: number | undefined`
  Id of related GameBus activity

GlucoseModel
Interface which holds information about a glucose measurement.

- `timestamp: number`
  The moment the measurement was made as Unix timestamp
- `glucoseLevel: number`
  Measured glucose level (in mmol/L)
- `activityId: number | undefined`
  Id of related GameBus activity

InsulinModel
Interface which holds information about an insulin injection.

- `timestamp: number`
  The moment the injection was made as Unix timestamp
- `insulinAmount: number`
  Injected amount of insulin
- `insulinType: InsulinType`
  Type of insulin injected
- `activityId: number | undefined`
  Id of related GameBus activity

MoodModel
Interface which holds information about a mood measurement.

- `timestamp: number`
  The moment the mood was given as Unix timestamp
- `arousal: number`
  Arousal level given

- `valence: number`
  Valence level given
- `activityId: number | undefined`
  Id of related GameBus activity

FoodModel
Interface which holds information about a meal. A meal has 3 basic properties; time of meal, calories of meal, description of meal (optional). Additionally, more properties can be included (in grams).

- `timestamp: number`
  The moment the meal was eaten as Unix timestamp
- `carbohydrates: number`
  Carbohydrates in the meal
- `meal_type: MEAL_TYPE | null | undefined`
  Indicates breakfast, lunch, snack etc.
- `glycemic_index: number | null | undefined`
  Glycemic index of the meal
- `fat: number | null | undefined`
  Fat in the meal
- `saturatedFat: number | null | undefined`
  Saturated fat in the meal
- `proteins: number | null | undefined`
  Proteins in the meal
- `fibers: number | null | undefined`
  Fibers in the meal
- `salt: number | null | undefined`
  Salt in the meal
- `water: number | null | undefined`
  Water in the meal
- `sugars: number | null | undefined`
  Sugars in the meal
- `description: string | null | undefined`
  Description of the meal
- `activityId: number | undefined`
  Id of related GameBus activity

EetmeterData
Interface which holds information about Eetmeter data.

- `Consumpties: Consumpties`
  Consumptions in the Eetmeter data.

Consumpties
Interface which holds information about an Eetmeter consumptions.

- `Attributes: any[]`
  Attributes of the consumptions
- `Consumptie: Consumptie[]`
  Consumption list

Consumptie
Type which holds information about an Eetmeter consumption.

- `Attributes: Periode`
  Attributes of the consumption as a Periode, which contains a single key called Periode with a string value.
- `Datum: Datum`
  Moment when the consumption took place
- `Product: Product`
  Consumed product, given as a Product which contains a single key called Value with the product name
- `Nutrienten: Nutrienten`
  Nutrients in the consumption

Nutrienten
Interface which holds information about the nutrients in an Eetmeter consumption. All value types of values in this interfaces are interfaces with a single key called Value and with a numeric value.

- `Koolhydraten: Koolhydraten`
  Carbohydrates in grams
- `Energie: Energie`
  Energy in grams
- `Vet: Vet`
  Fat in grams
- `VerzadigdVet: VerzadigdVet`
  Saturated fat in grams

- `Zout: Zout`
  Salt in grams
- `Water: Water`
  Water in grams
- `Suikers: Suikers`
  Sugars in grams

Datum
Interface which holds information about a date in Eetmeter.

- `Dag: number`
  Number of the day
- `Maand: number`
  Number of the month
- `Jaar: number`
  Number of the year

ConnectionData
Interface which holds information about a user connection to GameBus. Used for debugging.

- `authToken: string`
  Authentication token to use for the connection
- `playerID: number`
  Player id to use for the connection
- `userID: number`
  User id to use for the connection

ActivityPOSTData
Interface which holds information about an activity post to GameBus.

- `gameDescriptorTK: string`
  Game description translation key
- `dataProviderName: string`
  Data provider name
- `image: string`
  Link to activity image
- `date: number`
  Activity moment as Unix timestamp

- `propertyInstances: PropertyInstancePOST[]`
  Property instances
- `players: number[]`
  Ids of players involved

PropertyInstancePOST
Interface which holds information about a GameBus property instance using translation keys.

- `propertyTK: string`
  Property translation key
- `value: any`
  Value to set

IDActivityPOSTData
Interface which holds information about an activity post to Gamebus with IDPropertyInstancePOSTs instead of PropertyInstancePOSTs.

- `gameDescriptor: number`
  Game description translation key
- `dataProvider: number`
  Data provider name
- `image: string | undefined`
  Link to activity image
- `date: number`
  Activity moment as Unix timestamp
- `propertyInstances: IDPropertyInstancePOST[]`
  Property instances
- `players: number[]`
  Ids of players involved

IDPropertyInstancePOST
Interface which holds information about a GameBus property instance using ids instead of translation keys.

- `property: number`
  Property id
- `value: any`
  Value to set

ActivityGETData
Interface which holds information about a retrieved GameBus activity.

- `id: number`
  Id of activity
- `date: number`
  Moment of activity as Unix timestamp
- `isManual: boolean`
  Whether the activity has been manually added
- `group: any | undefined`
  If set, the group of which the activity is part
- `image: any | undefined`
  If set, an activity image
- `creator: UserReference`
  Creator of the activity
- `player: UserReference`
  Player involved in the activity
- `gameDescriptor: GameDescriptorReference`
  Game descriptor
- `dataProvider: DataProviderReference`
  Data provider used for the activity
- `propertyInstances: PropertyInstanceReference[]`
  Property instances
- `personalPoints: any[]`
  Points rewarded
- `supports: SupportReference[]`
  Support references related to the activity
- `chats: any[]`
  Chats related to the activity

UserReference
Interface which holds information about a GameBus user.

- `id: number`
  Id of user
- `user: {`
  - `id: number`
    Id of user

- `firstName: string`
  First name of user
- `lastName: string`
  Last name of user
- `image: string | null`
  Path to image of user if available

}

GameDescriptorReference
Interface which holds information about a GameBus game descriptor.

- `id: number`
  Id of game descriptor
- `translationKey: string`
  Translation key of game descriptor
- `image: string | null`
  Path to image if available
- `type: string`
  Game descriptor type
- `miniGames: any[] | undefined`
  Related mini-games if any
- `isAggregate: boolean | null`
  Whether this is an aggregate

DataProviderReference
Interface which holds information about a GameBus data provider.

- `id: number`
  Id of data provider
- `name: string`
  Name of data provider
- `image: string | null`
  Path to data provider image if available
- `isConnected: boolean`
  If the user is connected to the data provider

PropertyInstanceReference
Interface which holds information about a GameBus property instance.

- `id: number`
  Id of property instance
- `value: string`
  Value of property instance
- `property: PropertyInstanceProperty`
  Property of property instance

PropertyInstanceProperty
Interface which holds information about a property of a GameBus property instance.

- `id: number`
  Id of property
- `translationKey: string`
  Relevant translation key
- `baseUnit: string`
  Base unit used for the property
- `inputType: string`
  Type of input for the property
- `aggregationStrategy: string`
  Aggregation strategy used
- `propertyPermissions: PropertyPermissionsReference[]`
  Relevant property permissions

SupportReference
Interface which holds information about support in GameBus.

- `id: number`
  Id of support
- `date: number`
  Moment of support
- `supporter: UserReference`
  Who gave the support

GameBusUser
Interface which holds extended information about a GameBus user.

- `id: number`
  User id
- `email: string`
  Email of user
- `firstName: string`
  First name of user
- `lastName: string`
  Last name of user
- `image: string | null`
  Path to image of user if any
- `registrationDate: number`
  When the user registered
- `isActivated: boolean`
  Whether the user account has been activated
- `language: string`
  What language the user is using, 'en' for English, 'nl' for Dutch
- `player: {`
  - `id: number`
    Id of player

  `}`
- `notifications: Notification[]`
  The notifications the user received

Notification
Interface which holds information about a GameBus notification.

- `id: number`
  Id of notification
- `date: number`
  Moment the notification was received
- `translationKey: string`
  Translation key of notification
- `isRead: boolean`
  Whether the notification was read by the user

- `overrideImportance: null | boolean`
  Whether to override importance
- `params: NotificationParams[]`
  Notification parameters given

NotificationParams
Interface which holds information about a GameBus notification parameter.

- `paramKey: string`
  Parameter key
- `paramValue: string | null`
  Parameter value

PropertyPermissionsReference
Interface which holds information about a GameBus property permission.

- `id: number`
  Id of the permission
- `index: any`
  Index of the permission
- `lastUpdate: any`
  Last time the permission was updated
- `decisionNote: any`
  Note related to the decision
- `state: any`
  State of the permission
- `gameDescriptor: GameDescriptorReference`
  Game descriptor related to the permission
- `allowedValues: any[]`
  Values allowed through this permission

CircleGETData
Interface which holds information about a GameBus circle.

- `id: number`
  Id of the circle
- `name: string`
  Name of the circle

- `image: string | null`
  Path to image of the circle if any
- `type: string`
  Circle type
- `description: string`
  Description of the circle
- `isPrivate: boolean`
  Whether the circle is private
- `removed: boolean`
  Whether the circle has been removed
- `leadersCanLogActivities: boolean`
  Whether leaders can log activities
- `leadersCanSignUpPlayers: boolean`
  Whether leaders can sign up players
- `autoAcceptMembershipRequests: boolean`
  Whether invited players automatically accept these requests
- `displayPersonalPointsToCircleMembersInMutualChallenges: boolean`
  Whether to display personal points to circle members in mutual challenges
- `displayPersonalPointsToCircleMembers: boolean`
  Whether to display points to circle members
- `withNudging: boolean`
  Whether nudging is enabled
- `creator: UserReference`
  Creator of the circle
- `memberships: MembershipReference[]`
  Members of the circle
- `participations: any[]`
  Participations of the circle
- `showChallengeRights: any[]`
  Challenge rights shown
- `chats: any[]`
  Relevant chats

MembershipReference
Interface which holds information about a GameBus circle membership.

- `id: number`
  Id of membership

- `state: string`
  State of membership
- `player: UserReference`
  Player who is the member
- `initiatorOfMembership: UserReference`
  Who initiated the membership
- `initiatorOfLeadership: UserReference | null`
  Who made the member a leader if relevant

ChallengePOSTData
Inteface which holds information about challenge information to post to GameBus.

- `name: string`
  Name of the challenge
- `description: null | string`
  Description of the challenge if any
- `image: null | string`
  Path to image of the challenge if any
- `websiteURL: null | string`
  Website url relevant to the challenge if any
- `minCircleSize: number`
  Minimum size of the challenge
- `maxCircleSize: number`
  Maximum size of the challenge
- `availableDate: string | number`
  When the challenge if available
- `startDate: string | number`
  When the challenge gets activated
- `endDate: string | number`
  When the challenge ends
- `rewardDescription: any`
  Description of a reward
- `rewardInfo: any`
  Information about a reward
- `target: any`
  Target of the challenge
- `contenders: any`
  Contenders of the the

- `withNudging: any`
  Whether nudging is enabled
- `rules: ChallengeRulesPOSTData[]`
  Rules relevant to the challenge
- `circles: number[]`
  Ids of relevant circles

ChallengeRulesPOSTData
Interface which holds information about a GameBus challenge rule.

- `id: any`
  Id of the rule
- `name: string`
  Name of the rule
- `image: string | null`
  Path to the image set for the rule if any
- `imageRequired: boolean`
  Whether a image must be given
- `gameDescriptors: any`
  Relevant game descriptors
- `maxTimesFired: any`
  How many times a challenge may be fired
- `minDaysBetweenFire: any`
  Minimum number of times between firing the challenge
- `conditions: ChallengeConditionsPOSTData[]`
  Relevant challenge conditions
- `points: Points[]`
  Points for the challenge

ChallengeConditionsPOSTData
Interface which holds information about a GameBus challenge condition.

- `property: number`
  Relevant property
- `operator: string`
  Operator to use for the condition
- `value: string`
  Value to use for the condition

Points
Interface which holds information about GameBus points.

- `dataProviders: DataProviderReference | undefined`
  Relevant data providers if any

GameBusToken
Interface which holds information about the token that is used for authenticating GameBus requests.

- `playerId: string`
  Id of player for who the tokens can be used
- `accessToken: string`
  The access token to authenticate the player
- `refreshToken: string | undefined`
  The refresh token for refreshing the access token

OneDriveTokenModel
Interface which holds information about the credentials required to access OneDrive files.

- `homeAccountId: string`
  Id of account for which the access token can be used
- `accessToken: string`
  The access token to access OneDrive with
- `expiresOn: number`
  When the access token expires

CombinedDataParserOutput
Interface which holds information about all data types parsed by Diabetter.

- `food: FoodModel[] | null`
  Food data or null if unavailable
- `glucose: GlucoseModel[] | null`
  Glucose data or null if unavailable
- `insulin: InsulinModel[] | null`
  Insulin data or null if unavailable
- `mood: MoodModel[] | null`
  Mood data or null if unavailable

AbbottData
Type which holds information about the data from a raw Abbott csv file.

- `device: string`
- `serial_number: string`
- `device_timestamp: string`
- `record_type: string`
- `historic_glucose_mg_dl: string | undefined`
- `historic_glucose_mmol_l: string | undefined`
- `scan_glucose_mg_dl: string | undefined`
- `scan_glucose_mmol_l: string | undefined`
- `non_numeric_rapid_acting_insulin: string`
- `rapid_acting_insulin__units_: string`
- `non_numeric_food: string`
- `carbohydrates__grams_: string`
- `carbohydrates__servings_: string`
- `non_numeric_long_acting_insulin: string`
- `long_acting_insulin__units_: string`
- `long_acting_insulin_value__units_: string`
- `notes: string`
- `strip_glucose_mg_dl: string | undefined`
- `strip_glucose_mmol_l: string | undefined`
- `ketone_mmol_l: string`
- `meal_insulin__units_: string`
- `correction_insulin__units_: string`
- `user_change_insulin__units_: string`

FoodDiaryData
Type which holds information about the data from a food diary.

- `date: string`
- `time: string`
- `meal_type: string`
- `description: string`
- `carbohydrates: string`
- `glycemic_index: string`
- `base_insulin: string`
- `high_correction_insulin: string`
- `sports_correction_insulin: string`
- `total_insulin: string`

NightScoutEntryModel
Type which holds information about the data from a Nightscout entry.

- `type: string`
- `dateString: string | undefined`
- `date: number`
- `sgv: number`
- `_id: string | undefined`
- `direction: string | undefined`
- `noise: number | undefined`
- `filtered: number | undefined`
- `unfiltered: number | undefined`
- `rssi: number | undefined`
- `utcOffset: number | undefined`
- `sysTime: string | undefined`

NightScoutTreatmentModel
Type which holds information about the data from a Nightscout treatment.

- `eventType: string`
- `created`$_a t : string$`_id: string | undefined`
- `glucose: string | undefined`
- `glucoseType: string | undefined`
- `carbs: number | undefined`
- `protein: number | undefined`
- `fat: number | undefined`
- `insulin: number | undefined`
- `units: string | undefined`
- `notes: string | undefined`
- `enteredBy: string | undefined`
- `utcOffset: number | undefined`

EndpointParameters
Interface which holds information about the parameters passed to the /data endpoint.

- exerciseTypes: ExerciseGameDescriptorNames[] | undefined
  Exercise types to retrieve, undefined if endpoint is not being used to retrieve exercises or if requester wants all possible exercise types

EndpointData
Interface which holds information about the output of the /data endpoint.

- `glucose: GlucoseModel[] | undefined`
  Glucose data, undefined if not found or not requested
- `exercise: ExerciseModel[] | undefined`
  Exercise data, undefined if not found or not requested
- `insulin: InsulinModel[] | undefined`
  Insulin data, undefined if not found or not requested
- `mood: MoodModel[] | undefined`
  Mood data, undefined if not found or not requested
- `food: FoodModel[] | undefined`
  Food data, undefined if not found or not requested

LoginAttemptToken
Interface which holds information about a started login attempt.

- `loginToken: string`
  Token to use to finish the login attempt
- `expires: number`
  When the login attempt expires

DecodedJWT
Interface which holds information about what should be in a JWT.

- `playerId: string`
  Id of authenticated player
- `accessToken: string`
  Access token for GameBus
- `refreshToken: string`
  Refresh token for GameBus
- `iat: number`
  Issued at time (as Unix timestamp)
- `exp: number`
  Moment the JWT expires (as Unix timestamp)
- `iss: string`
  Issuer of the JWT

3.2.2.2 Enums
For each public enum, a list of of all their elements is given. For each element, a value may be given. If no value is given, we won't use the value and let JavaScript automatically generate a value.

MEAL_TYPE
Types of meals recognized by Diabetter.

- BREAKFAST: 'Breakfast'
- LUNCH: 'Lunch'
- DINNER: 'Dinner'
- SNACK: 'Snack'
- UNDEFINED: ”

GlucoseUnit
Units for defining the glucose level differ per region. In the USA mg/dL is more common, whereas mmol/L is more common in Europe.

- MG_DL: ’mg/dL’
- MMOL_L: ’mmol/L’
- UNDEFINED: ’undefined’

RecordType
Different record type meanings. Glucose levels (0 & 1) are in mmol/L. Insulin (4) includes both rapid-acting insulin and long-acting insulin (in units). Carbohydrates are in grams.

- HISTORIC_GLUCOSE_LEVEL: 0
- SCAN_GLUCOSE_LEVEL: 1
- STRIP_GLUCOSE_LEVEL: 2
- INSULIN: 4
- CARBOHYDRATES: 5
- NOTES: 6

InsulinType
Types of insulin recognized by Diabetter.

- RAPID: 0
- LONG: 1

RequestMethod
Simple enum for different request methods.

- GET: ’GET’
- POST: ’POST’
- PUT: ’PUT’

BMIPropertyKeys
Relevant properties to map properties of activities to the bmiModel.

- weight: ’WEIGHT’
- length: ’LENGTH’
- age: ’AGE’
- gender: ’GENDER’
- waistCircumference: ’WAIST_CIRCUMFERENCE’
- bmi: ’BODY_MASS_INDEX’

ExercisePropertyKeys
Relevant properties to map properties of activities to the exerciseModel.

- duration: ’DURATION’
- steps: ’STEPS’
- distance: ’DISTANCE’
- calories: ’KCALORIES’
- groupSize: ’GROUP_SIZE’
- penalty:’PENALTY’
- score: ’SCORE’
- maxSpeed: ’SPEED.MAX’
- avgSpeed: ’SPEED.AVG’
- maxHeartrate: ’MAX_HEART_RATE’
- avgHeartrate: ’AVG_HEART_RATE’
- minHeartrate: ’MIN_HEART_RATE’
- heartrate: ”

FoodPropertyKeys
Relevant properties to map properties of activities to the foodModel.

- carbohydrates: ’FOOD_CARBOHYDRATES_GRAMS’
- calories: ’KCAL_CARB’
- meal_type: ’FOOD_MEAL_TYPE’

- glycemic_index: 'FOOD_GLYCEMIC_INDEX'
- fat: 'FOOD_FAT_GRAMS'
- saturatedFat: 'FOOD_SATURATED_FAT_GRAMS'
- proteins: 'FOOD_PROTEINS_GRAMS'
- fibers: 'FIBERS_WEIGHT'
- salt: 'FOOD_SALT_GRAMS'
- water: 'FOOD_WATER_GRAMS'
- sugars: 'FOOD_SUGAR_GRAMS'
- description: 'DESCRIPTION'

GlucosePropertyKeys
Relevant properties to map properties of activities to the glucoseModel.

- glucoseLevel: 'eAG_MMOLL'
- glucoseLevelMgdl: 'eAG_MGDL'

InsulinPropertyKeys
Relevant properties to map properties of activities to the insulinModel.

- insulinAmount: 'INSULIN_DOSE'
- insulinType: 'INSULIN_SPEED'

InsulinDataProviderNames
Data provider names for known insulin data sources.

- GameBuS: 'GameBus'
- Daily_run: 'Daily_run'

MoodPropertyKeys
Relevant properties to map properties of activities to the moodModel.

- arousal: 'MOOD_AROUSAL'
- valence: 'MOOD_VALENCE'

MoodDataProviderNames
Data provider names for known mood data sources.

- DAILY_RUN: 'Daily_run'

MoodGameDescriptorNames
Data property names for known mood data properties.

- logMood: 'LOG_MOOD'

ExerciseGameDescriptorNames
Game descriptor for exercise types.

- WALK: 'WALK'
- RUN: 'RUN'
- BIKE: 'BIKE'
- SOCCER: 'SOCCER'
- BASKETBALL: 'BASKETBALL'
- VOLLEYBALL: 'VOLLEYBALL'
- RUGBY: 'RUGBY'
- BASEBALL: 'BASEBALL'
- HORSE_RIDING: 'HORSE_RIDING'
- ATHLETICS: 'ATHLETICS'
- SWIMMING: 'SWIMMING'
- WATER_POLO: 'WATER_POLO'
- SURFING: 'SURFING'
- GOLF: 'GOLF'
- LACROSSE: 'LACROSSE'
- TENNIS: 'TENNIS'
- SQUASH: 'SQUASH'
- BADMINTON: 'BADMINTON'
- TABLE_TENNIS: 'TABLE_TENNIS'
- SKIING: 'SKIING'

- ICE_HOCKEY: 'ICE_HOCKEY'
- FIELD_HOCKEY: 'FIELD_HOCKEY'
- ICE_SKATING: 'ICE_SKATING'
- ROLLER_SKATING: 'ROLLER_SKATING'
- FITNESS: 'FITNESS'
- YOGA: 'YOGA'
- AEROBICS: 'AEROBICS'
- MARTIAL_ARTS: 'MARTIAL_ARTS'
- DANCE: 'DANCE'
- POOL: 'POOL'
- DARTS: 'DARTS'
- AIR_HOCKEY: 'AIR_HOCKEY'
- BOWLING: 'BOWLING'
- CHESS: 'CHESS'
- GYMNASTICS: 'GYMNASTICS'
- HIKE: 'HIKE'
- MOUNTAINBIKE: 'MOUNTAINBIKE'
- WALK_DETAIL: 'WALK(DETAIL)'
- RUN_DETAIL: 'RUN(DETAIL)'
- BIKE_DETAIL: 'BIKE(DETAIL)'

DataSource
All possible data sources. The FoodSource enum has exactly the same elements.

- ABBOTT: 0
- FOOD_DIARY: 1
- EETMETER: 2
- NIGHTSCOUT: 3

OutputDataType
All possible output models.

- GLUCOSE: 0
- INSULIN: 1
- FOOD: 2
- MOOD: 3

NightScoutDatatype
All retrievable data from Nightscout.

- ENTRY: 0
- TREATMENT: 1

GlucoseSource
Current glucose sources available.

- ABBOTT: 0
- NIGHTSCOUT: 1

InsulinSource
Current insulin sources available.

- ABBOTT: 0
- FOOD_DIARY_EXCEL: 1
- NIGHTSCOUT: 2

DataType
Types of data which can be requested.

- GLUCOSE
- INSULIN
- MOOD
- FOOD
- EXERCISE

3.2.2.3 Classes

For each class, all public and protected methods, constructors and variables are given.

DBClient

Class which manages the database connection.

Methods

- `constructor(log: boolean | undefined)`
  Constructs the database clients. If `log` is set and true, database queries and statements will be logged.
- `initialize(): void`
  Initializes the database with tables. Only needs to be executed at the start of the server.
- `reset(): void`
  Removes all entries from the database.
- `cleanLoginAttempts(): boolean`
  Removes all invalidated login attempts. Returns true if attempt succeeded, else false.
- `removeFinishedLoginAttempt(playerId: string): boolean`
  Removes a login attempt (assumed to be finished) with player id set to `playerId`. Returns true if attempt was successfully made, else false.
- `registerLoginAttempt(playerId: string, loginToken: string, expireTime: Date): boolean`
  Registers a new login attempt for details given in the parameters. Returns true if attempt was registered successfully, returns false if attempt has already been started or if the attempt could not be started for any other reason.
- `registerCallback(playerId: string, accessToken: string, refreshToken: string): boolean`
  Registers a callback from GameBus. Sets details given in the parameter for the player with `playerId` as id. Returns true if details have been registered, returns false if not.
- `getLoginAttemptByLoginToken(loginToken: string): any`
  Tries to get a row from the database by login token. Returns the row if found, else undefined.
- `getLoginAttemptByPlayerId(playerId: string): any`
  Tries to get a row from the database by player id. Returns the row if found, else undefined.
- `registerFileParse(playerId: string, file_name: string, timestamp: number): boolean`
  Adds an entry to the database stating the last parse time (`time_stamp`) for a certain file (`file_name`) of a certain player (`playerId`). Returns true if attempt succeeded, else false.
- `getLastUpdate(playerId: string, file_name: string): number`
  Fetches the last update of the file of the player. Returns the timestamp of this last update or 0 if not found.
- `cleanFileParseEvents()`
  Throws out all entries of the file_parse_events table.

- `getChildTokens(supervisorEmail: string): any`
  Gets all child tokens for a supervisor user with email `supervisorEmail`. Returns the child tokens or undefined if not found.
- `logToken(email: string, token: string): boolean`
  Logs a new token for a user in the tokens table. Returns true if attempt succeeded, else false.
- `requestSupervisor(supervisorEmail: string, childEmail: string): boolean`
  Requests supervisor role for a user. Returns true if attempt succeeded, else false.
- `confirmSupervisor(supervisorEmail: string, childEmail: string): boolean`
  Confirms supervisor role for a user. Returns true if attempt succeeded, else false.
- `getRequestedSupervisors(childEmail: string): any`
  Gets a list of requested supervisors for a normal user. Returns false if not an error occurred.
- `getChildren(supervisorEmail: string): any`
  Gets a list of normal users for a supervisor. Returns false if not an error occurred.
- `getApprovedSupervisors(childEmail: string): any`
  Gets a list of supervisors which role is approved by the normal users. Returns false if not an error occurred.
- `checkRole(email: string): boolean`
  Checks if the user is a supervisor. Returns true if so, return false if not or if an error occurred.
- `retractPermission(childEmail: string, supervisorEmail: string): boolean`
  Retracts supervisor role for a user. Returns true if attempt succeeded, else returns false.
- `close(): void`
  Closes the database connection.

GameBusClient
Class which manages the GameBus connection.

Methods

- `activity()`
  Retrieves GameBus activity object
- `exercise()`
  Retrieves GameBus exercise object
- `food()`
  Retrieves GameBus food object
- `glucose()`
  Retrieves GameBus glucose object
- `insulin()`
  Retrieves GameBus insulin object
- `mood()`
  Retrieves GameBus mood object

- `circle()`
  Retrieves GameBus circle object
- `challenge()`
  Retrieves GameBus challenge object
- `bmi()`
  Retrieves GameBus bmi object
- `user()`
  Retrieves GameBus user object
- `constructor(private readonly tokenHandler: TokenHandler | undefined, private readonly verbose: boolean | undefined)`
  Creates an Axios instance, can add options if needed.
- `async put(path: string, body: FormData | undefined, headers: Headers | undefined, query: Query | undefined, authRequired: boolean | undefined, fullResponse: boolean | undefined ): Promise<any>`
  Makes a PUT request. Returns the response data.
- `async post(path: string, body: FormData | undefined, headers: Headers | undefined, query: Query | undefined, authRequired: boolean | undefined, fullResponse: boolean | undefined ): Promise<any>`
  Makes a POST request. Returns the response data.
- `async get(path: string, headers: Headers | undefined, query: Query | undefined, authRequired: boolean | undefined, fullResponse: boolean | undefined ): Promise<any>`
  Makes a GET request. Returns the response data.
- `async request(path: string, method: RequestMethod, body: FormData | unknown | undefined, headers: Headers | undefined, query: Query | undefined, authRequired: boolean | undefined, fullResponse: boolean | undefined): Promise<any>`
  Makes a HTTP request using the given input. If `authRequired` is true, authorization header will be added to the request (note that the tokenHandler must have been given in the constructor for this to happen, else an error will be thrown). Returns the response data. An error will be thrown if the request returns an error code.
- `createHeader(authRequired: boolean | undefined, extraHeaders: Headers | undefined): Headers`
  Creates the request headers based on provided headers.
- `createURL(path: string, query: Query | undefined): string`
  Creates the request URL based on provided path and queries.

TokenHandler
Token handler that supplies access tokens.

Methods

- `constructor(accessToken: string, refreshToken: string, playerId: string`
  Sets the initial tokens and id.
- `getToken(): GameBusToken`
  Gets the current token.

Activity
Class which is used to GET/POST activities to GameBus. This is a general class which can be used for all activity types.

Variables

- `dataProviderName`
  Name of the data provider in GameBus to use
- `dataProviderID`
  Id of the data provider in GameBus to use

Methods

- `constructor(gamebus: GameBusClient, authRequired: boolean)`
  Constructs the object
- `async putActivity(data: IDActivityPOSTData, activityId: number, headers: Headers | undefined, query: Query | undefined): Promise<unknown>`
  PUTs an activity using given data on given activity ID.
- `async postActivity(data: ActivityPOSTData, headers: Headers | undefined, query: Query | undefined): Promise<unknown>`
  Posts an activity using the given data.
- `async postActivities(data: IDActivityPOSTData[], headers: Headers | undefined, query: Query | undefined): Promise<unknown>`
  Posts all activities using the given data in a single POST.
- `async getActivityById(activityId: number, headers: Headers | undefined, query: Query | undefined): Promise<ActivityGETData>`
  Gets activity for an activity ID.
- `async getAllActivities(playerId: number, headers: Headers | undefined, query: Query | undefined): Promise<ActivityGETData[]>`
  Gets all activities for a given player.
- `async getAllActivitiesWithGd(playerId: number, gameDescriptors: string[], headers: Headers | undefined, query: Query | undefined): Promise<ActivityGETData[]>`
  Gets the activities for given game descriptor translation keys.
- `async getAllAcitivitiesBetweenDate(playerId: number, startDate: Date, endDate: Date, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<ActivityGETData[]>`
  Gets all activities on a specified date range.
- `async getAllActivitiesBetweenUnix(playerId: number, startDate: number, endDate: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<ActivityGETData[]>`
  Gets all activities on a specified date range (UNIX timestamps).

- `async getAllActivitiesBetweenUnixWithGd(playerId: number, startDate: number, endDate: number, gameDescriptors: string[], order?: QueryOrder, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<ActivityGETData[]>`
  Combination of getting certain activities between given unix dates.

- `async getActivitiesOnDate(playerId: number, date: Date, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<ActivityGETData[]>`
  Shortcut function to get all activities of given user on a specific date.

- `async getActivitiesOnUnixDate(playerId: number, date: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<ActivityGETData[]>`
  Shortcut function to get all activities of given user on a specific date (UNIX).

- `async getActivitiesOnUnixDateWithGd(playerId: number, date: number, gameDescriptors: string[], order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<ActivityGETData[]>`
  Shortcut function to get all activities of a given user on a specific date with given game descriptors.

- async deleteActivityById(activityId: number, headers: Headers | undefined, query: Query | undefined): Promise<unknown>
  Deletes an activity with activity ID.

- async deleteAllActivities(playerID: number): Promise<void>
  Deletes all activities for a player.

- async checkActivityType(activityId: number, expectedTranslationKey: string, headers: Headers | undefined, query: Query | undefined): Promise<boolean>
  Checks whether the specified activity is of the correct type (correct game descriptor).

GameBusObject
Base GameBus object used for constructor-inheritance.

Variables

- `protected readonly activity: Activity`
  Activity object for communication with GameBus

- `protected readonly authRequired: boolean`
  Whether authentication is required to access GameBus

Methods

- `constructor(activity: Activity, authRequired: boolean)`
  Sets the variables

BMI
Class that allows BMI data retrieval from and posting to GameBus. Inherits GameBusObject.

Methods

- `async getBMIActivities(playerId: number, headers: Headers | undefined, query: Query | undefined): Promise<BMIModel[]>`
  Returns all BMI activities (date descending).
- `async postSingleBMIActivity(model: BMIModel, playerId: number, headers: Headers | undefined, query: Query | undefined): Promise<unknown>`
  Posts a single model for a given player.
- `toPOSTData(model: BMIModel, playerId: number): ActivityPOSTData`
  Creates POSTData from a model and playerID.
- `private static convertBMIResponseToModel(response: ActivityGETData): BMIModel`
  Converts a single response of ActivityGETData to a BMIModel.
- `static convertResponseToBMIModels(response: ActivityGETData[] | undefined): BMIModel[]`
  Converts an entire response to BMIModels.

Exercise
Class that allows exercise data retrieval from and posting to GameBus. Inherits GameBusObject.

Methods

- `async getAllExerciseActivities(playerId: number, headers: Headers | undefined, query: Query | undefined): Promise<ExerciseModel[]>`
  Returns all exercise activities.
- `async getExerciseActivityFromGd(playerId: number, gameDescriptors: ExerciseGameDescriptorNames[], headers: Headers | undefined, query: Query | undefined): Promise<ExerciseModel[]>`
  Returns all exercises of the given exercise type (game descriptors).
- `async getExerciseActivityFromGdBetweenUnix(playerId: number, gameDescriptors: ExerciseGameDescriptorNames[], startDate: number, endDate: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<ExerciseModel[]>`
  Returns all activities of given types between given dates (as unix).
- `async getAllExerciseActivitiesBetweenUnix(playerId: number, startDate: number, endDate: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<ExerciseModel[]>`
  Returns all exercise activities between given dates.
- `async getExerciseActivityFromGdOnUnixDate(playerId: number, gameDescriptors: ExerciseGameDescriptorNames[], date: number, order?: QueryOrder, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<ExerciseModel[]>`
  Returns all activities of given types on given date (as Unix).

- `async getAllExerciseActivitiesOnUnixDate(playerId: number, date: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<ExerciseModel[]>`
  Returns all exercise activities on given date.
- `static convertResponseToExerciseModels(response: ActivityGETData[] | undefined): ExerciseModel[]`
  Converts an entire response to ExerciseModels.

Food
Class that allows food data retrieval from and posting to GameBus. Inherits GameBusObject.

Methods

- `async postSingleFoodActivity(model: FoodModel, playerID: number, headers: Headers | undefined, query: Query | undefined): Promise<unknown>`
  Posts a single model for a given player.
- `async postMultipleFoodActivities(models: FoodModel[], playerID: number, headers: Headers | undefined, query: Query | undefined): Promise<unknown>`
  Posts multiple food models for a given player.
- `toPOSTData(model: FoodModel, playerID: number): ActivityPOSTData`
  Creates a POSTData from a model and playerID.
- `toIDPOSTData(model: FoodModel, playerID: number): IDActivityPOSTData`
  Creates a POSTData from a model and playerID with ID's instead of TK's.
- `async getAllFoodActivities(playerId: number, headers: Headers | undefined, query: Query | undefined): Promise<FoodModel[]>`
  Returns all food data.
- `async getFoodActivitiesBetweenUnix(playerId: number,startDate: number, endDate: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined): Promise<FoodModel[]>`
  Returns all activities of given types between given dates (as Unix).
- `async getFoodActivitiesOnUnixDate( playerId: number, date: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined ): Promise<FoodModel[]>`

  Returns all activities of given types on given date (as Unix).
- `static convertResponseToFoodModels(response: ActivityGETData[] | undefined): FoodModel[]`
  Converts an entire response to GlucoseModels.

Glucose
Class that allows glucose data retrieval from and posting to GameBus. Inherits GameBusObject.

Methods

- `async getGlucoseActivities( playerId: number, headers: Headers | undefined, query: Query | undefined ): Promise<GlucoseModel[]>`
  Returns all glucose activities.
- `async postSingleGlucoseActivity( model: GlucoseModel, playerID: number, headers: Headers | undefined, query: Query | undefined ): Promise<unknown>`
  Posts a single model for a given player.
- `async postMultipleGlucoseActivities( models: GlucoseModel[], playerID: number, headers: Headers | undefined, query: Query | undefined ): Promise<unknown>`
  Posts multiple models for a given player.
- `toPOSTData(model: GlucoseModel, playerID: number): ActivityPOSTData`
  Creates a POSTData from a model and playerID.
- `toIDPOSTData(model: GlucoseModel, playerID: number): IDActivityPOSTData`
  Creates a POSTData from a model and playerID with IDs instead of TKs.
- `async getGlucoseActivitiesBetweenUnix( playerId: number, startDate: number, endDate: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined ): Promise<GlucoseModel[]>`
  Returns all glucose activities between given dates (as unix).
- `static convertResponseToGlucoseModels(response: ActivityGETData[] | undefined): GlucoseModel[]`
  Converts an entire response to GlucoseModels.

Insulin
Class that allows insulin data retrieval from and posting to GameBus. Inherits GameBusObject.

Methods

- `async getInsulinActivities( playerId: number, headers: Headers | undefined, query: Query | undefined ): Promise<ActivityGETData[]>`
  Returns all insulin activities.
- `async getInsulinActivitiesBetweenUnix( playerId: number, startDate: number, endDate: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined ): Promise<InsulinModel[]>`
  Returns all insulin activities between given dates (as Unix).
- `async getInsulinActivitiesOnUnixDate( playerId: number, date: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined ): Promise<ActivityGETData[]>`
  Returns all insulin activities on given date (as Unix).
- `async postSingleInsulinActivity( model: InsulinModel, playerID: number, headers: Headers | undefined, query: Query | undefined ): Promise<InsulinModel>`
  Posts a single model for a given player.

- `static convertResponseToInsulinModels(response: ActivityGETData[] | undefined): InsulinModel[]`
  Converts an entire response to InsulinModels.
- `async postMultipleInsulinActivities( models: InsulinModel[], playerID: number, headers: Headers | undefined, query: Query | undefined ): Promise<unknown>`

  Posts multiple models for a given player.
- `async putSingleInsulinActivity( model: InsulinModel, playerId: number, headers: Headers | undefined, query: Query | undefined ): Promise<InsulinModel>`
  Replaces the insulin model with a new model.
- `toPOSTData(model: InsulinModel, playerID: number): ActivityPOSTData`
  Creates a POSTData from a model and playerID.
- `toIDPOSTData(model: InsulinModel, playerID: number): IDActivityPOSTData`
  Creates a POSTData from a model and playerID with IDs instead of TKs.

Mood
Class that allows mood data retrieval from and posting to GameBus. Inherits GameBusObject.

Methods

- `static convertResponseToMoodModels(response: ActivityGETData[] | undefined): MoodModel[]`
  Converts an entire response to MoodModels.
- `async getAllMoodActivities( playerId: number, headers: Headers | undefined, query: Query | undefined ): Promise<ActivityGETData[]>`
  Returns all moods of the given mood type (game descriptors).
- `async getMoodActivitiesOnUnixDate( playerId: number, date: number, order: QueryOrder | undefined, limit: number | undefined, page: number | undefined, headers: Headers | undefined, query: Query | undefined ): Promise<ActivityGETData[]>`
  Returns all activities of given types on given date (as Unix).
- `async getMoodActivitiesBetweenUnix( playerId: number, startDate: number, endDate: number, order: QueryOrder | undefined, limit: numbe | undefinedr, page: number | undefined, headers: Headers | undefined, query: Query | undefined ): Promise<MoodModel[]>`
  Returns all activities of given types between given dates (as Unix).
- `async postSingleMoodActivity( model: MoodModel, playerID: number, headers: Headers | undefined, query: Query | undefined ): Promise<MoodModel>`
  Posts a single model for a given player.
- `async postMultipleMoodActivities( models: MoodModel[], playerID: number, headers: Headers | undefined, query: Query | undefined ): Promise<unknown>`
  Posts multiple models for a given player.
- `async putSingleMoodActivity( model: MoodModel, playerId: number, headers: Headers | undefined, query: Query | undefined ): Promise<MoodModel>`
  Replaces the mood model with a new model.

- `toPOSTData(model: MoodModel, playerID: number): ActivityPOSTData`
  Creates a POSTData from a model and playerID.
- `toIDPOSTData(model: MoodModel, playerID: number): IDActivityPOSTData`
  Creates a POSTData from a model and playerID with IDs instead of TKs.

Challenge
Class which is used for posting challenges.

Methods

- `constructor(gamebus: GameBusClient, authRequired: boolean)`
  Constructs the object
- `async postChallenge( data: ChallengePOSTData, headers: Headers | undefined, query: Query | undefined ): Promise<unknown>`
  Posts a challenge to GameBus.
- `async postCircleMembership( circleId1: number, circleId2: number, challengeId: number, headers: Headers | undefined, query: Query | undefined ): Promise<unknown>`
  Posts a challenge membership to GameBus.

Circle
Class which is used for posting and retrieving circles.

Methods

- `constructor(gamebus: GameBusClient, authRequired: boolean)`
  Constructs the object
- `async getCircleById( circleId: number, headers: Headers | undefined, query: Query | undefined ): Promise<CircleGETData>`
  Gets circle information for a circle ID.
- `async getAllCircles( playerId: number, headers: Headers | undefined, query: Query | undefined ): Promise<CircleGETData[]>`
  Gets all circles for a given player.
- `async getAllCirclesLeaderDiabetter( playerId: number, headers: Headers | undefined, query: Query | undefined ): Promise<CircleGETData[]>`
  Gets all circles where a player is a leader and the circle names has "Diabetter" in it.
- `async getPlayersForAGivenCircle( circleId: number, headers: Headers | undefined, query: Query | undefined ): Promise<number[]>`
  Gets player ids in a circle.

User
Class which is used to retrieve user information.

Methods

- `constructor(gamebus: GameBusClient, authRequired: boolean)`
  Constructs the object
- `async getCurrentUser(headers: Headers | undefined, query: Query | undefined): Promise<GameBusUser>`
  Gets the currently authenticated user.

NightScoutClient
Class which is used to retrieve data from and post data to Nightscout.

Methods

- `constructor(nightScoutHost: string, token: string | undefined)`
  Constructs the object and prepares Axios. The given `nightScoutHost` is used as endpoint.
- `async postEntry(entry: NightScoutEntryModel): Promise<void>`
  Posts a night scout entry to the nightscout instance.
- `async postTreatment(treatment: NightScoutTreatmentModel): Promise<void>`
  Posts a night scout treatment to the nightscout instance.
- `async getEntries(): Promise<NightScoutEntryModel[]>`
  Gets glucose entries via the nightscout API.
- `async getTreatments(): Promise<NightScoutTreatmentModel[]>`
  Gets treatments (containing food and/or insulin data) from the nightscout API.
- `async getGlucoseUnit(): Promise<GlucoseUnit>`
  Retrieves the measurement unit for glucose entries on the Nightscout host.
- `getNightscoutHost(): string`
  Helper function to retrieve nightscout host.
- `getMaxRetrieved(): number`
  Helper function to retrieve the maximum amount of entries or treatments.

OneDriveClient
Class which is used to retrieve files from OneDrive.

Methods

- `constructor( token: string, fileName: string, folderPath: string | undefined, tableName: string | undefined, sheetName: string | undefined, doPrint: boolean | undefined, printDeep: boolean | undefined )`
  Constructs the object. The specified `token` is used for authentication, specified file is read. If `tableName` is not specified, it defaults to 'fooddiary'. If `sheetName` is not specified, it defaults to 'sheetName'.
- `async getTableValues(): Promise<any[][]>`
  Gets the table content of the table specified in the constructor.
- `async getTableList(): Promise<any>`
  Gets all tables in the file specified in the constructor.

- `async getRangeText(topLeft: string, bottomRight: string): Promise<any>`
  Retrieves the data in the remote excel file as a text array.
- `async getRangeValues(topLeft: string, bottomRight: string): Promise<any>`
  Retrieves the data in the remote excel file as a value array.

DataParser
Abstract DataParser class that can take a CSV/Excel/XML/OneDrive file as input and pass it onto other parsers. The class defines all file parsers as protected class variables and initializes these on object construction. When the data parser is created, the parser for that specific type will also be initialized as protected class variable.

Variables

- `protected csvParser: CSVParser`
  CSV file parser
- `protected excelParser: ExcelParser`
  Excel file parser
- `protected xmlParser: XMLParser`
  XML file parser
- `protected oneDriveExcelParser: OneDriveExcelParser`
  OneDrive stored Excel parser
- `protected rawData: Record<string, string>[]`
  Raw data loaded from a file
- `protected dateFormat: DateFormat`
  Date format to use
- `protected foodParser: FoodParser | undefined`
  Food parser, initialized only if specified that data is food data.
- `protected glucoseParser: GlucoseParser | undefined`
  Glucose parser, initialized only if specified that data is glucose data.
- `protected insulinParser: InsulinParser | undefined`
  Insulin parser, initialized only if specified that data is insulin data.
- `protected moodParser: MoodParser | undefined`
  Mood parser, initialized only if specified that data is mood data.
- `protected lastUpdated: number`
  UNIX timestamp in ms that indicates when it was last parsed. By default 0.
- `protected only_parse_newest: boolean`
  Whether to parse all incoming data or only new data. By default false.
- `protected readonly dataSource: DataSource`
  Data source being parsed.

- `protected filePath: string`
  Path to read data from.
- `protected userInfo: GameBusToken`
  Credentials to use for GameBus authentication.
- `protected oneDriveToken: string | undefined`
  If file is on OneDrive: token to use for OneDrive authentication.
- `protected tableName: string | undefined`
  If Excel file: table to read.

Methods

- `protected async parse(): Promise<Record<string, string | number>[] | undefined>`
  Parses data file by looking at its extension and choosing the correct file parser.
- `setFilePath(path: string): void`
  Allows the program to define the path after the object has been created.
- `getFilePath(): string`
  Gets the path to file which is being parsed.
- `abstract process(): Promise<void>`
  Processes the file to parse.
- `protected createParser(type: OutputDataType, data: any[], dataSource: any): void`
  Creates parsers and allows easy non-duplicative class variable insertion.
- `getData( outputType: OutputDataType ): | InsulinModel[] | FoodModel[] | GlucoseModel[] | MoodModel[] | CombinedDataParserOutput | undefined`
  To be called after processing, for retrieving processed data.
- `protected async postProcessedData(): Promise<void>`
  Posts processed data to GameBus.
- `protected retrieveLastUpdate(fileName: string): void`
  Returns the last timestamp when the file was parsed or the client was called for updates.
- `protected setLastUpdate(fileName: string, timestamp: number)`
  Returns the last timestamp when the file was parsed or the client was called for updates including the file name and playerId.
- `protected getLastProcessedTimestamp(): number`
  Looks over all parsers and returns the timestamp of the newest datapoint that was parsed and processed.
- `parseOnlyNewest(only_parse_newest: boolean): void`
  Configures whether to upload all incoming data or only data after the last known update.

AbbottParser
Class that reads the Abbott files and passes the data onto the relevant parsers. Inherits DataParser.

Methods
- `constructor(abbotFile: string, userInfo: GameBusToken)`
  Constructs the object and sets class variables.

EetMeterParser
Class that reads the Eetmeter files and passes the data onto the relevant parsers. Inherits DataParser.

Methods
- `constructor(xmlFile: string, userInfo: GameBusToken)`
  Constructs the object and sets class variables.
- `getData(): FoodModel[] | undefined`
  Retrieves food data if available.

FoodDiaryParser
Default class for parsing food diaries. Inherits DataParser.

Methods
- `constructor(foodDiaryFile: string, userInfo: GameBusToken, protected oneDriveToken: string | undefined)`
  Constructs the object and sets class variables.
- `static preprocess( rawData: FoodDiaryData[], mealTimeMap: Map<string, string> | undefined ): FoodDiaryData[]`
  Automatically fills rows where the date is left out with the last date above it that has been entered (for ease of use in maintaining the file). Automatically computes total insulin if not specified, but other subtypes of insulin are. Removes empty rows if present.
- `static fillDate(entry: FoodDiaryData, lastDate: string): any[]`
  Automatically fills in missing dates with the last known date in the column.
- `static fillTime( entry: FoodDiaryData, lastTime: string, mealTimeMap: Map<string, string> | undefined ): any[]`
  If no time value is specified for the element, but a meal type is the method automatically fills in a default value.
- `static computeTotalInsulin(entry: FoodDiaryData): FoodDiaryData`
  Automatically computes the sum of insulin when not filled in and other data about insulin is present.

NightscoutParser
Class that reads the Nightscout data and passes the data onto the relevant parsers. Inherits DataParser.

Methods
- `constructor( nightScoutHost: string, userInfo: GameBusToken, token?: string, private testEntries: NightScoutEntryModel[] | undefined, private testTreatments: NightScoutTreatmentModel[] | undefined)`
  DataParser construction with DataSource set.

FileParser
Abstract class for file parsers.

Methods

- `abstract parse( filePath: string, v2: any | undefined, v3: any | undefined, v4: any | undefined, v5: any | undefined ): Promise<Record<string, string>[]> | Record<string, string>[]`
  Reads raw data of a file and returns it. Can have multiple arguments that are not predefined.

CSVParser
Generic CSV reader and parser to be used for all CSV files. Inherits FileParser.

ExcelParser
Default class for parsing .xlsx files. Inherits FileParser.

Methods

- `static getMappingTableValues(filePath: string): Map<string, string>`
  Converts an Excel table with two columns into a mapping with keys in the first column and values in the second.

OneDriveExcelParser
Default class for parsing .xlsx files from a OneDrive. Inherits FileParser.

Methods

- `static async getMappingTableValues( filePath: string, oneDriveToken: string, tableName: string ): Promise<Map<string, string»`
  Converts an Excel table with two columns into a mapping with keys in the first column and values in the second.

- `assignKeys(array2D: string[][], keys: string[]): Record<string, string>[]`
  Helper function to convert the 2D array input from the OneDrive data collecter into an array of objects.

XMLParser
Generic XML reader and parser to be used for all XML files. Inherits FileParser. This class does not have any public methods or variables, except those from FileParser.

FoodMapper
Helper class to map the different food sources to 1 foodModel.

Methods

- `public static mapFood( foodSource: FoodSource, dateFormat: DateFormat ): (entry: any) => FoodModel`
  Main function that returns the correct mapping function based on given source.

GlucoseMapper
Helper class to map the different glucose sources to 1 glucoseModel.

Methods

- `public static mapGlucose( glucoseSource: GlucoseSource, dateFormat: DateFormat, glucoseUnit: GlucoseUnit ): (entry: any) => GlucoseModel`
  Main function that returns the correct mapping function based on given source.

InsulinMapper
Helper class to map the different insulin sources to 1 insulinModel.

Methods

- `public static mapInsulin( insulinSource: InsulinSource, dateFormat: DateFormat ): (entry: any) => InsulinModel`
  Main function that returns the correct mapping function based on given source.

ModelParser
Abstract class defining methods and variables to be used by the parsers for all specific data types.

Variables

- `protected newestEntry: number`
  Newest entry that is processed.
- `protected gbClient: GameBusClient`
  Client for posting to GameBus.
- `protected userInfo: GameBusToken`
  Credentials to use for GameBus authentication.

Methods

- `constructor( userInfo: GameBusToken, only_process_newest: boolean, lastUpdated: number | undefined )`
  Constructs the object, initializes the GameBus client and sets the class variables.
- `setNewestEntry(items: any[]): void`
  Calculates the most recent entry in the items array by checking the timestamps and updates class variable this.newestEntry to it.
- `getNewestEntry(): number`
  This function retrieces the timestamp of the newest entry that is processed w.r.t. its timestamp (so not the time of processing!).
- `protected filterAfterLastUpdate(entries: any[])`
  Filters a list of items with a timestamp to only include items after the last updated time given in the constructor, i.e. only return entries that took place after last update.
- `abstract post(): void`
  Posts the data to GameBus.

FoodParser
Food parser class that opens a .csv file and processes it to foodModels. Inherits ModelParser.

Variables

- `foodData: FoodModel[] | undefined`
  Food data to be exported.

Methods

- `constructor( foodInput: FoodInput, foodSource: FoodSource, dateFormat: DateFormat, userInfo: GameBusToken, only_process_newest: boolean, lastUpdated: number | undefined )`
  Create foodparser with list of food datapoints that can stem from several sources.

GlucoseParser
Glucose parser class that loads a food source and processes it to glucoseModel. Inherits ModelParser.

Variables

- `glucoseData: GlucoseModel[] | undefined`
  Glucose data to be exported.

Methods

- `constructor( glucoseInput: GlucoseInput, glucoseSource: GlucoseSource = GlucoseSource.ABBOTT, readonly dateFormat: DateFormat, userInfo: GameBusToken, only_process_newest: boolean, lastUpdated: number | undefined, glucoseUnit: GlucoseUnit | undefined )`
  List of glucose datapoints that can stem from several sources.

InsulinParser
Insulin parser class that loads a glucose source and processes it to insulinModel. Inherits ModelParser.

Variables

- `insulinData: InsulinModel[] | undefined`
  Insulin data to be exported.

Methods

- `constructor( insulinInput: InsulinInput, insulinSource: InsulinSource, dateFormat: DateFormat, userInfo: GameBusToken, only_process_newest: boolean, lastUpdated: number | undefined )`
  List of insulin datapoints that can stem from several sources.

MoodParser
Mood parser class that allows GameBus posting. Inherits ModelParser.

Variables

- `mood: MoodModel | undefined`
  Mood data to be exported.

Methods

- `constructor(moodInput: MoodModel[], userInfo: GameBusToken)`
  Create mood parser that makes sure mood data to reach Gamebus.

DataEndpoint
Class which can be used to transform requested data such it can directly be returned by the endpoint.

Methods

- `constructor( gbClient: GameBusClient, playerId: number, dataTypes: string[], parameters: EndpointParameters )`
  Constructs the endpoint object.
- `async retrieveData(dateSlice: DateSlice): Promise<EndpointData>`
  Retrieves data from GameBus.
- `static unionData(data: EndpointData): Array<any>`
  Unions given data and returns it as an array.

3.2.3 API endpoints

3.2.3.1 Authentication

Logging in GET /login
Initiates or finishes a login attempt.

Request
No body content. At least one query parameter must be given.

Argument	Type	Occurrence	Meaning
email	string	Optional	Email of the GameBus user account to initiate a login attempt for.
loginToken	string	Optional	Token of login attempt to finish. Ignored if email is set.

Table 3: Query parameters

Response
Body encoding: JSON
Status: 200 (OK) - The request has been executed successfully.

Key	Type	Occurrence	Meaning
loginToken	string	Optional	Token to be used to finish a login attempt.
expires	number	Optional	The time as Unix timestamp in milliseconds when the login token expires.
newJwt	string	Optional	Json web token for a successfully finished login attempt.

Table 4: Body content for status 200 response

Status: 400 (Bad Request) - Query parameters are invalid.
Status: 403 (Forbidden) - Cannot execute the request. If trying to initiate a new login attempt: the login may already have been started for the given email or the email may be invalid. If trying to finish a login attempt: the connect button in GameBus may not have been pressed yet.

GameBus callback POST /gamebus/callback
Processes a callback from GameBus.

Request
No body content.

Argument	Type	Occurrence	Meaning
player_id	string	Required	Player id of player for who callback is made.
access_token	string	Required	Access token to be used to authenticate the given player to GameBus.
refresh_token	string	Required	Refresh token to refresh the access token.

Table 5: Query parameters

Response
No body content.
Status: 200 (OK) - The request has been executed successfully.
Status: 400 (Bad Request) - Query parameters are invalid.
Status: 403 (Forbidden) - Cannot execute the request. It may be the case that no login attempt for the given player id has been started or that the attempt has already expired.

3.2.3.2 Health data

Query data GET /data
Retrieves health data.

Request
No body content. The authorization header must be set to Bearer JWT, where the JWT contains the credentials used for the request.

Argument	Type	Occurrence	Meaning
startDate	string	Required	Start date of timeframe to get data for. Formatted as dd-MM-yyyy.
endDate	string	Optional	End date of timeframe to get data for (the specified day itself will not be included). Formatted as dd-MM-yyyy. If not specified, it will be the day after the startDate.
dataTypes	strings	Required	Types of data to retrieve. May be food, glucose, mood, insulin or exercise.
exerciseTypes	strings	Optional	Types of exercise to retrieve. Ignored if exercise data type is not requested. If not set and requesting exercise data, all exercise types are retrieved.
union	boolean	Optional	If set, only one single list will be returned which contains per timestamp all available data for that specific timestamp.

Table 6: Query parameters

Response
Body encoding: JSON
Status: 200 (OK) - The request has been executed successfully and the requested data is in the body.

Key	Type	Occurrence	Meaning
exercise	ExerciseModels	Optional	A list of retrieved exercise data.
glucose	GlucoseModels	Optional	A list of retrieved glucose data.
insulin	InsulinModels	Optional	A list of retrieved insulin data.
mood	MoodModels	Optional	A list of retrieved mood data.
food	FoodModels	Optional	A list of retrieved food data.

Table 7: Body content for status 200 response if union is not set

Key	Type	Occurrence	Meaning
timestamp	number	Required	The moment the model(s) were added.
exercise	ExerciseModel	Optional	If available, the exercise data for the timestamp, else null.
glucose	GlucoseModel	Optional	If available, the glucose data for the timestamp, else null.
insulin	InsulinModel	Optional	If available, the insulin data for the timestamp, else null.
mood	MoodModel	Optional	If available, the mood data for the timestamp, else null.
food	FoodModel	Optional	If available, the food data for the timestamp, else null.

Table 8: Objects in the array in the body of a status 200 response if union is set

Status: 400 (Bad Request) - Query parameters are invalid.
Status: 401 (Unauthorized) - Authorization header is not set or invalid.
Status: 503 (Service Unavailable) - Cannot execute the request.

Upload a file for parsing POST /upload
Uploads and parses a file containing health data.

Request
Body is encoded as multipart/form-data and must contain a file. The authorization header must be set to Bearer JWT, where the JWT contains the credentials used for the request.

Argument	Type	Occurrence	Meaning
format	string	Required	Must be eetmeter, abbott or fooddiary and reflect the type of file uploaded.

Table 9: Body parameters

Response
Body only contains error descriptions.
Status: 200 (OK) - The request has been executed successfully.
Status: 400 (Bad Request) - Query parameters are invalid or specified file format is invalid.
Status: 401 (Unauthorized) - Authorization header is not set or invalid.
Status: 500 (Server Error) - Failed to process uploaded file.
Status: 503 (Service Unavailable) - Failed to parse uploaded file.

Delete activities POST /activities/delete
Deletes an activity from GameBus.

Request
Body encoding: JSON
The authorization header must be set to Bearer JWT, where the JWT contains the credentials used for the request.

Key	Type	Occurrence	Meaning
activityId	number	Required	Id of activity to delete.

Table 10: Body content for request

Response
No body content.
Status: 204 (No Content) - The request has been executed successfully.
Status: 400 (Bad Request) - Request body is invalid.
Status: 401 (Unauthorized) - Authorization header is not set or invalid.
Status: 503 (Service Unavailable) - Unknown error occurred.

Store or update mood data POST /mood
Stores or updates mood data.

Request
Body encoding: JSON
The authorization header must be set to Bearer JWT, where the JWT contains the credentials used for the request.

Key	Type	Occurrence	Meaning
moodTime	number	Required	The moment mood was recorded, as Unix timestamp in milliseconds.
valence	number	Required	Given valence, number ≥ 1 and ≤ 3.
arousal	number	Required	Given arousal, number ≥ 1 and ≤ 3.
activityId	string	Optional	Id of mood entry to update, only set if updating.

Table 11: Body content for request

Response
No body content.
Status: 201 (Created) - The request has been executed successfully.
Status: 400 (Bad Request) - Request body is invalid.
Status: 401 (Unauthorized) - Authorization header is not set or invalid.
Status: 503 (Service Unavailable) - Unknown error occurred.

Store or update single insulin measurement POST /insulin
Stores or update one single insulin measurement.

Request
Body encoding: JSON
The authorization header must be set to Bearer JWT, where the JWT contains the credentials used for the request.

Key	Type	Occurrence	Meaning
insulinTime	number	Required	The moment the insulin was injected, as Unix timestamp in milliseconds.
insulinAmount	number	Required	Amount of insulin injected, must be ≥ 0.
insulinType	InsulinType	Required	Type of injected insulin.
activityId	number	Optional	Id of insulin activity to update, only set if updating.

Table 12: Body content for request

Response
No body content.
Status: 201 (Created) - The request has been executed successfully.
Status: 400 (Bad Request) - Request body is invalid.

Status: 401 (Unauthorized) - Authorization header is not set or invalid.
Status: 503 (Service Unavailable) - Unknown error occurred.

3.2.3.3 Nightscout

Retrieve and parse Nightscout data GET /nightscout
Retrieves and parses Nightscout data. In the background, the Nightscout data is also posted to GameBus.

Request
No body content.

Argument	Type	Occurrence	Meaning
host	string	Required	Host to get Nighscout data from.

Table 13: Query parameters

Response
Body contains retrieved Nightscout data if available encoded as JSON and formatted in the same way as the /data endpoint with union set.
Status: 200 (OK) - The request has been executed successfully.
Status: 400 (Bad Request) - Query parameters are invalid or specified host cannot be used.
Status: 401 (Unauthorized) - Authorization header is not set or invalid.
Status: 503 (Service Unavailable) - Failed to parse uploaded file.

3.2.3.4 OneDrive

Logging in GET /onedrive/login
Starts a login attempt for OneDrive.

Request
No body content.

Argument	Type	Occurrence	Meaning
homeAccountId	string	Optional	Try to log in to a specific user account of which the home account id is known.

Table 14: Query parameters

Response
Body encoding: JSON
Status: 200 (OK) - The authorization has finished without any user input being required.

Key	**Type**	**Occurrence**	**Meaning**
homeAccountId	string	Required	Id of logged in account.
accessToken	string	Required	Access token for logged in account.
expiresOn	number	Required	The moment the given access token will expire as Unix timestamp in milliseconds.

Table 15: Body content for status 200 response

Status: 302 (Found) - Redirect to Microsoft - The user must login manually. A redirect to the given URL is required to proceed.
Status: 302 (Found) - Redirect to Diabetter - The authorization has finished, result will be redirected.

Key	**Type**	**Occurrence**	**Meaning**
homeAccountId	string	Required	Id of logged in account.
accessToken	string	Required	Access token for logged in account.
expiresOn	number	Required	The moment the given access token will expire as Unix timestamp in milliseconds.

Table 16: Query parameters in redirect

Status: 403 (Forbidden) - Cannot execute the request.

Redirect from OneDrive GET /onedrive/redirect
Processes a redirect from OneDrive after successfully logging in.

Request
No body content.

Argument	**Type**	**Occurrence**	**Meaning**
code	string	Required	Authorization code for successful login.

Table 17: Query parameters

Response
No body content.
Status: 302 (Found) - The authorization has finished, result will be redirected.

Key	**Type**	**Occurrence**	**Meaning**
homeAccountId	string	Required	Id of logged in account.
accessToken	string	Required	Access token for logged in account.
expiresOn	number	Required	The moment the given access token will expire as Unix timestamp in milliseconds.

Table 18: Query parameters in redirect

Status: 400 (Bad Request) - Query parameters are invalid.
Status: 403 (Forbidden) - Cannot execute the request.

Retrieve and parse a OneDrive file GET /onedrive/onedrive
Retrieves and parses a file stored on OneDrive. In the background, the data is also posted to GameBus.

Request
No body content. The authorization header must be set to Bearer JWT, where the JWT contains the credentials used for the request.

Argument	Type	Occurrence	Meaning
oneDriveToken	string	Required	Access token for OneDrive.
filePath	string	Required	Path to file on OneDrive to parse.

Table 19: Query parameters

Response
Body contains retrieved OneDrive data if available encoded as JSON and formatted in the same way as the /data endpoint with union set.
Status: 200 (OK) - The request has been executed successfully.
Status: 400 (Bad Request) - Query parameters are invalid or the specified file is invalid.
Status: 401 (Unauthorized) - Authorization header is not set or invalid.
Status: 503 (Service Unavailable) - Failed to parse the specified file.

3.2.3.5 Profile

Retrieve a profile GET /profile
Retrieves a profile from GameBus.

Request
No body content. The authorization header must be set to Bearer JWT, where the JWT contains the credentials used for the request.

Response
Body encoding: JSON
Status: 200 (OK) - The request has been executed successfully.

Key	Type	Occurrence	Meaning
timestamp	number	Required	Moment the profile data was retrieved as Unix timestamp.
activityId	number	Optional	ID of GameBus activity
weight	number	Optional	Weight in kg or null if not set.
length	number	Optional	Height in cm or null if not set.
age	number	Optional	Age in years or null if not set.
gender	string	Optional	Either m, f, o or null.
waistCircumference	number	Optional	In cm or null if not set.
bmi	number	Optional	In kg/m^2 or null if not set.
id	number	Required	User ID
email	string	Required	User email
firstName	string	Required	First name of user
lastName	string	Required	Last name of user
image	string	Optional	Path to image or null if no image.
registrationDate	number	Required	When the user registered as Unix timestamp in ms.
isActivated	boolean	Required	Whether the email has been verified.
language	string	Required	'en' for English, 'nl' for Dutch
player.id	number	Required	Player ID
notifications	Notification[]	Required	List of notifications

Table 20: Body content for status 200 response

Status: 401 (Unauthorized) - Authorization header is not set or invalid.

Update a profile POST /profile
Updates a profile on GameBus.

Request
Body encoding: JSON
The authorization header must be set to Bearer JWT, where the JWT contains the credentials used for the request.

Key	Type	Occurrence	Meaning
weight	number	Required	Weight to set in the profile, must be > 0.
length	number	Required	Length to set in the profile, must be > 0.
age	number	Required	Age to set in the profile, must be ≥ 0
gender	number	Optional	Gender to set in profile.
waistCircumference	number	Optional	Waist circumference to set in profile.
bmi	number	Optional	BMI to set in profile.

Table 21: Body content for request

Response

Body only contains error descriptions.
Status: 200 (OK) - The request has been executed successfully.
Status: 400 (Bad Request) - Request body content is invalid.
Status: 401 (Unauthorized) - Authorization header is not set or invalid.
Status: 503 (Service Unavailable) - Failed to post data.

3.2.3.6 Supervisor

Log an access token POST /supervisor/logToken
Logs an access token for a user. Required to get a supervisor.

Request
Body encoding: JSON

Key	Type	Occurrence	Meaning
email	string	Required	Email to log token for
token	string	Required	Token to log

Table 22: Body content for request

Response
Body only contains error descriptions.
Status: 200 (OK) - The request has been executed successfully.
Status: 400 (Bad Request) - Failed to execute the request.
Status: 404 (Not Found) - Request body content is invalid.

Request or confirm supervisor access POST /supervisor/request
Requests supervisor state or confirms it.

Request
Body encoding: JSON

Key	Type	Occurrence	Meaning
supervisorEmail	string	Required	Email of the supervisor's account
childEmail	string	Required	Email of the child's account
confirm	boolean	Optional	Whether a supervisor-child relation is being confirmed, which is only possible if requested before.

Table 23: Body content for request

Response

Body may contain a JSON with key success and value true for status 200. All other body content is about error descriptions.
Status: 200 (OK) - The request has been executed successfully.
Status: 400 (Bad Request) - Failed to execute the request.
Status: 404 (Not Found) - Request body content is invalid.

Get an access token GET /supervisor/getToken
Retrieves the child's access token for a confirmed supervisor-child relation.

Request
No body content.

Key	Type	Occurrence	Meaning
supervisorEmail	string	Required	Email of the supervisor's account
childEmail	string	Required	Email of the child's account

Table 24: Query parameter for request

Response
Body encoding: JSON
Status: 200 (OK) - The request has been executed successfully.

Key	Type	Occurrence	Meaning
token	string	Optional	Child's access token. Undefined if supervisor-child relation is invalid.

Table 25: Body content for status 200 response

Status: 404 (Not Found) - Request body content is invalid.

Get all supervisors GET /supervisor/getSupervisors
Retrieves all child's supervisors who have not accepted the child's supervisor request yet.

Request
No body content.

Key	Type	Occurrence	Meaning
childEmail	string	Required	Email of the child's account

Table 26: Query parameter for request

Response
Body encoding: JSON
Status: 200 (OK) - The request has been executed successfully.

Key	Type	Occurrence	Meaning
supervisors	string[]	Optional	List of the supervisors' emails, undefined if none

Table 27: Body content for status 200 response

Status: 404 (Not Found) - Request body content is invalid.

Get supervisors who approved a request GET /supervisor/getApproved
Retrieves all child's supervisors who approved the child's supervisor request.

Request
No body content.

Key	Type	Occurrence	Meaning
childEmail	string	Required	Email of the child's account

Table 28: Query parameter for request

Response
Body encoding: JSON
Status: 200 (OK) - The request has been executed successfully.

Key	Type	Occurrence	Meaning
supervisors	string[]	Optional	List of the supervisors' emails, undefined if none

Table 29: Body content for status 200 response

Status: 404 (Not Found) - Request body content is invalid.

Get all children GET /supervisor/getChildren
Retrieves all supervisor's children.

Request
No body content.

Key	Type	Occurrence	Meaning
supervisorEmail	string	Required	Email of the supervisor's account

Table 30: Query parameter for request

Response
Body encoding: JSON
Status: 200 (OK) - The request has been executed successfully.

Key	Type	Occurrence	Meaning
children	string[]	Optional	List of the children's emails, undefined if none

Table 31: Body content for status 200 response

Status: 404 (Not Found) - Request body content is invalid.

Retract given supervisor permission POST /supervisor/retractPermission
Retracts an earlier confirmed supervisor-child relation.

Request
Body encoding: JSON

Key	Type	Occurrence	Meaning
supervisorEmail	string	Required	Email of the supervisor's account
childEmail	string	Required	Email of the child's account

Table 32: Body content for request

Response
Body may contain a JSON with key success and value true for status 200. All other body content is about error descriptions.
Status: 200 (OK) - The request has been executed successfully.
Status: 400 (Bad Request) - Failed to execute the request.
Status: 404 (Not Found) - Request body content is invalid.

Retrieve account role GET /supervisor/role
Retrieves account role of a user account.

Request
No body content.

Key	Type	Occurrence	Meaning
mail	string	Required	Email to get role of

Table 33: Query parameter for request

Response
Body encoding: JSON
Status: 200 (OK) - The request has been executed successfully.

Key	Type	Occurrence	Meaning
supervisor	string	Optional	Role, undefined if not found

Table 34: Body content for status 200 response

Status: 404 (Not Found) - Request body content is invalid.

3.3 State dynamics

The following sequence diagrams display the state dynamics of the system. These sequence diagrams correspond to the use cases from the User Requirements Document.

3.3.1 Modify User Profile

To modify the user profile, the user navigates to the "Profile" page. Here, the user can change personal information. The changes will be stored in GameBus.

Goal:	The user modifies his profile information.
Actor:	Normal User/ Supervisor.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The profile information is modified by the actor.
Summary:	The actor user modifies the profile information.
Priority:	Should have

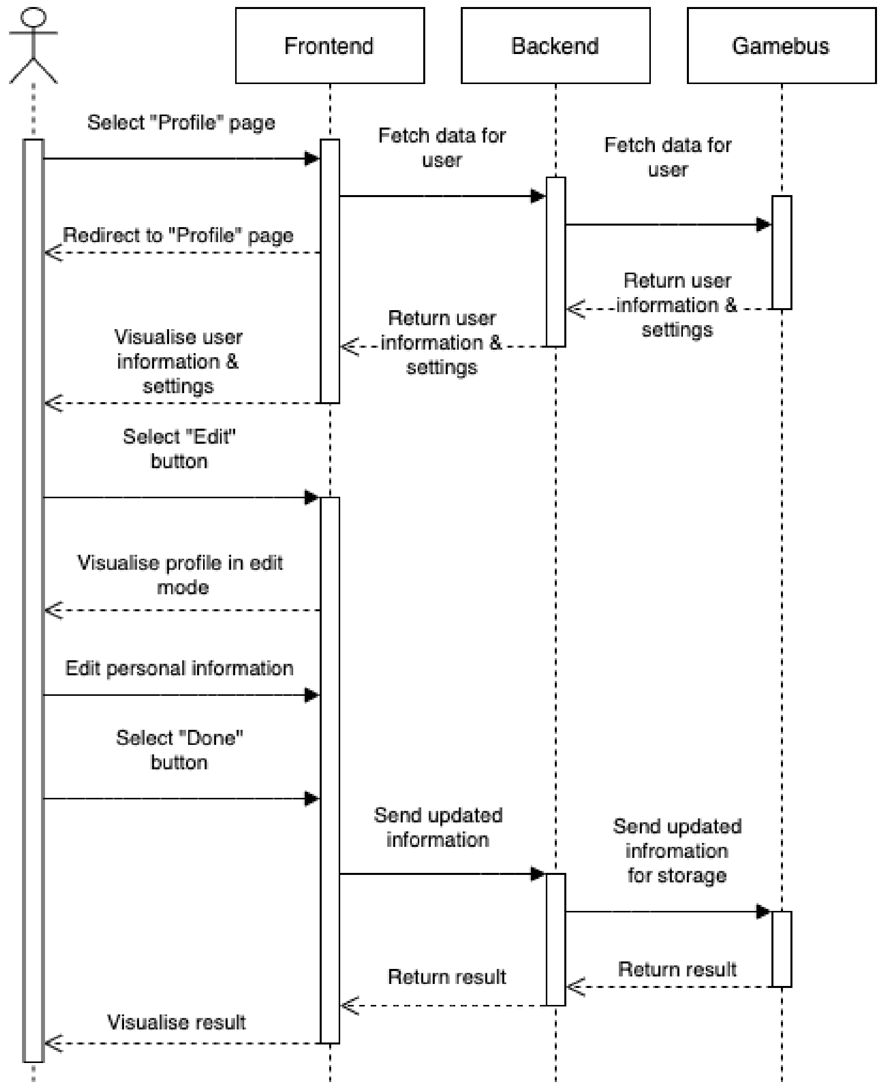

Frontend
Backend
Gamebus
Select "Profile" page
Fetch data for user
Fetch data for user
Redirect to "Profile" page
Return user information & settings
Return user information & settings
Visualise user information & settings
Select "Edit" button
Visualise profile in edit mode
Edit personal information
Select "Done" button
Send updated information
Send updated infromation for storage
Return result
Return result
Visualise result

Figure 7: Modify user profile

3.3.2 Specified Time Frame

To specify the time frame to visualize, first the user navigates to the "Main" page. Then, the user selects the filter button and sets the time frame.

Goal:	The user sets a specified time frame for a visualization
Actor:	Normal User/ Supervisor user
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	A specified time frame for a visualization is set by the actor.
Summary:	The actor sets a specified time frame for a visualization.
Priority:	Must have

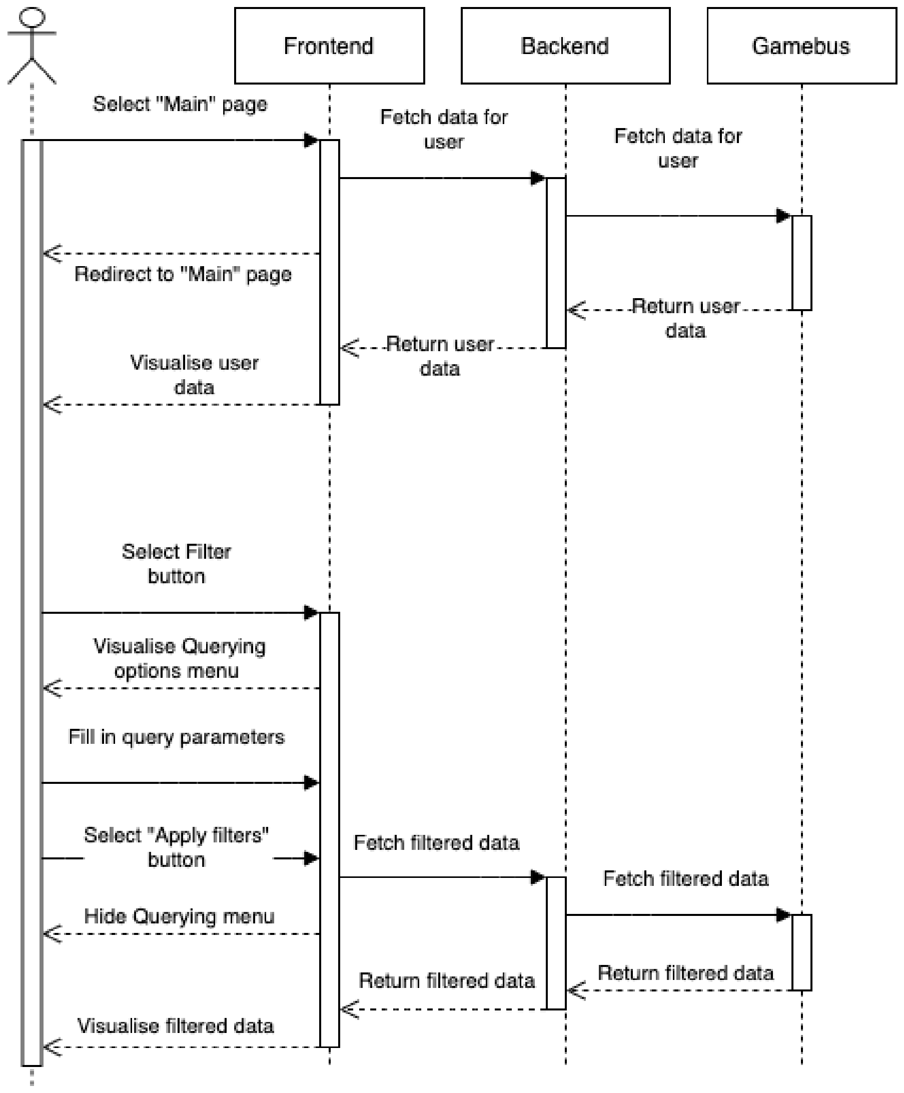


Figure 8: Make custom query

3.3.3 Upload food data

To upload a file with food data, the user navigates to the "Profile" page. On this page, the user selects the file to upload and sets the type. Then, the user presses "Upload". The file will be parsed and the content will be stored in GameBus. See figure 9 for the sequence diagram.

Goal:	The user uploads food data using the application.
Actor:	Normal User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	Food data is uploaded using the application.
Summary:	The actor uploads food data using the application.
Priority:	Must have

3.3.4 Upload glucose data

To upload a file with glucose data, the user navigates to the "Profile" page. On this page, the user selects the file to upload and sets the type. Then, the user presses "Upload". The file will be parsed and the content will be stored in GameBus. See figure 9 for the sequence diagram.

Goal:	The user uploads glucose data using the application.
Actor:	Normal User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	Glucose data is uploaded using the application.
Summary:	The actor uploads glucose data using the application.
Priority:	Must have

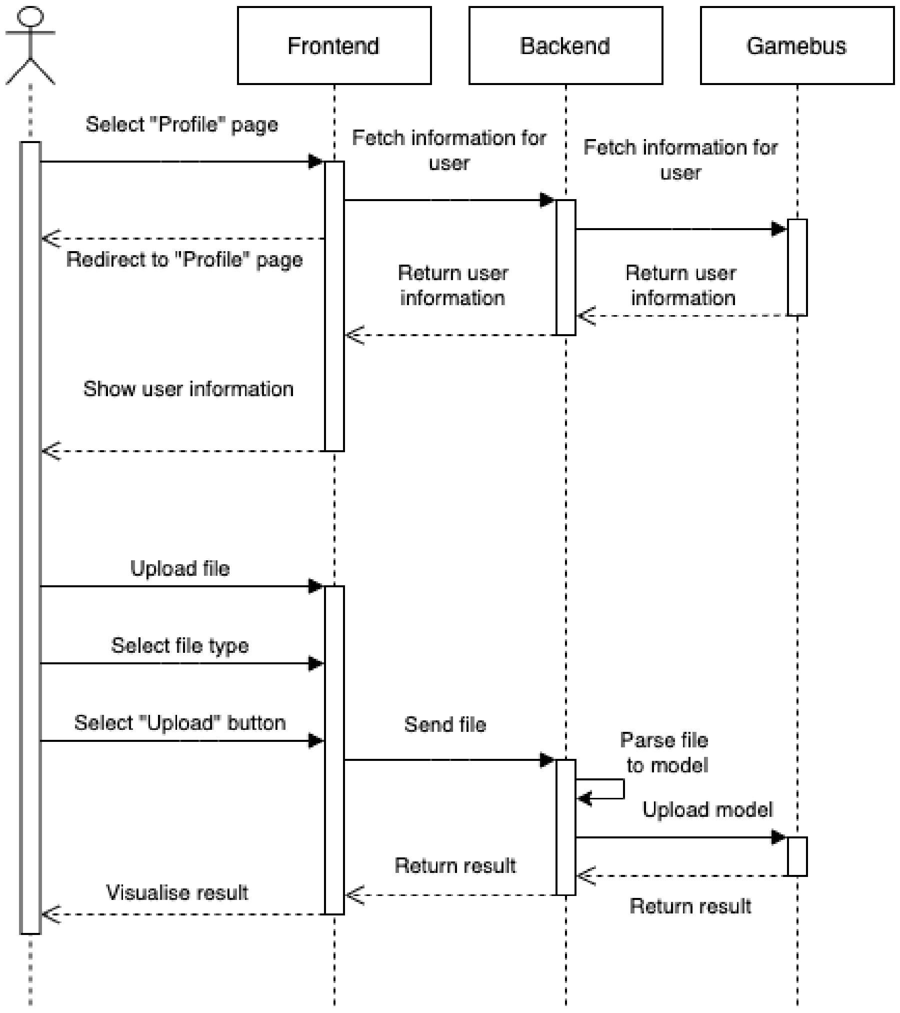


Figure 9: Upload a file with health data.

3.3.5 Modify the ranges of the glucose level types

To modify the ranges of glucose level types, the user first navigates to the "Profile" page and then makes the changes. See figure 10 for the sequence diagram on how to change glucose related settings.

Goal:	The user changes the ranges of the glucose level types in the dashboard.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The ranges of the glucose level types are changed in the dashboard.
Summary:	The actor changes the ranges of the glucose level types in the dashboard with new values.
Priority:	Must have

3.3.6 Modify the classification threshold of the hyperglycemia occurrences

To modify the hyperglycemia classification threshold, the user first navigates to the "Profile" page and can then make the change on this page. See figure 10 for the sequence diagram on how to change glucose related settings.

Goal:	The user changes the classification threshold of the hyperglycemia occurrences in the dashboard.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The classification threshold of the hyperglycemia occurrences is changed in the dashboard.
Summary:	The actor changes the classification threshold of the hyperglycemia occurrences in the dashboard with a new value.
Priority:	Should have

3.3.7 Modify the classification threshold of the hypoglycemia occurrences

To modify the hypoglycemia classification threshold, the user first navigates to the "Profile" page and can then make the change on this page. See figure 10 for the sequence diagram on how to change glucose related settings.

Goal:	The user changes the classification threshold of the hypoglycemia occurrences in the dashboard.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The classification threshold of the hypoglycemia occurrences is changed in the dashboard.
Summary:	The actor changes the classification threshold of the hypoglycemia occurrences in the dashboard with a new value.
Priority:	Should have

3.3.8 Modify the A1C estimate goal

To modify the A1C estimate goal, the user first navigates to the "Profile" page and can then make the change on this page. See figure 10 for the sequence diagram on how to change glucose related settings.

Goal:	The user changes the A1C estimate goal in the dashboard.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The A1C estimate goal value is changed in the dashboard.
Summary:	The actor changes the current value for the A1C estimate goal in the dashboard with a new value.
Priority:	Should have

3.3.9 Change the glucose level units

To modify the glucose level unit to use, the user first navigates to the "Profile" page and can then make the change on this page. See figure 10 for the sequence diagram on how to change glucose related settings.

Goal:	The user changes the glucose level units in the dashboard
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The glucose level units are changed in the dashboard.
Summary:	The actor changes the glucose level units and the new units are displayed in the dashboard.
Priority:	Could have

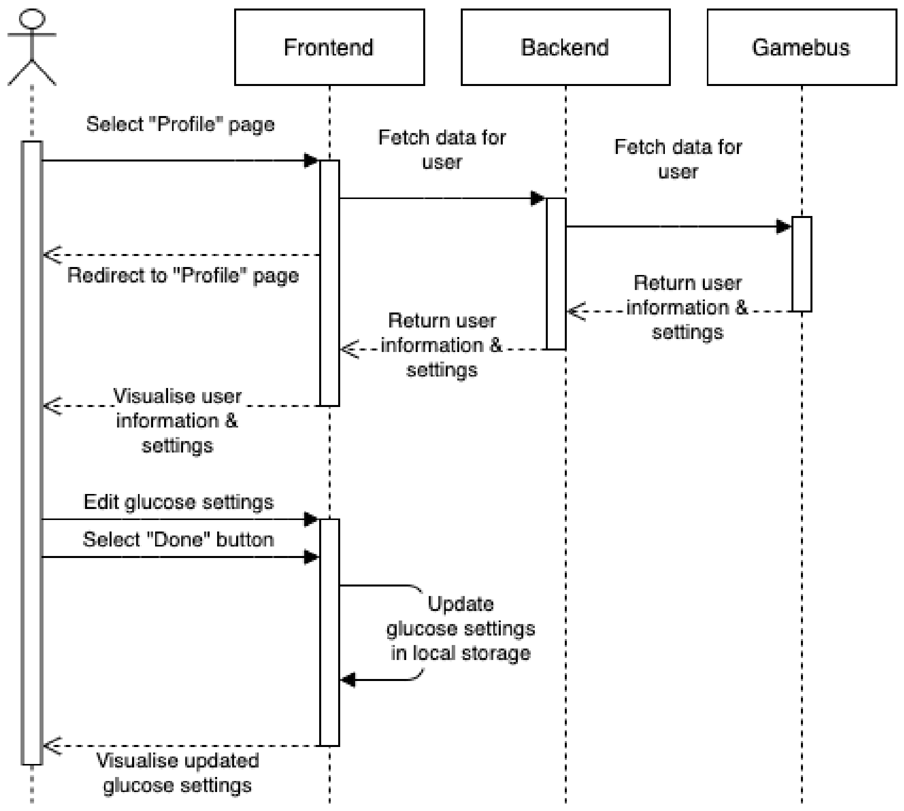

Frontend
Backend
Gamebus
Select "Profile" page
Fetch data for user
Fetch data for user
Redirect to "Profile" page
Return user information & settings
Return user information & settings
Visualise user information & settings
Edit glucose settings
Select "Done" button
Update glucose settings in local storage
Visualise updated glucose settings

Figure 10: Edit glucose settings.

3.3.10 Enter insulin intake

To enter an insulin intake, the user first navigates to the "History" page, then selects the "Insulin" tab and adds a new entry.

Goal:	The user enters insulin intake in the dashboard.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The amount, type, date, and time of the insulin intake are logged in the application.
Summary:	The actor enters the amount, type, date, and time of the insulin intake in the application, and this information is saved.
Priority:	Should have

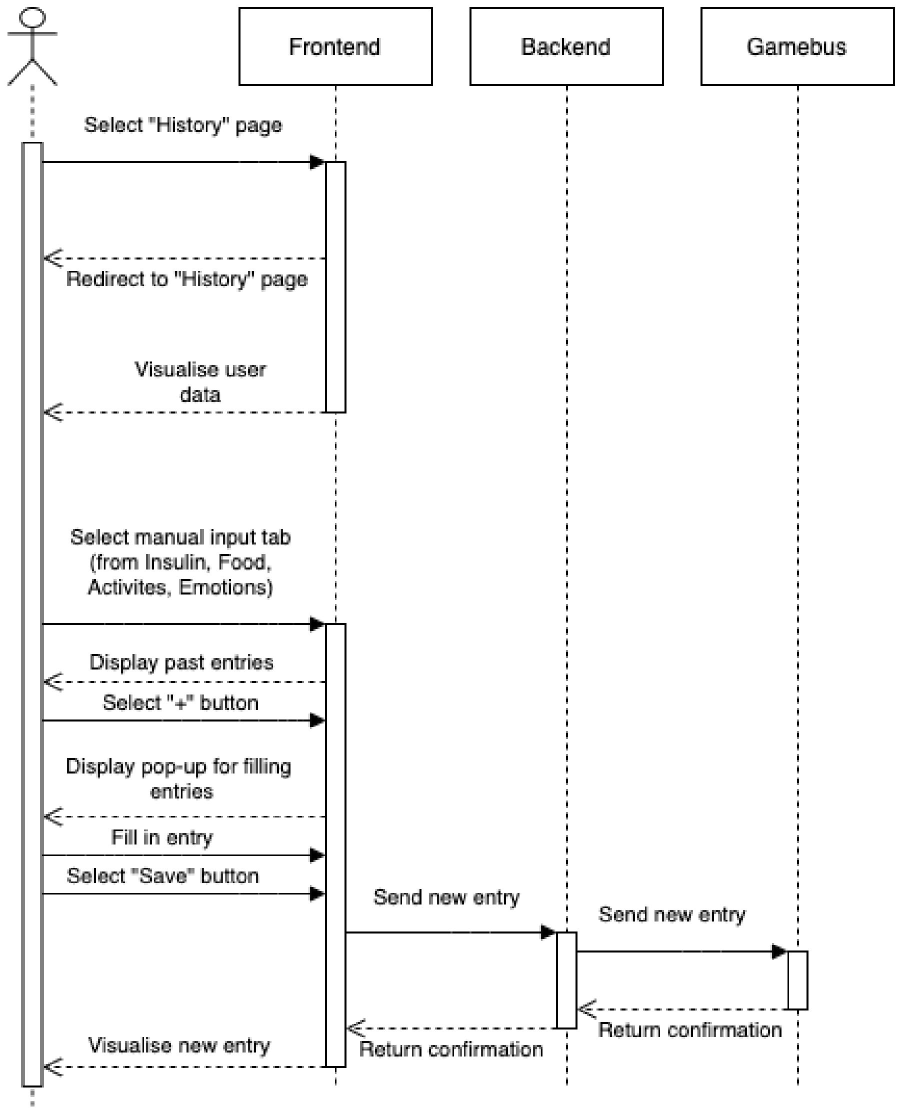


Figure 11: Make manual entry.

3.3.11 Enter emotional status

To enter an emotional status, the user first navigates to the "Main" page, looks for the emotion components, enters the happiness and excitement level and then submits the data. This will post the emotion data to GameBus.

Goal:	The user enters emotional status in the dashboard.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The actor has chosen his emotional status out of the list with 12 emotional status types.
Summary:	The actor enters his emotional status in the application and this information is saved.
Priority:	Must have

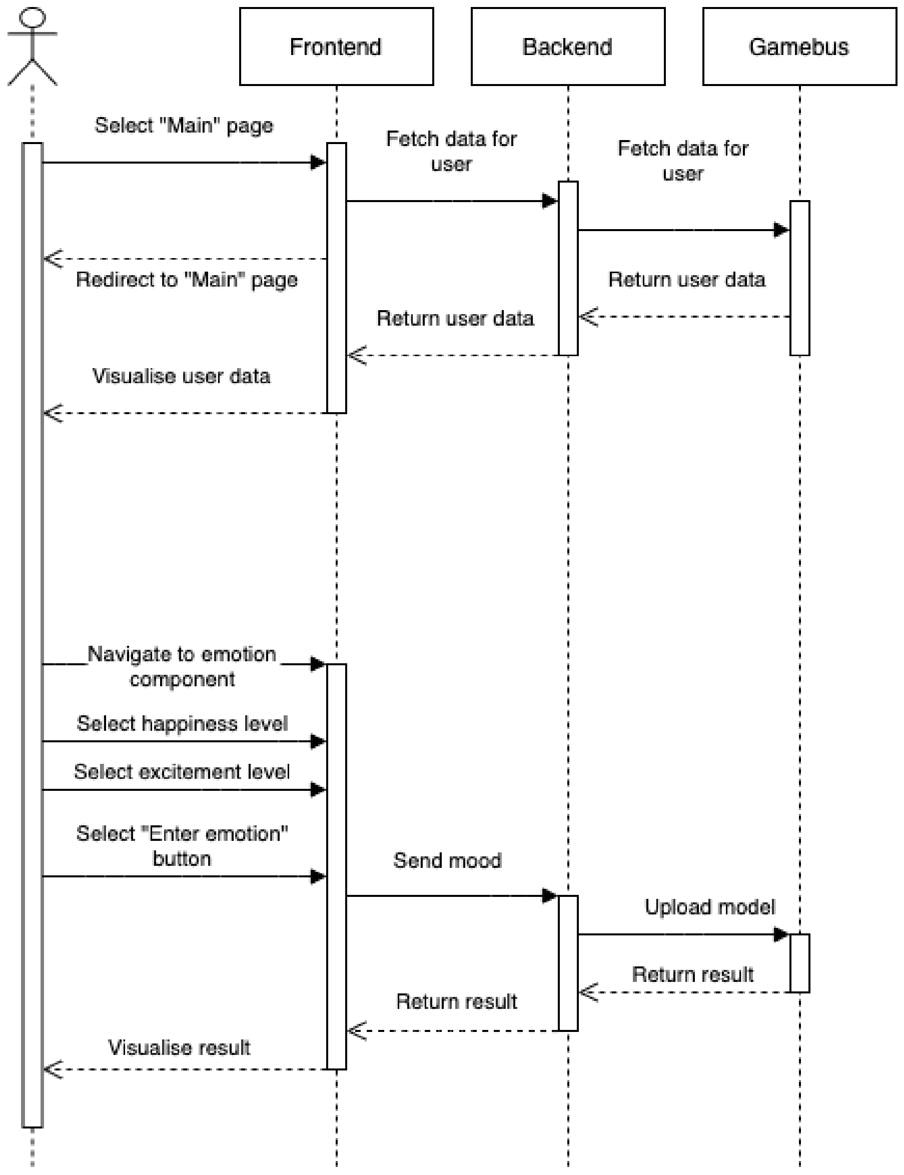


Figure 12: Make emotional entry.

3.3.12 Enable/disable emotion reminders

To switch on or off emotion reminders, the user can navigate to the "Profile" page and change this preference.

Goal:	The user enables/ disables emotion reminders.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The actor enables/ disables emotion reminders.
Summary:	The actor enables/ disables emotion reminders.
Priority:	Must have

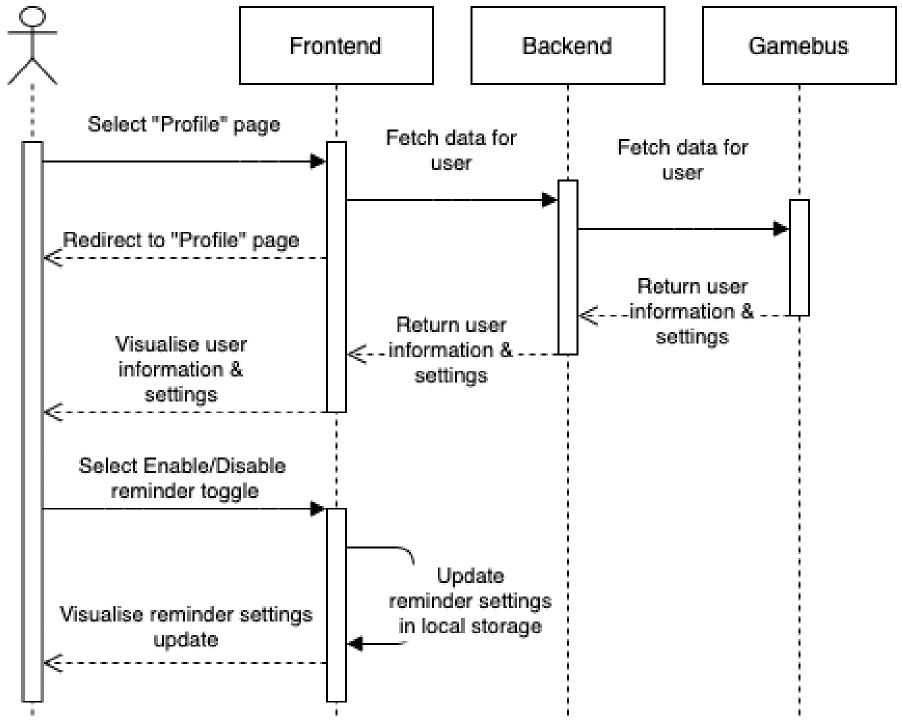

Frontend
Backend
Gamebus
Select "Profile" page
Fetch data for user
Fetch data for user
Redirect to "Profile" page
Return user information & settings
Return user information & settings
Visualise user information & settings
Select Enable/Disable reminder toggle
Update reminder settings in local storage
Visualise reminder settings update

Figure 13: Make emotional entry.

3.3.13 Edit insulin intake

To edit an insulin intakes previously entered, the user first navigates to the "History" page, selects the "Insulin" tab and then edits the entry. See figure 14 for a sequence diagrams with general information on how to edit entries.

Goal:	The user edits insulin intake in the dashboard.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The amount, type, date, and time of the insulin intake are edited in the application.
Summary:	The actor edits the amount, type, date, and time of the insulin intake in the application and this information is saved.
Priority:	Should have

3.3.14 Edit activity

To edit an activity previously entered, the user first navigates to the "History" page, selects the "Activities" tab and then edits the entry. See figure 14 for a sequence diagrams with general information on how to edit entries.

Goal:	The user edits an activity
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application. There is activity information available in the application.
Postcondition:	An activity event is edited
Summary:	The actor edits an activity
Priority:	Could have

3.3.15 Edit emotional status

To edit an emotion previously entered, the user first navigates to the "History" page, selects the "Emotions" tab and then edits the entry. See figure 14 for a sequence diagrams with general information on how to edit entries.

Goal:	The user edits emotional status input in the dashboard.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	The emotional status input is edited in the application.
Summary:	The actor edits the emotional status input in the application and this information is saved.
Priority:	Should have

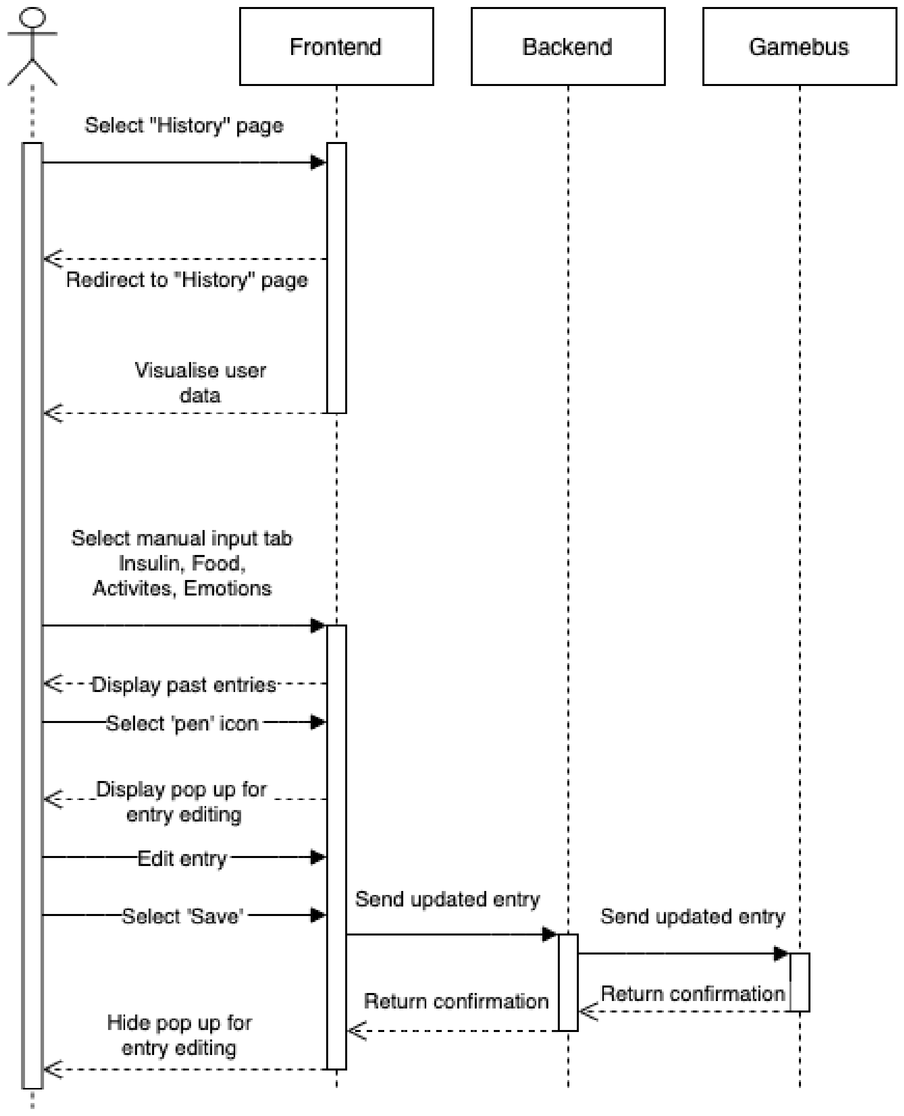


Figure 14: Edit entry.

3.3.16 Delete insulin intake

To delete an insulin intake, the user first navigates to the "History" page, selects the "Insulin" tab and then deletes the entry. See figure 15 for a sequence diagrams with general information on how to delete entries.

Goal:	The user deletes insulin intake in the dashboard.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	An insulin intake input is deleted from the application.
Summary:	The actor deletes an insulin intake from the application.
Priority:	Should have

3.3.17 Delete activity

To delete an activity, the user first navigates to the "History" page, selects the "Activities" tab and then deletes the activity. See figure 15 for a sequence diagrams with general information on how to delete entries.

Goal:	The user deletes an activity
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application. There is activity information available in the application.
Postcondition:	An activity event is deleted from the dashboard.
Summary:	The actor deletes an activity from the dashboard.
Priority:	Could have

3.3.18 Delete emotional status

To delete an emotional status, the user first navigates to the "History" page, selects the "Emotions" tab and then deletes the entry. See figure 15 for a sequence diagrams with general information on how to delete entries.

Goal:	The user deletes emotional status in the dashboard.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application.
Postcondition:	An emotional status input is deleted from the application.
Summary:	The actor deletes an emotional status from the application.
Priority:	Should have

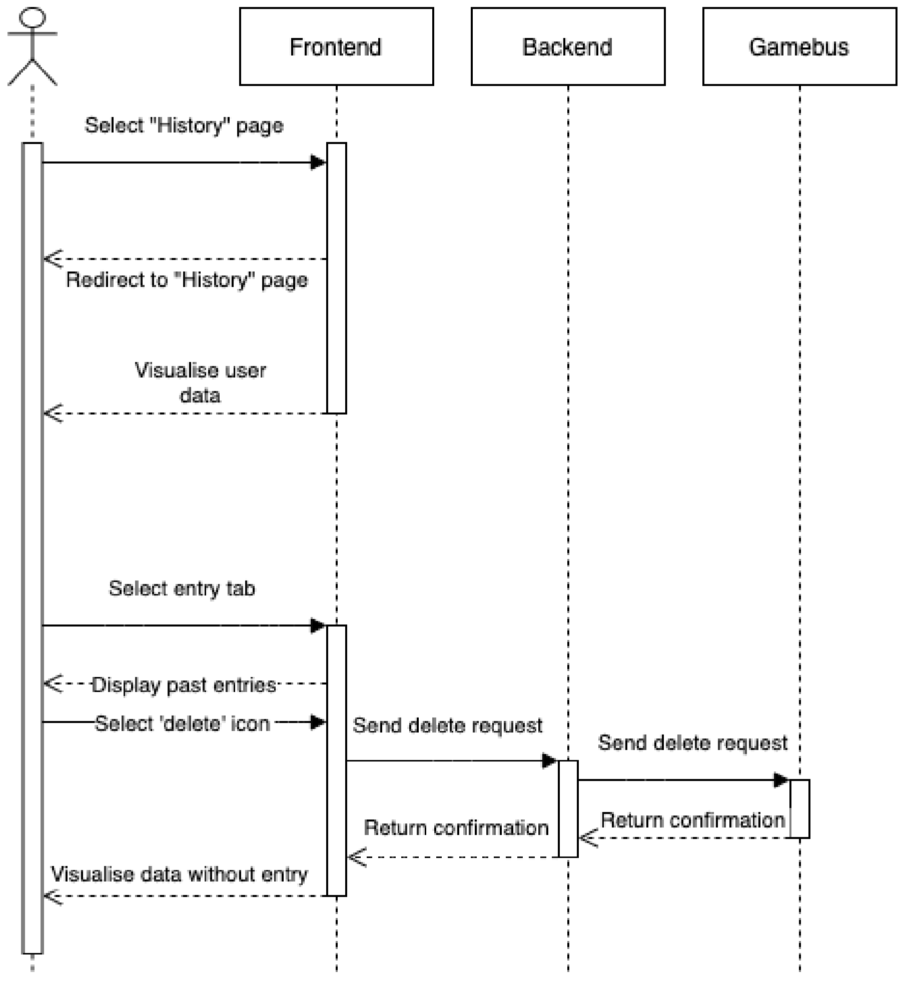


Figure 15: Delete entry.

3.3.19 Filter the activity history

To filter the activity history, the user first navigates to the "History page" and then enters the filter criteria. The table will now only display the entries meeting the entered criteria.

Goal:	The user filters the activity history on date, starting time, ending time, and type.
Actor:	User/ Supervisor user.
Precondition:	The actor has an active account and is logged in to the application. There is activity information available in the application.
Postcondition:	The activity history is filtered on the date, starting time, ending time, and type.
Summary:	The actor filters the activity history on the date, starting time, ending time, and type.
Priority:	Should have

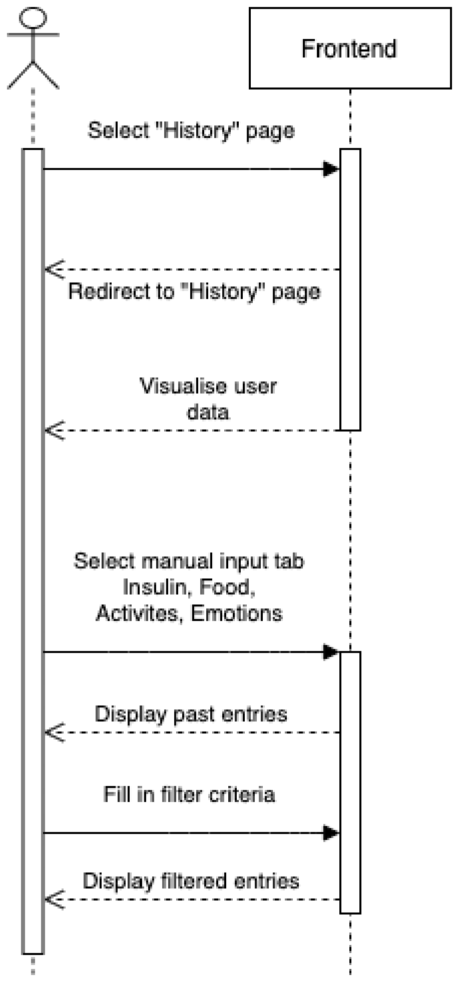


Figure 16: Filter entries.

3.3.20 Logging in

For a third-party system like Diabetter to access the GameBus-accounts of a user, a user specific access token is required. This token can be shared by the user through the data settings in the GameBus front-end. The user must press a "connect" button in there, which will make the GameBus server post an access token, refresh token and player id to a pre-defined external server.
Diabetter requires users to enter the email-address belonging to their Gamebus-account into our interface before the "connect" button in GameBus is pressed, such that the Diabetter server knows to who the tokens posted by the GameBus server belong. A so-called login token is given to the user after entering their email-address. After pressing "connect" in GameBus, this login token can be exchanged for the access and refresh token. In this way, only the user who started the login can finish it.
The access token, refresh token and player id are given to the user encoded as part of a Json Web Token (JWT). This token can be verified by the Diabetter server, meaning the end-user cannot alter it. The JWT is stored locally and will be in the header of any server request requiring authentication. If the local JWT

is lost, the user will be 'logged out'.

Goal:	Let users link their GameBus-account to Diabetter.
Actor:	All users
Precondition:	No GameBus-account is currently linked to Diabetter.
Postcondition:	The GameBus-account has been linked to Diabetter.
Summary:	Links a GameBus-account to Diabetter and saves the link in local storage.
Priority:	Must have

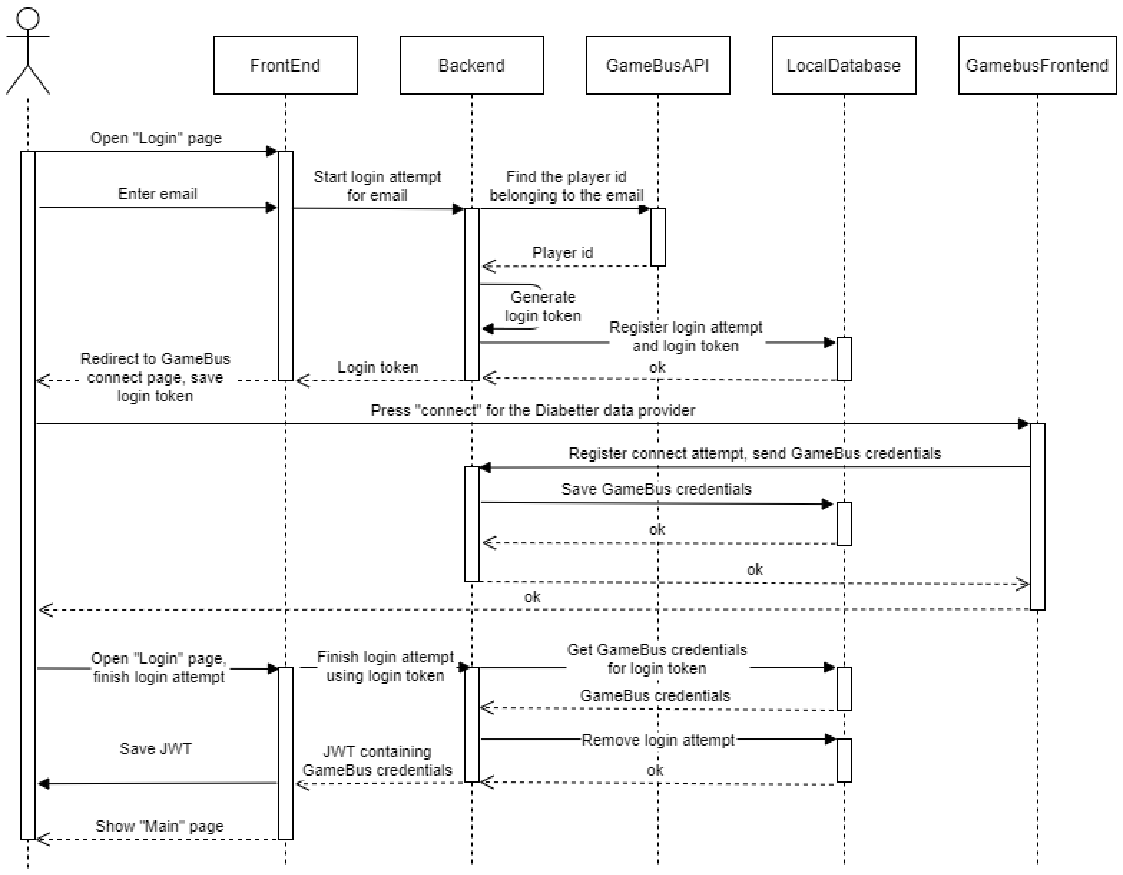

FrontEnd
Backend
GameBusAPI
LocalDatabase
GamebusFrontend
Open "Login" page
Enter email
Start login attempt for email
Find the player id belonging to the email
Player id
Generate login token
Register login attempt and login token
ok
Login token
Redirect to GameBus connect page, save login token
Press "connect" for the Diabetter data provider
Register connect attempt, send GameBus credentials
Save GameBus credentials
ok
ok
ok
Open "Login" page, finish login attempt
Finish login attempt using login token
Get GameBus credentials for login token
GameBus credentials
Remove login attempt
ok
JWT containing GameBus credentials
Save JWT
Show "Main" page

Figure 17: Log in.

3.3.21 Logging out

Logging out only requires the local storage to be emptied. This is simply achieved by having the user press logout in the user interface.

Goal:	Let users disable the link between their GameBus-account and Diabetter.
Actor:	All users
Precondition:	A GameBus-account is currently linked to Diabetter.
Postcondition:	The GameBus-account is no longer linked to Diabetter.
Summary:	Disables the link between GameBus and Diabetter by emptying the local storage.
Priority:	Must have

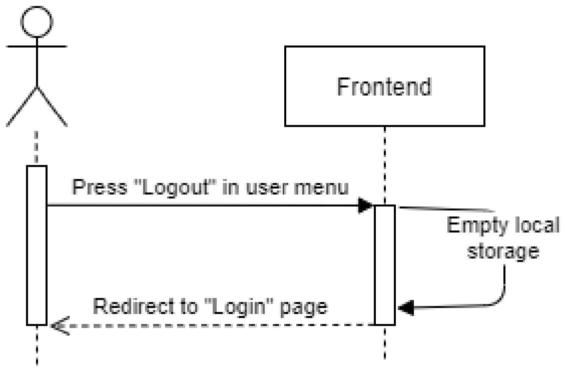


Figure 18: Log out.

3.4 Data model

3.4.1 Local database

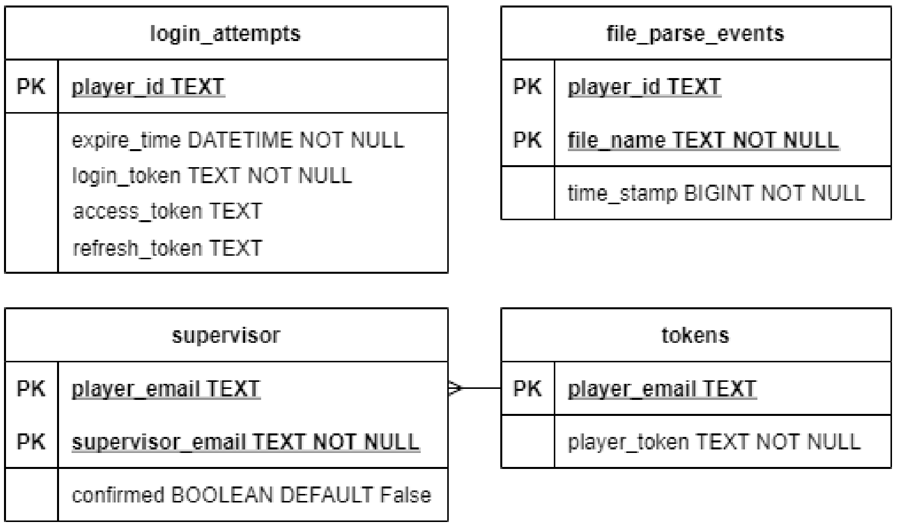

login_attempts
PK player_id TEXT
expire_time DATETIME NOT NULL
login_token TEXT NOT NULL
access_token TEXT
refresh_token TEXT
file_parse_events
PK player_id TEXT
PK file_name TEXT NOT NULL
time_stamp BIGINT NOT NULL
supervisor
PK player_email TEXT
PK supervisor_email TEXT NOT NULL
confirmed BOOLEAN DEFAULT False
tokens
PK player_email TEXT
player_token TEXT NOT NULL

Figure 19: Model of the local SQLite database

The login_attempts table contains all temporary data related to ongoing login attempts. A row is added if a new login is started. The access and refresh tokens will be filled after GameBus posts these credentials to Diabetter. When an attempt has expired, the corresponding row is removed automatically.
The file_parse_events table contains the relevant data regarding parsed files.
The supervisor and tokens tables contain data about the child-supervisor relations and the tokens to use by the supervisor to retrieve and post data for the child. A certain token may be involved in multiple child-supervisor relations.

3.5 External interface definitions

This section will give a description of the external systems that Diabetter interacts with. It will also provide the interfaces it takes advantage of to make sure the external systems function as expected.

3.5.1 GameBus

3.5.1.1 Data providers

Data providers are the entities that can post data to GameBus. On the settings screen of GameBus, a user can allow certain data providers to post to their account. Data providers can be identified by their unique name or unique ID. Because IDs can change between the test server and production server, the name is used as much as possible.

3.5.1.2 Game descriptors

Game descriptors are the types of activities (like WALK or RUN). They can have multiple property instances where data is stored (like duration or calories burned). A data provider requires permission from GameBus to post any game descriptors, so that must be requested before the data provider can post data. Game descriptors and property instances are uniquely identified by their ID and translation key. Because IDs can change between the test server and production server, the translation key is used as much as possible.

3.5.1.3 Authentication

For a request to be processed, the Authorization header must be set to Bearer + the access token provided by GameBus. If this token is invalid or has expired, GameBus will return status 401 and a new token must be retrieved.

3.5.1.4 Create and store an activity

Creating a new activity is done by making a POST request:
POST /me/activities?dryrun=false
with a JSON body in the following structure:

```
gameDescriptorTK: string,
dataProviderName: string,
image: string,
date: number,
```

propertyInstances: [
propertyTK: string;
value: string | number;
],
players: number[];

3.5.1.5 Read an already existing activity

Reading an existing activity can be done by its ID by making a GET request:
GET /activities/activityId
Reading existing activities of specific game descriptors for a specific player can be done by adding by parameters to a GET request:
GET /players/playerID/activties?gds=gamedescriptor1,gamedescriptor2
Reading existing activities between specific dates for a specific player can be done by adding by adding parameters to a GET request:
GET /players/playerID/activities?start={start_date_in_ISO_format}end={end_date_in_ISO_format}sort={- or +}date
Reading a limited number of existing activities (and aplying pagination) for a specific player can be done by adding by adding parameters to a GET request:
GET /players/{playerID}/activities?limit={max_number_activites}page={pagenumber}

These GET parameters can be combined to make more specific queries

3.5.2 OneDrive

3.5.2.1 Authentication and user authorization

Diabetter has a client secret and id, obtained from the Azure app registration page after registering the client. The Microsoft Authentication Library (MSAL) for JavaScript is used to let users authorize Diabetter to get access to their OneDrive. It automates most of the process.
The first time users try to use the OneDrive option in Diabetter, they are asked to authorize Diabetter to read files from their OneDrive. If permitted, an authorization code will be shared with the back-end. This token can then be exchanged for an access token and a user id. The access token can be used for authentication. This token has only a short lifespan, but MSAL can refresh it using the user id.

3.5.2.2 Retrieving files

- Creating a non-persistent session and getting the token for that (non-persistent because we don't make any changes to the document), can be done with the following POST request which must include the previously obtained authentication header.
  POST /v1.0/me/drive/items/{workbookID}/workbook/createSession
- Retrieving list of tables in an Excel file from Onedrive can be done with the following GET request GET /v1.0/me/drive/items/{workbookID}/workbook/worksheets({workSheetName})/tables to the server.

- Retrieving data (only cells values were used) from a table from Onedrive can be done with the following GET request
GET /v1.0/me/drive/items/{workbookID}/workbook/tables({tableName})/rows
- Retrieving data (only cells values were used) from a range from Onedrive can be done with the following GET request
GET /v1.0/me/drive/items/{workbookID}/workbook/worksheets
/range(address={workSheetName}!{topLeft}:{bottomRight}) to the server.
- Retrieving information (only file name and ID are used) for an Excel file in the root directory from Onedrive can be done with the following GET request
GET /v1.0/me/drive/root/children.
- Retrieving information (only file name and ID are used) for an Excel file in a non-root directory from Onedrive can be done with the following GET request
GET /v1.0/me/drive/root:/pathToFileExcludingFilename/children.

3.5.3 NightScout

All Nightscout instances have automatic APIs attached to them that can be accessed through <nightscout web address>/api/v1. Our own client is able to use these APIs once the web address is specified by the user. The POST requests are not used in the main flow of the program and are mostly added for testing purposes. The GET requests, however, are important for retrieving the necessary data. Three things are important to explain. For the GET requests, we had to specify a starting date in order to retrieve all glucose entries. The very old date of 1970-1-1 was chosen for this, as this is the UNIX time starting point. Furthemore, a maximum number of retrieved instances has to be specified. This number '{MAX_NIGHTSCOUT}' is set to 1,000,000 to indicate all entries have to be retrieved. The Nightscout module can be adapted to only retrieve more recent data or have a maximum on the amount of retrieved data by adjusting these numbers. The final important thing to mention is that authentication through a token is only needed for posting data and since this is not part of our main program, we never ask the user to give this token. For getting data from Nightscout, the token is not needed.

- Retrieves a list of all entries by sending the request
GET api/v1/entries/sgv.json?find[date][$gte]=0count{MAX_NIGHTSCOUT} to the server.
- Adds a new entry by sending the request
POST /api/v1/entries?token={token} to the server.
- Retrieves a list of all treatments by sending the request
GET /api/v1/treatments?find[created_at][$gte]=1970-01-01count={MAX_NIGHTSCOUT} to the server.
- Adds a new treatment by sending the request
POST /api/v1/treatments?token={token} to the server.
- Retrieves the measurement unit for glucose entries, stored in the instance status, by sending the request
GET /api/v1/status?token={token} to the server.

3.6 Design rationale

3.6.1 Separation of front-end and back-end

The web application has two very distinctive components. The first component, the front-end, makes sure data is properly visualized, processes direct user input and shows feedback to the user. The second component, the back-end, processes, stores and fetches data, further processes the user input and returns results to the front-end if available. The back-end never shows anything directly to the user.
Since both components have such different tasks, they also have different requirements regarding the development. Both use languages and frameworks optimized for their respective tasks. This already forces us to make a clear separation between front-end and back-end. Another advantage of separating the components is that the back-end could be ported to a different front-end if necessary.

3.6.2 Front-end language and framework

The following approaches were all considered for the front-end part of the project.

3.6.3 Plain HTML / CSS

One of the most simple ways of developing front-end applications is using just plain HTML. This approach has a relatively low learning curve, yet forces you to re-invent the wheel for everything you want to do, since there are no standardized, reusable components to pick up. This quickly removed this approach as a serious consideration.

3.6.4 React.js

One of the biggest frameworks being used in modern web development, React.js is a declarative JavaScript library for making fast, responsive User Interfaces. One of React's most important features is the availability of reusable components. Developers can make a React component available to other developers to use in their projects. This way, components that a lot of sites use (like a Calendar) do not have to be made over and over, but can simply be imported and used through the Node Package Manager.

3.6.5 Vue.js

Vue is another highly used framework. There are a lot of similarities between Vue and React. First and foremost, both make use of components and they are similarly fast. One of the downsides of React, however, is its steep learning curve. React uses JSX to render the HTML, which can be hard to learn for those with little experience with JavaScript. Furthermore, Vue provides more customizability and more in-depth documentation.

Because of these advantages, Vue.js was used for this project.

3.6.6 Back-end language and framework

We considered two major platforms for the back-end: ASP.NET and NodeJS.

ASP.NET
ASP.NET is created by Microsoft and was released in 2002. It uses C# as main programming language, which is a strongly-typed and object oriented language. Development is most often done in Visual Studio, offering all features required to get started. ASP.NET has built-in support for developing web APIs.

NodeJS
NodeJS is being maintained by the OpenJS Foundation and was released in 2009. It is an asynchronous event-driven JavaScript runtime, mainly used for web applications. Modules can easily be installed using the built-in module installer. Since JavaScript is very open and does not really require compiling, many different editors can be used. Visual Studio Code is one of the editors often used.

One major advantage of NodeJS is that is uses JavaScript. This language is, in comparison to C#, easier to start working with. This is important for our group, since not everyone has the same amount of programming experience. Starting with ASP.NET also requires to have or get some experience with the Visual Studio IDE, which is much more sophisticated than the lightweight Visual Studio Code editor which is often used for JavaScript development. In conclusion, this difference in learning curve was enough for us to decide to go for NodeJS instead of ASP.NET.
To have some more typing than in JavaScript, we decided to go with TypeScript instead. This reduces the amount of runtime errors caused by incorrect typing, and is at the same time completely compatible with JavaScript.

3.6.7 Back-end libraries

3.6.7.1 Express
The built-in HTTP module in NodeJS only offers very basic HTTP support and does not offer any basic web application framework interface, nor does NodeJS itself. In the first week of the project, the decision was made to start using a module which can offer this functionality and has more advanced HTTP support. Express is by far the most popular and well-known framework. It offers all features required to build a basic web application. Therefore, it had been selected as the framework to use.

3.6.7.2 Jest
In order to streamline and automate the process of adding test cases, it had been decided to start using a test framework early in the development. Jest and Mocha were considered, but Jest was selected, since it requires almost no setting up and is lightweight.

3.6.7.3 SQLite
In comparison to many major database engines like MongoDB, MySQL and SQL Server, SQLite is easy to install and easy to use and very suited for simple and low-volume database tasks. As Diabetter only

sporadically needs a few rows of data from the local database, SQLite has been selected as engine.

3.6.8 Data storage

Part of the project is that most of the data used by Diabetter is stored in GameBus. Our starting point was to follow this requirement as much as possible. For some specific types of data we had to deviate and use local storage or storage in our database instead. These exceptions are listed in table 35.

Data type	Storage location	When removed	Why not in GameBus
GameBus credentials	SQLite database	After 5 minutes	Credentials must be passed on to the user to get access to GameBus.
GameBus credentials (in a JWT)	Local storage	After 30 days	Credentials are required to access GameBus.
OneDrive access token, expiration moment and home account id	Local storage	After GameBus or OneDrive credentials have expired	Credentials cannot be used by GameBus and are issued to Diabetter only, so these credentials add no value to GameBus.
Supervisor tokens	SQLite database	Never	Supervisors need the credentials of their children to access their data.

Table 35: Types of data not stored in GameBus

3.6.9 Continuous Integration and Deployment

The team aimed to adhere to all best practices regarding the process of development. That is why GitHub and Trello were used to set up a sustainable DevOps life cycle. The group took advantage of Jest and GitHub actions for integrated automatic testing for each pull request and estimate the code test coverage. This guaranteed that only code adhering to high test standards and approved on peer review is merged into the main branch.

4 Feasibility and resource estimates

The requirements specified in this section are the minimums and are based on the results of our own experiments and on the requirements of comparable applications. Having better specifications may significantly improve the user experience.

4.1 Minimum requirements

4.1.1 Development machine minimum requirements

CPU:	Intel Core i3 (sixth generation or newer) or equivalent
Operating System:	Windows 10
Memory:	4 GB
Disk Space:	10 GB
Network:	250 KB/s
Browser:	Chrome 91
Other Software:	Node.js 14.17.1

4.1.2 Client machine requirements

CPU:	Intel Core i3 (sixth generation or newer) or equivalent
Operating System:	Windows 10
Memory:	4 GB
Disk Space:	100 MB
Network:	250 KB/s
Browser:	Chrome 91

4.1.3 Server machine requirements

CPU:	A modern x64 Intel or AMD CPU with at least 2 cores
Operating System:	Windows 10, Ubuntu Server 18.04
Memory:	2 GB
Disk Space:	10 GB
Network:	500 KB/s per user
Other Software:	Node.js 14.17.1

4.2 Performance

We have measured the performance of the system to estimate if our application can realistically be used by end-users. Chrome has an integrated development tool to measure the performance of a site, which we used to make our measurements. To get a realistic view of the performance, we picked a few day-to-day activities and took 5 measurements.

4.2.1 Test setup

For the tests, we used atp@user.nl, an account which we also used during the acceptance test. The server used during the tests is running Ubuntu Server 18.04 on a 2 core Intel Xeon E5-2690 CPU with 2 GB RAM. The client is running Windows 10 on an Intel Core i7-6700K with 16 GB RAM. Both machines have at least 50 Mbit/s of bandwidth.

4.2.2 Results

Action	Attempt	Duration (s)	Average (s)
Load main page	1	2.00	
	2	2.14	
	3	2.27	
	4	2.27	
	5	2.05	
			2.15
Filter data - June 17th	1	0.987	
	2	0.881	
	3	0.878	
	4	0.893	
	5	0.891	
			0.906
Load profile page	1	0.554	
	2	0.318	
	3	0.384	
	4	0.389	
	5	0.411	
			0.411
Upload file with 60 entries	1	2.43	
	2	2.01	
	3	2.29	
	4	2.33	
	5	2.21	
			2.25
Enter emotion	1	0.463	
	2	0.442	
	3	0.438	
	4	0.437	
	5	0.462	
			0.448
Update profile	1	0.311	
	2	0.455	
	3	0.399	
	4	0.421	
	5	0.418	
			0.400

Table 36: Action performance measurements

No very long waiting times were seen during these day-to-day activities. The performance is as expected and should be acceptable for the average user.