# Design document

# Outlr.

**A web application for analyzing subspace-outlier on high dimensional datasets**

Bennet Alexander Hörmann, Salomo Hummel,
Simeon Hendrik Schrape, Erik Bowen Wu, Udo Ian Zucker

13.01.2023

# Contents

## 1. Changes to the requirements

The functional requirement "No ground-truth file needed" (**RO18**) is now mandatory instead of optional.

# 2. Basic architecture and structure

## 2.1. 3-Tier

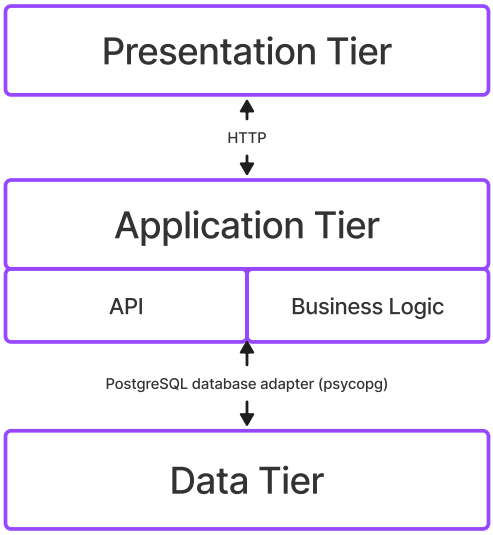


Figure 1: 3-Tier Architecture

"Outlr." is designed as a 3-Tier architecture, meaning that the application consists of following three tiers:

- The presentation tier: Allows the user to interact with the application. HTML, CSS and Javascript, along with the frameworks listed in 2.2 under the frontend section is used to provide the user interface.
- The application tier has two main tasks:
  - It provides an API allowing for communication with the presentation tier using HTTP requests. This will be implemented using the web micro framework Flask written in python. An overview of provided API endpoints is given in 4.2. Authentication is also handled by Flask and is further explained in 5.
  - It executes the business logic, which means running experiments created by users.
- The data tier: Stores all information about users, ODMs and experiments which need to be available permanently in a PostgreSQL database. As the application tier is object oriented we use SQLAlchemy as an object relational mapper. A detailed overview of what is stored in the data base is provided in 7.

  Technology used in the application and data tier is also listed in 2.2 under the backend section.

## 2.2. Technology used

As stated in the software requirements specification we decided on:

### As frontend technology:

- TypeScript $\geq$ 4.9
- Vue.js $\geq$ 3.2.45 (using Vuex[1] and Vue Router[2])
- HTML
- CSS

### As backend technology:

- Python $\geq$ 3.11.0
- PostgreSQL $\geq$ 15
- SQLAlchemy $\geq$ 1.4.46
- Pandas $\geq$ 1.5.2
- Numpy $\geq$ 1.24.1
- PyOD $\geq$ 1.0.6
- Flask $\geq$ 2.2.3

for our implementation phase.

## 2.3. Deviation from OOP

Since the frameworks we rely on for development listed in 2.2, are not developed in an object oriented manner, it is not possible to make Outlr. fully object oriented. However, to still achieve the advantages of OOP we utilize a combination of OOP and non-OOP, where the non-OOP parts are optimized for OOP characteristics such as encapsulation or modularity.

[1] see 5 for an explanation on what Vuex is and how it is used for "Outlr."
[2] see 4.1 for its usage

## 2.4. Module structure

### Frontend package structure

The following diagram depicts the package structure of the frontend:

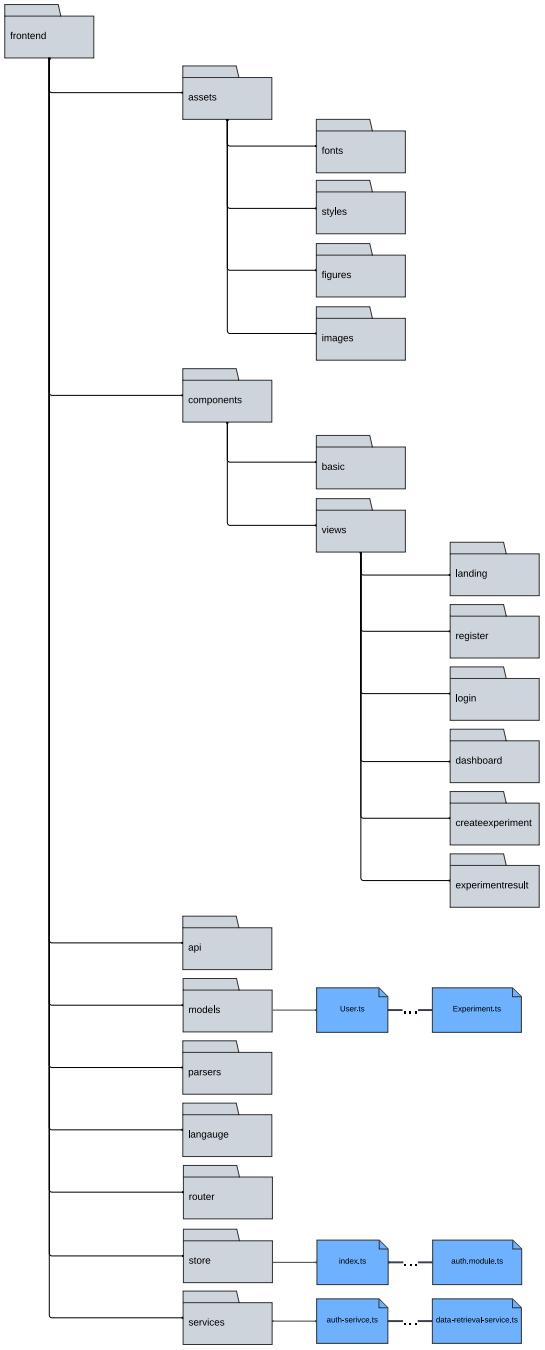


Figure 2: package structure of the frontend

The frontend consists of the usual Vue packages:

- assets: images, svgs and .css files used for the website's appearance
- components and views [3]
- router [4]
- store: contains all (Vuex) state management related files such as the authentication state module and an index.ts file initialising/exporting a store, to which the authentication module is subscribed, for the web app to use.

and is extended with

- models: here types and classes are defined, which are used in the web app. For example, the User.ts model is used for authentication related tasks and Experiment.ts e.g. is used on the dashboard page.
- services: provide a set of functionality needed in a certain part of the web app. For example, the auth-service sends the authentication related http requests using methods defined in APIRequests and handles changes to the local storage depending on the response. The data-retrieval-service.ts service includes methods easing http requests, e.g. a method, which returns the authorization header, which should be passed when an API call requires a token/authentication.

[3] detailed overview in 3.2
[4] detailed overview in 4.1

## Frontend package diagram

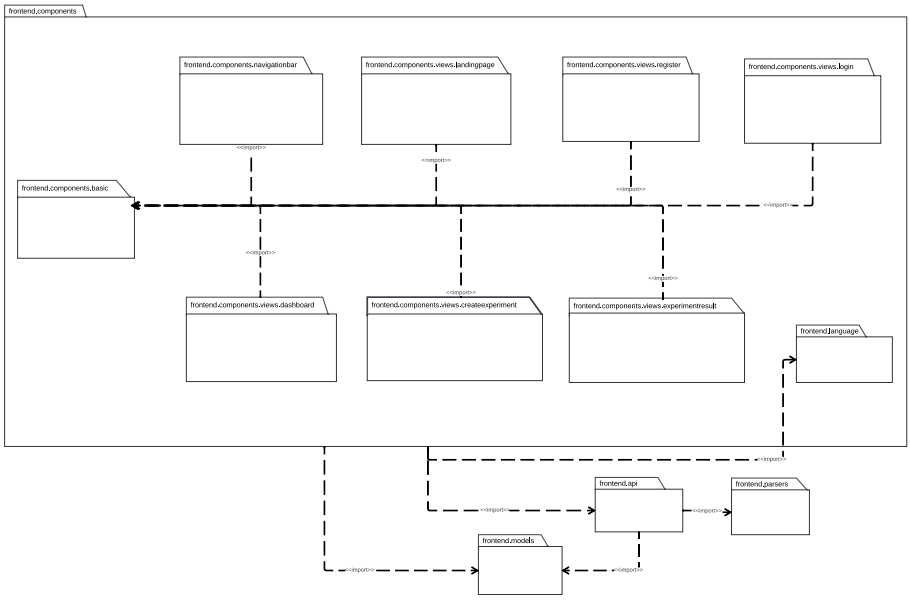


Figure 3: package diagram of the frontend

## Backend package diagram

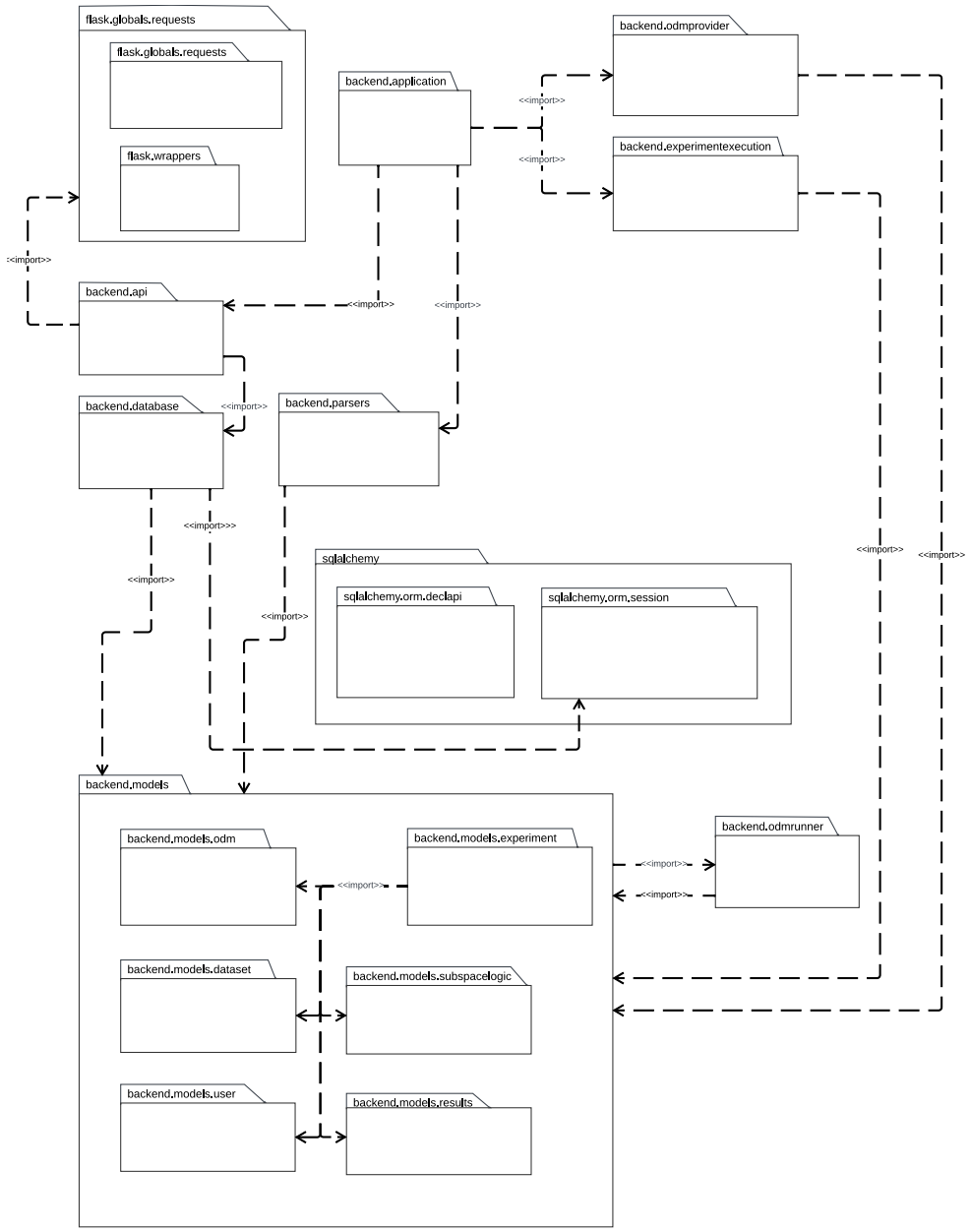


Figure 4: package diagram of the backend

# 3. Object model

## 3.1. External Packages

Outlr. utilizes a diverse array of advanced packages to provide optimal performance and functionality, while still ensuring room for scalability and own ideas.

### 3.1.1. Package sqlalchemy

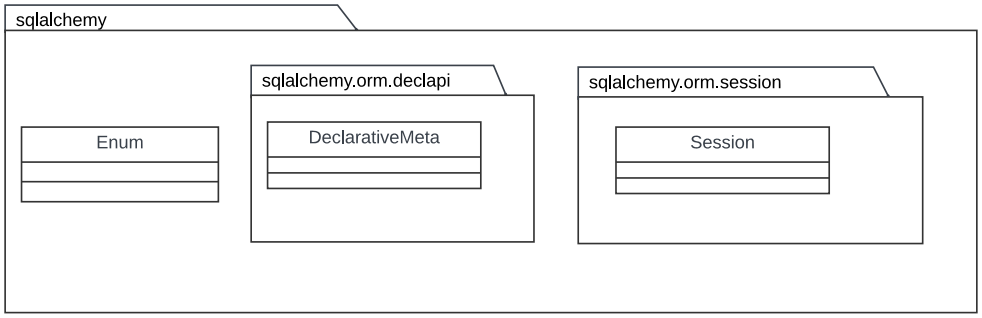


Figure 5: `sqlalchemy`

#### Class `Enum`

This class provides enum functionality, especially for databases.

### 3.1.2. Package sqlalchemy.orm.declapi

#### Class `DeclarativeMeta`

DeclarativeMeta is a metaclass provided by the SQLAlchemy library that is used to define ORM models. It allows for the declarative definition of models using class-based syntax and automates the creation of the underlying database table and columns.

### 3.1.3. Package sqlalchemy.orm.session

#### Class `Session`

This class is used to manage the connection to the database, and to keep track of the changes made to the objects in the session. The session's commit method is used to persist the changes to the database and the rollback method is used to undo changes. Sessions are also used to retrieve data from the database by querying the mapped classes or by using the ORM's query API.

### 3.1.4. Package flask

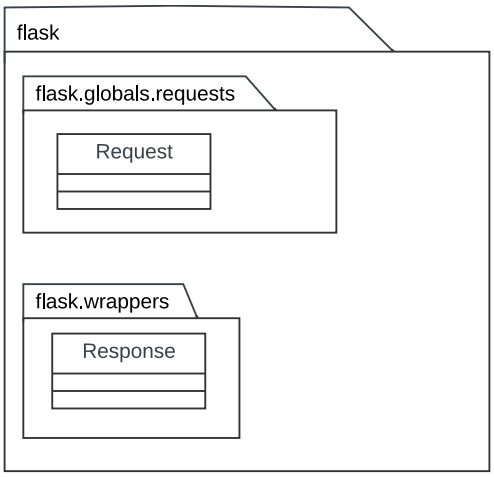


Figure 6: Flask

### 3.1.5. Package flask.globals.requests

#### Class `Request`

This class contains all the information about the current HTTP request. It contains data from the client, such as form data, headers, and query parameters. The request object is unique for each request, and is created by the Flask web server and provided to the view function when a request is received.

### 3.1.6. Package flask.wrappers

#### Class `Response`

This class is used to create and send an HTTP response to the client. The developer can use the response object to set the status code, headers, and body of the response. The response object can be returned directly from a view function, or it can be created and modified by the developer before being returned.

## 3.2. Front-end

The front-end is developed using the Vue.js framework with TypeScript.

Vue.js is a JavaScript framework that uses a component-based architecture to build user interfaces. The following terms are used in Vue.js to build a component:

- Data: The data object of a Vue component contains the properties that the component uses to render itself.
- Computed: Computed properties are used to perform calculations based on the data properties of a component and can be used in the template.
- Props: Props are used to pass data from a parent component to a child component.
- Components: Components are reusable Vue instances that can be used to build complex user interfaces.
- Watches: Watches are used to perform an action when a specific data property changes.
- Emits: Emits are used to trigger events in the parent component from a child component.
- Slots: Slots are used to define areas within a component where the parent component can insert content.

Vue.js is built around a reactive data model and a template-based syntax, which makes it easy to declaratively render dynamic data into the DOM. This is not ideal for object-oriented programming, where the focus is on encapsulation and abstraction. Vue.js components are built to be very simple, flexible, and easy to use, which means that they don't have the abstraction and encapsulation features of Object-oriented languages like classes and objects, but instead it is built around a more functional and reactive programming paradigm.

To translate this into readable diagrams and still trying to model the class diagram structure, Vue components have sections for each of the above terms.

Vue.js uses a concept called "views" to display multiple pages. A view in Vue.js is a component that represents a specific route or URL. When a user navigates to a specific route, the corresponding view component is rendered and displayed on the screen. Vue.js uses a library called vue-router to handle client-side routing, which maps URLs to specific view components and renders them when the user navigates to that URL. The router also allows you to define dynamic segments in the URL, which can be used to pass data to the view component, like the id of an object to be displayed. With vue.js you can also use different views for different conditions like authentication, or different views for different roles.

To communicate with the backend, Vue.js applications typically define services, such as the auth-service.ts service. These services use the axios library to make HTTP requests to the backend and receive responses. They provide an abstraction layer between the application and the backend, allowing for a separation of concerns and making it easier to test and maintain the code. Additionally, services are used to encapsulate all the logic related to a specific backend endpoint such as authentication, authorization, or data management. This makes it possible to reuse the logic in different parts of the application and easily update it if the backend endpoint changes.

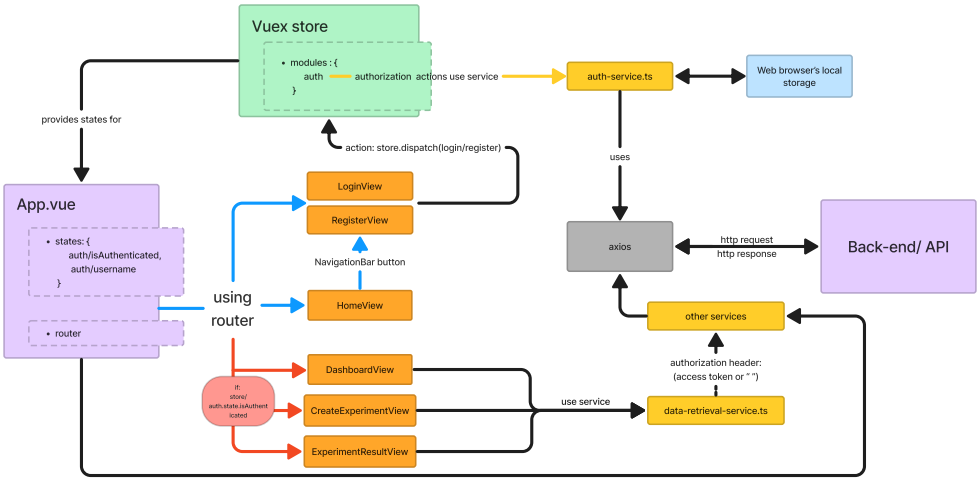


Figure 7: Basic overview of front-end (inspired by diagram from https://www.bezkoder.com/vue-3-authentication-jwt/)

### 3.2.1. Package frontend.models

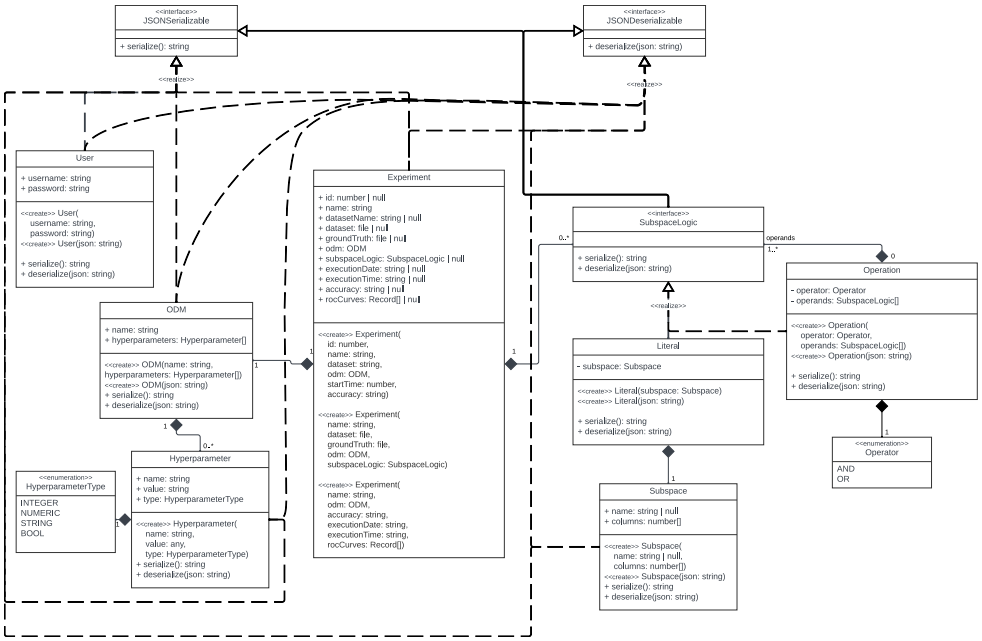


Figure 8: Overview of all frontend datamodels

### Class `Experiment`

This class models the Experiment with all of its required data. There are 3 different constructors for the use cases of the experiment data. Different sites have/need different data of the experiment. The sites then create the experiment only with the data they have/need to ensure the sites work properly.

*Attributes*

- `public id: number | null`
- `public name: string`
- `public datasetName: string | null`
- `public dataset: file | null`
- `public groundTruth: file | null`
- `public odm: ODM`
- `public subspaceLogic: SubspaceLogic | null`
- `public executionDate: string | null`
- `public executionTime: string | null`
- `public accuracy: string | null`

- `public rocCurves: Record[] | null`

*Methods*

- ```
  public Experiment(
      id: number,
      name: string,
      dataset: string,
      odm: ODM,
      startTime: number,
      accuracy: string
  ): void
  ```

  This method is the constructor for creating the experiments on the `DashboardView` site.

- ```
  public Experiment(
      name: string,
      dataset: file,
      groundTruth: file,
      odm: ODM,
      subspaceLogic: SubspaceLogic
  ): void
  ```

  This method is the constructor for creating the experiment on the `CreateExperimentView` site.

- ```
  public Experiment(
      name: string,
      odm: ODM,
      accuracy: string,
      executionDate: string,
      executionTime: string,
      rocCurves: Record[]
  ): void
  ```

  This method is the constructor for creating an experiment on the `ExperimentResultView` site.

### Class `User`

This class models the user.

`implements JSONSerializable JSONDeserializable`

*Attributes*

- `public username: string`
- `public password: string`
  ```

*Methods*

- `public User(username: string, password: string): void`

  This method is the constructor for creating a user with a username and a password.

- `public User(json: string): void`

  This method is the constructor for creating a user with a json file.

- `public serialize(): string`
- `public deserialize(json: string): void`

### Class `ODM`

This class models an outlier detection method.

`implements JSONSerializable JSONDeserializable`

*Attributes*

- `publci name: string`
- `public hyperparameters: Hyperparameter[]`

*Methods*

- `public ODM(name: string, hyperparameters: Hyperparameter[]): void`
- `public ODM(json: string): void`
- `public serialize(): string`
- `public deserialize(json: string): void`

### Class `Hyperparameter`

This class models an Hyperparameter.

`implements JSONSerializable JSONDeserializable`

*Attributes*

- `public name: string`
- `public value: string`
- `public type: HyperparameterType`

*Methods*

- `public Hyperparameter(`
  `name: string,`
  `value: any,`
  `type: HyperparameterType`
  `): void`
- `public serialize(): string`
- `public deserialize(json: string): void`

### Enum `HyperparameterType`

This enum describes all possible hyperparameter types.

- `INTEGER`
- `NUMERIC`
- `STRING`
- `BOOL`

### Interface `JSONSerializable`

This interface provides the functionality of serializing an object to JSON.

*Methods*

- `public serialize(): string`

  Serialize object. Returns json as a string.

*Extended by* `SubspaceLogic`

*Implemented by* `User ODM Hyperparameter Operation Literal Subspace`

### Interface `JSONDeserializable`

This interface provides the functionality of deserializing an object from JSON.

*Methods*

- `public deserialize(json: string): void`

  Deserialize from JSON. This overrides the current object.

*Extended by* `SubspaceLogic`

*Implemented by* `User ODM Hyperparameter Operation Literal Subspace`

### Interface `SubspaceLogic`

This interface represents the subspace logic using a composite pattern.

`extends JSONSerializable JSONDeserializable`

*Methods*

- `public serialize(): string`
- `public deserialize(json: string): void`

*Implemented by* `Operation Literal`

### Class `Operation`

This class represents an operation in the subspace logic.

`implements JSONSerializable JSONDeserializable SubspaceLogic`

*Attributes*

- `public operator: Operator`
- `public operands: SubspaceLogic[]`

*Methods*

- ```
  public Operation(
      operator: Operator,
      operands: SubspaceLogic[]
  ): void
  ```
- `public Operation(json: string): void`
- `public serialize(): string`
- `public deserialize(json: string): void`

### Enum `Operator`

This enum contains all operators that the subspace logic syntax provides.

- `AND`

  Describes a logical and operation

- `OR`

  Describes a logical or operation
  ```

### Class `Literal`

This class represents a literal in the subspace logic, which is a leaf in the tree.

`implements JSONSerializable JSONDeserializable SubspaceLogic`

*Attributes*

- `public literal: Subspace`

*Methods*

- `public Literal(subspace: Subspace): void`
- `public Literal(jason: string): void`
- `public serialize(): string`
- `public deserialize(json: string): void`

### Class `Subspace`

This class represents a subspace.

`implements JSONSerializable JSONDeserializable`

*Attributes*

- `public name: string | null`

  Optional name of the subspace assigned by the user.

- `public columns: number[]`

  List of the column indices that make up this subspace

*Methods*

- `public Subspace(name: string | null, columns: number[]): void`

- `public Subspace(jason: string): void`
- `public serialize(): string`
- `public deserialize(json: string): void`

### 3.2.2. Package frontend.api

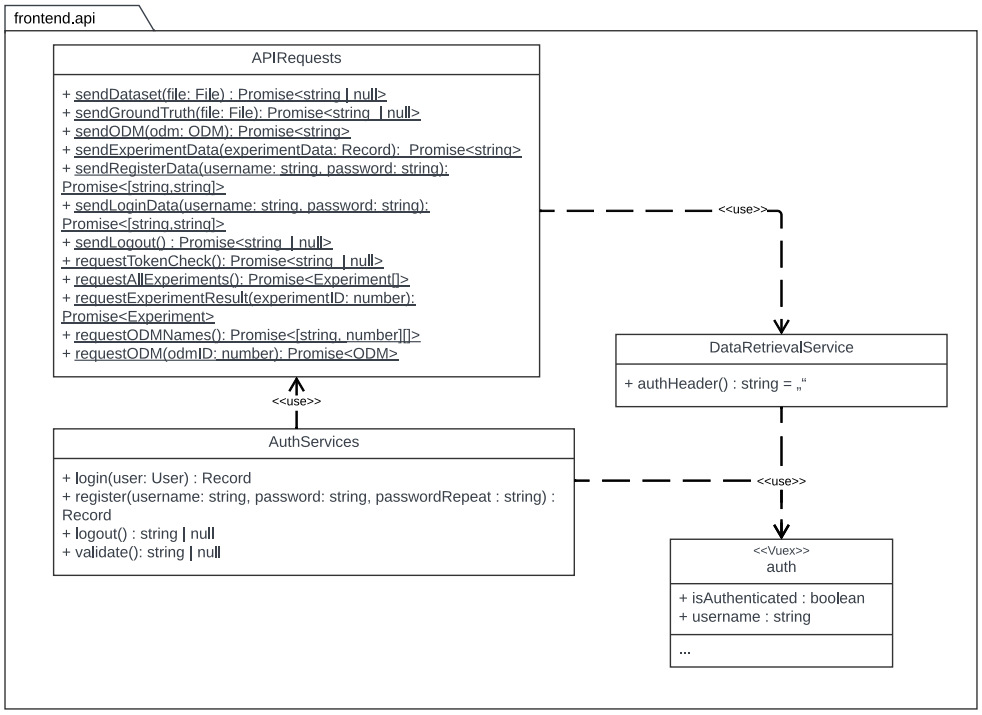


Figure 9: API requests package

#### Class `APIRequests`

Contains methods with which sending requests to the backend is handled. This class uses `DataRetrievalService`.

*Methods*

- `public static sendDataset(file: File): Promise<string | null>`

  Sends the provided file, the dataset, to the backend. It returns a promise, which contains whether sending the file was successful (= null) or not (= string, the error message).

- `public static sendDataset(file: File): Promise<string | null>`

  Sends the provided file, the dataset, to the back-end - Returns a promise, which contains whether sending the file was successful (= null) or not (= string, the error message

- `public static sendODM(odm: ODM): Promise<string | null>`

Sends the selected ODM to the back-end. Returns a promise, which contains whether sending the ODM, to be used, was successful (= null) or not (= string, the error message)

- `public static sendExperimentData(experimentData: Record): Promise<string | null>`

  Sends data necessary for creation of an experiment to the back-end - Returns a promise, which contains whether sending the data necessary for experiment creation was successful (= null) or not (= string, the error message)

- `public static sendRegisterData(username: string, password: string): Promise<[string, string]>`

  Sends data necessary for creation of an experiment to the back-end - Returns a promise, which contains the access token and username, when registering was successful

- `public static sendLoginData(username: string, password: string): Promise<[string, string]>`

  Sends data necessary for logging in to the back-end - Returns a promise, which contains the access token and username, when logging in was successful

- `public static sendLogout(): Promise<string | null>`

  Sends request to back-end to delete/invalidate the access token provided with the http request

- `public static requestTokenCheck(): Promise<string | null>`

  Sends request to back-end to validate whether the access token provided with http request still is valid - Returns null, when token still is valid and a string, the error message to be displayed, when the token has expired

- `public static requestExperimentResult(expId: number): Promise<Experiment>`

  Sends request to back-end to respond with the result of the experiment belonging to the user with id expId

- `public static requestExperimentResult(experimentId: number): Promise<Experiment>`

  Sends request to back-end to respond with (the result of) the experiment belonging to the user with id = experimentId

- `public static requestODMNames(): Promise<[string, id]>`

  Sends request to back-end to respond with all ODMs

- `public static requestODM(odmId: number): Promise<ODM>`

  Sends request to back-end to respond with the ODM with id = odmId

### Class `AuthServices`

Handles the login, register, logout and uses `APIRequests` to send the requests

*Methods*

- `public static login(user: User): Record`

  Handles the frontend login process by calling `sendLoginData` and evaluating its promise. The method returns a record containing either the access token and username or an error message.

- `public static register(username: string, password: string, passwordRepeat : string): Record`

  Handles the front-end register process by first validating the input and then by calling `sendRegisterData` and evaluating its promise. The method returns a record containing either the access token and username or an error message.

- `public static logout(): string | null`

  Handles the front-end logout process by deleting the access token and by calling `sendLogout` and evaluating its promise. The method returns either null, when successful, or an error message string.

- `public static validate(): string | null`

  Handles the front-end access token validation process by calling `requestTokenCheck` and evaluating its promise. The method returns either null, when successful, or an error message string. If the token has expired it is removed from local storage.

### Class `DataRetrievalService`

Contains methods easing sending requests and uses the states defined in the Vuex auth module. This class is used by `APIRequests`.

*Methods*

- `public static authHeader(): string`

  Returns the authentication header containing the access token, which has to be included in the http request if the user has a valid access token. If the user is not authenticated the authentication header is `""`.

### 3.2.3. Package frontend.language

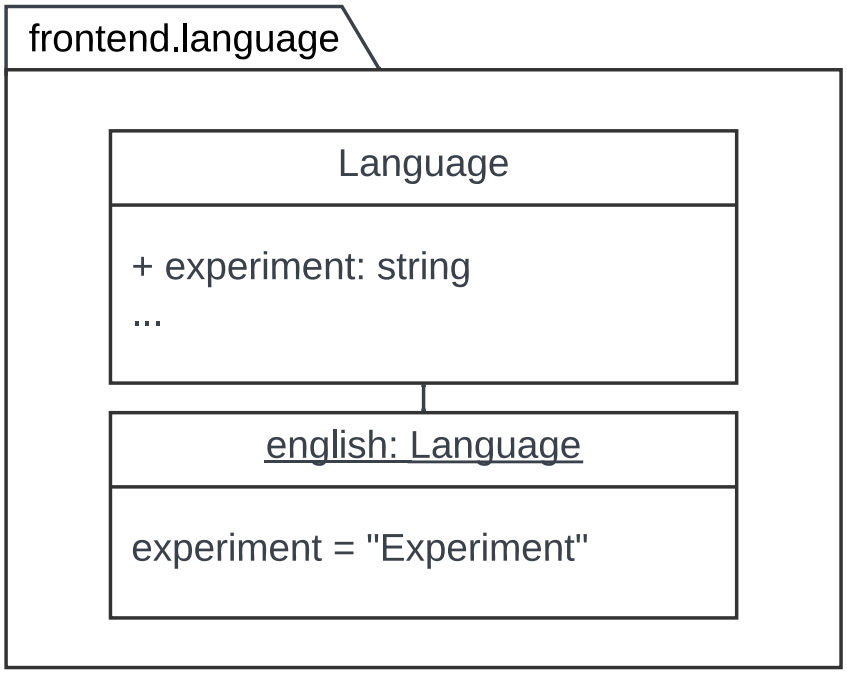


Figure 10: `Language`

#### Class `Language`

Models a language and has as attributes the terms, which are assigned the corresponding translation in the language. For example, for German a `german` object would be created where e.g. the attribute `createExperiment` is assigned `"Neues Experiment"`.

### 3.2.4. Package frontend.parsers

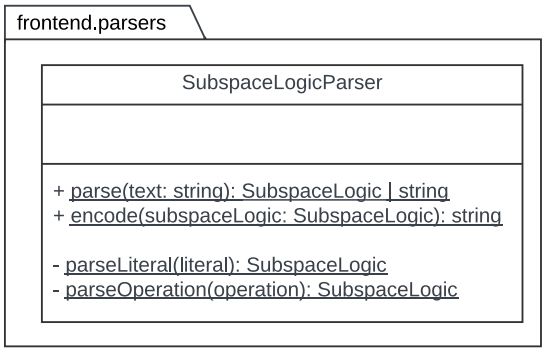


Figure 11: `SubspaceLogicParser`

#### Class `SubspaceLogicParser`

This class provides the functionality to parse and encode `SubspaceLogic` from and to the subspace logic syntax that is shown to the user. Note, that this class does not convert `SubspaceLogic` from and into JSON.

*Methods*

- `public static parse(text: string): SubspaceLogic | string`

  Parses text to a `SubspaceLogic` according to the subspace logic syntax. If the text does not contain a correct subspace logic the method returns an error string.

- `public static encode(subspaceLogic: SubspaceLogic): string`

  Encode a `SubspaceLogic` into the subspace logic syntax

- `private static parseLiteral(literal): SubspaceLogic`

  Parse a substring that represents a literal in the subspace logic syntax

- `private static parseOperation(operation): SubspaceLogic`

  Parse a substring that represents an operation in the subspace logic syntax

### 3.2.5. Package frontend.components.basic

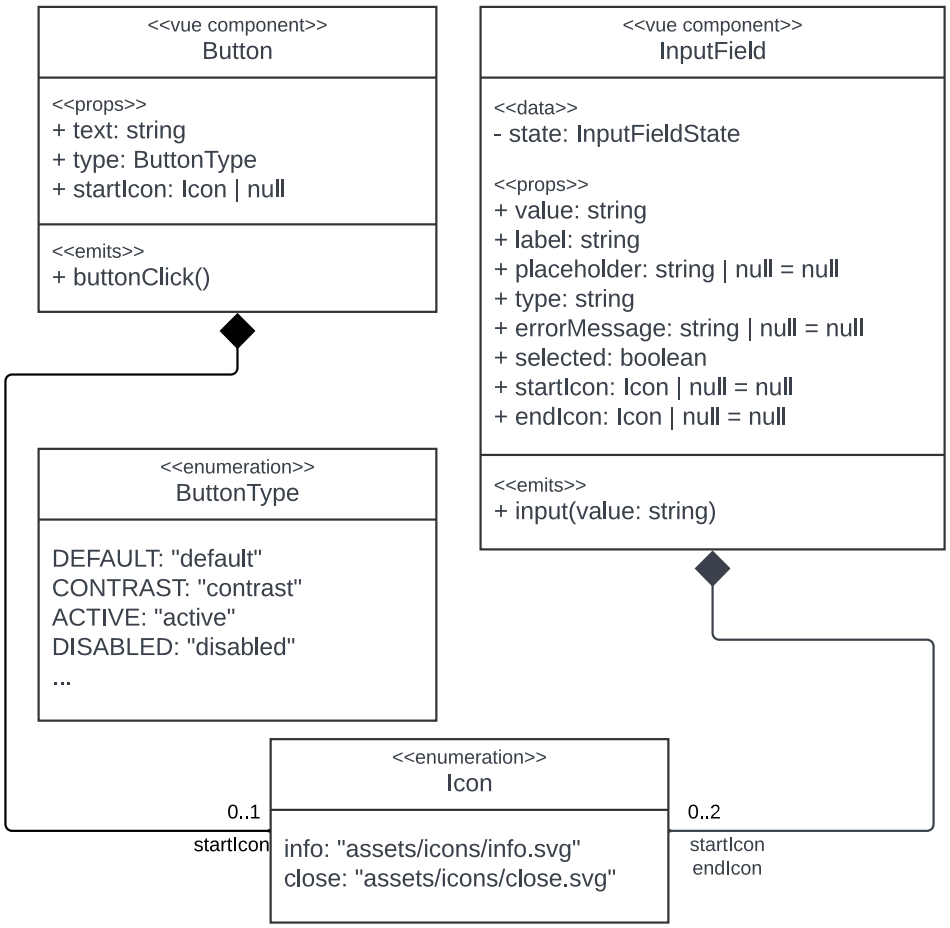


Figure 12: `InputField`, `Button` components with related enums

#### Vue component `Button`

Is clickable, can have an icon and has a type. Which types exist is provided by the `ButtonType` enumeration.

*Props*

- `public text: string`

  Is displayed in the button. It is specified in parent component.

- `public type: ButtonType`

  Is the current type selected for the button.

- `public startIcon: Icon | null`

Is the icon the `Button` is decorated with.

*Emits*

- `public buttonClick(): void`

  The component containing the `Button` can listen to this event and can specify, what happens when this event, the button click occurs.

### Vue component `InputField`

This components allows the user to enter data, which is emitted

*Data*

- `private state: boolean`

  Used to know whether the input field is selected by the user.

*Props*

- `public value: string`

  The input passed in the input field by the user is bound to the variable (of the parent component) specified for the value prop.

- `public label: string`

  Displayed to the user to specify what should be entered.

- `public placeholder: string | null = null`

  Is displayed when the InputField is empty. By default nothing is displayed as placeholder in it

- `public placeholder: string | null`

  Is displayed when the InputField is empty. By default nothing is displayed as placeholder in it

- `public errorMessage: string | null = null`

  Is displayed when input passed by the user causes an error.

- `public selected: boolean`

  Is true when the component is selected.

- `public startIcon: Icon | null = null`

  Is the icon the `InputField` is decorated with. It is displayed at the start, that is on the left. By default no icon is displayed.

- `public startIcon: Icon | null = null`

Is another icon the `InputField` is decorated with. It is displayed at the end, that is on the right. By default no icon is displayed.

*Emits*

- `public input(value: string): void`

  The component containing the `InputField` can listen to this event and can specify, what happens when something is entered in the `InputField` For this the value of the `InputField` is passed with the event.

### Enum `Icon`

This enum contains all Icons that exist in the application. It also stores the source of the icon for each enum entry. There will be more entries to this enum in the future.

- `INFO = "res/icons/info.svg"`

  Is used for decorating tips.

- `CLOSE = "res/icons/close.svg"`

  Is the symbol used for closing, e.g. in the `ClosableCard` component.

### Enum `ButtonType`

This enum describes all possible types a button can have. They are not only used for styling but also affect e.g. the `buttonClick` method.

- `DEFAULT = "default"`

  Represents a default button.

- `CONTRAST = "contrast"`

  Represents buttons with a complementary color to the color scheme (e.g. green download button).

- `ACTIVE = "active"`

  Represents a button that is active and draws the user's focus.

- `DISABLED = "disabled"`

  Represents a button that is not clickable at the moment.

*Attributes*

- `private name: string`

  String representation of each enum entry to be used for CSS classes.

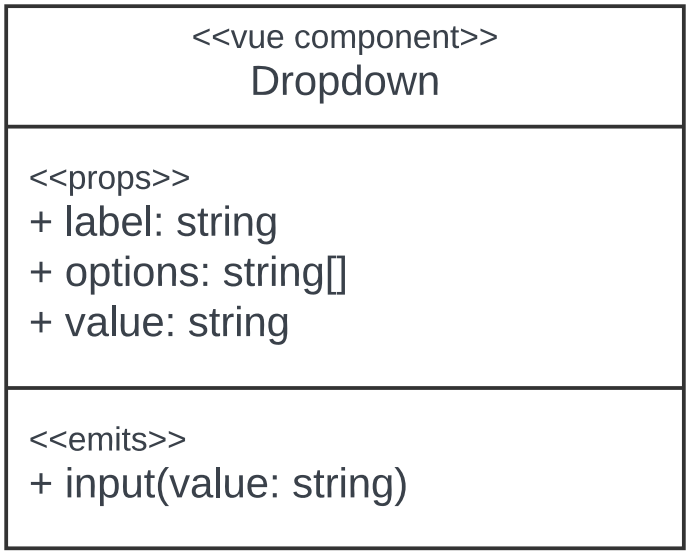


Figure 13: drop-down bar

**Vue component** `Dropdown`

This component shows a drop-down bar.

*Props*

- `public label: string`

  Displayed to the user to specify what is to be selected.

- `public options: string[]`

  Contains all dropdown entries.

- `public value: string`

  Contains the choice the user has selected.

*Emits*

- `public input(value: string): void`

  Emits the choice the user has selected.

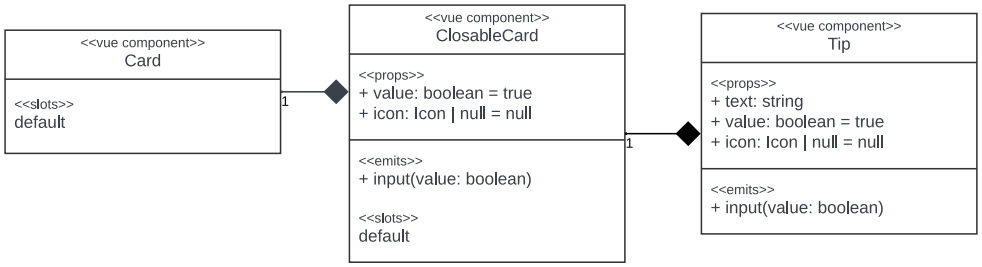


Figure 14: `Card`, `ClosableCard` and `Tip` component

### Vue component `Card`

This component shows a card.

*Slots*

- `default`

  For inserting content into this card

### Vue component `ClosableCard`

This component shows a card, which can be closed by clicking an icon.

*Props*

- `public value: boolean = true`

  Indicates, if the card is closed.

- `public icon: Icon | null = null`

  Indicates, if the card is closed.

*Emits*

- `public input(value: boolean): void`

  Emits when the card is closed or opened.

*Slots*

- `default`

  For inserting content into this card

### Vue component `Tip`

This component shows a tip, which can be closed.

*Props*

- `public text: string`

  Defines the content of the tip.

- `public value: boolean = true`

  Indicates if the card is closed.

- `public icon: Icon | null = null`

  Indicates if the card is closed.

*Emits*

- `public input(value: boolean): void`

  Emits, when the card is closed or opened.

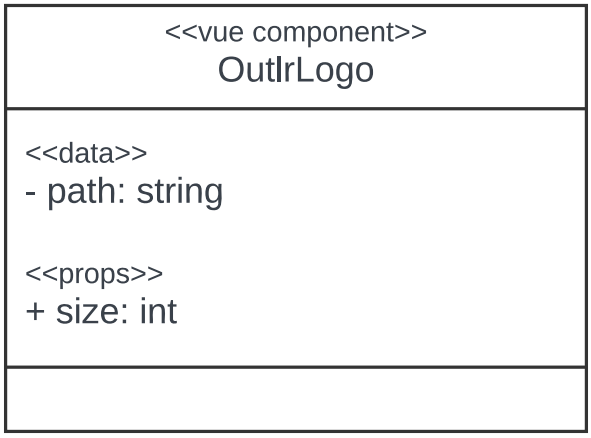


Figure 15: `OutlrLogo`

#### Vue component `OutlrLogo`

This component shows the Logo of Outlr.

*Data*

- `private path: string`

  Stores the path, where the logo is stored.

*Props*

- `public size: int`

  The size, to scale the logo.

### 3.2.6. Package frontend.components.navigationbar

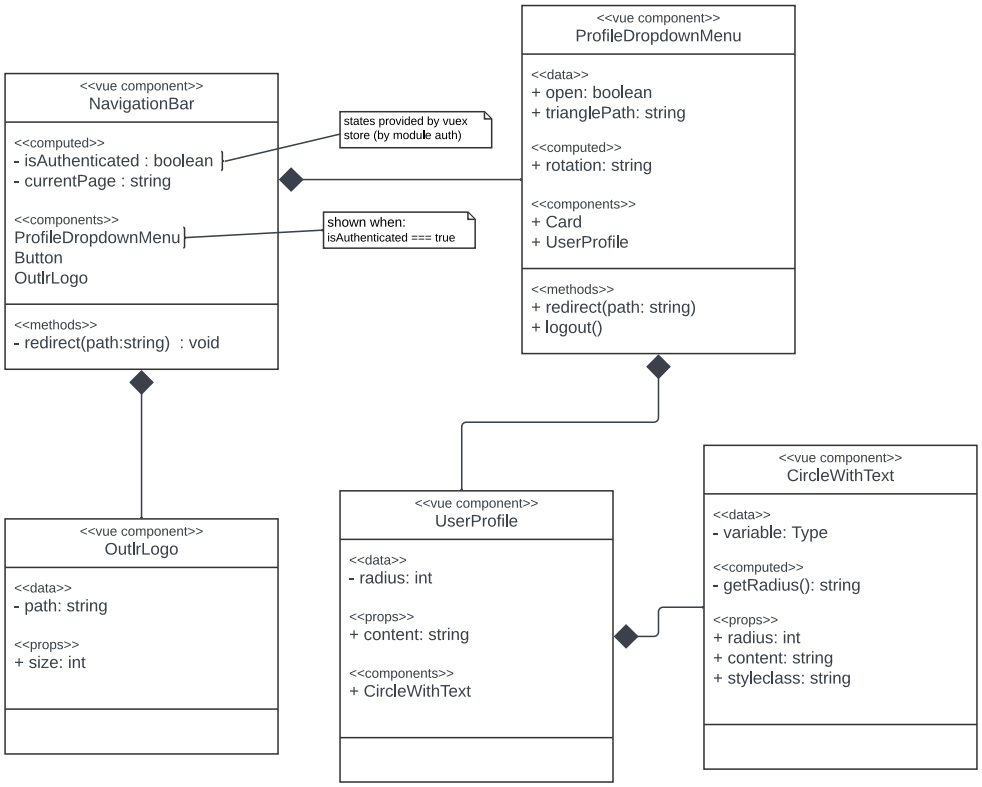


Figure 16: `NavigationBar`, `ProfileDropdownMenu`, `OutlrLogo`, `UserProfile` and `CircleWithText`

#### Vue component `NavigationBar`

Allows the user to navigate the website, that is change the route etc. It is always visible at the top of the website. Depending on the authentication state a login and register button or a create experiment and profile button are shown.

*Computed*

- `private isAuthenticated: boolean`

  Indicates whether the user is logged in to an account, which is derived from the Vuex auth module state. It is passed as a variable for the conditional rendering with the directive `v-if` since a login and register button or a create experiment and profile button are shown depending on the authentication state.

- `private currentPage: string`

Has the current view name as a value and is displayed in the center of the navigation bar.

*Used components*

- `ProfileDropdownMenu`

  If the user is authenticated this component is displayed on the right side of the navigation bar allowing the user to e.g. logout.

- `Button`

  If the user is authenticated a create experiment button is shown and when pressed, redirects the user to the create experiment page.

  If the user is not authenticated a log in button is shown and when pressed, redirects the user to the login page.

  If the user is not authenticated a sign up button is shown and when pressed, redirects the user to the register page.

- `OutlrLogo`

  Is placed at the top right corner.

*Methods*

- `public redirect(path: string): void`

  This (wrapper) method redirects the user to the component linked to the path passed as a parameter if a valid route path was provided. Internally this method calls a method from the Vue router for redirection.

### Vue component `ProfileDropdownMenu`

This component can display a card in a dropdown style. Clicking a triangle will toggle if a card is shown. The rotation of the triangle indicates if the card is open or closed. The card shows the username and clickable text: "Logout". After clicking "Logout", the user will be logged out from his account.

*Data*

- `private open: boolean`

  Indicates, if the card is open or not

- `private trianglePath: string`

  Contains the path, where the triangle image is stored

*Computed*

- `public rotation: string`

  Calculates the rotation of the triangle

*Used components*

- `Card`

  Displays a card in dropdown style

- `UserProfile`

  Shows a user icon

*Methods*

- `public redirect(path: string): void`

  Redirects the user to the component linked to the path passed as a parameter

- `public logout(): void`

  Logs the user out of his account

#### Vue component `UserProfile`

This component shows a user icon, consisting of the username's first letter on a purple circle with a radius. This component shows a user icon, consisting of the username's first letter on a purple circle with a radius.

*Data*

- `private radius: int`

  Describes the radius of the circle

*Props*

- `public content: string`

  Contains the letter, which will be displayed on the circle

*Used components*

- `CircleWithText`

  Used as a background circle

#### Vue component `CircleWithText`

This component draws a circle with text inside it and a given radius. The styling is defined through its style class.

*Computed*

- `private getRadius: string`

  Retrieves the radius for the circle

*Props*

- `public radius: int`

  Describes the radius of the circle

- `public content: string`

  Describes the content inside of the circle

- `public styleclass: string`

  Defines the styleclass of this component

### 3.2.7. Package frontend.components.basic

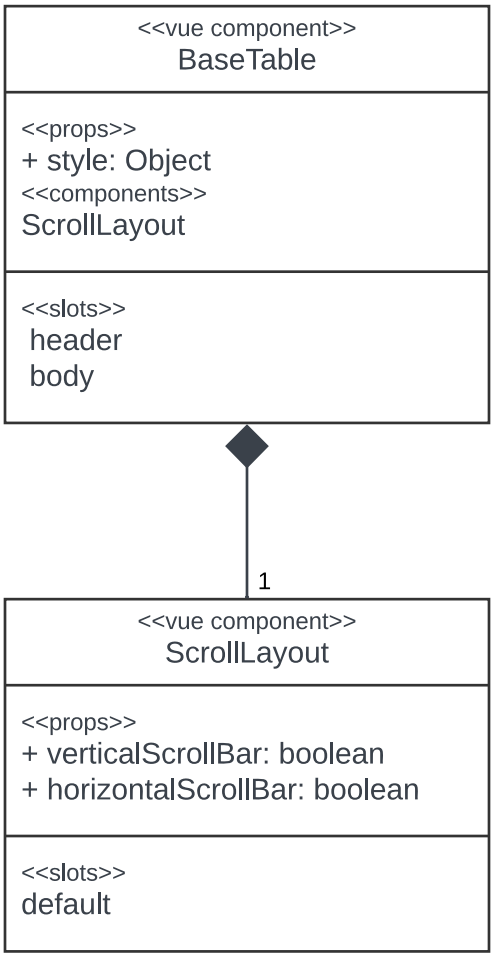


Figure 17: `BaseTable`

### Vue component `BaseTable`

This component is the base table that is used by all of our tables.

*Props*

- `public style: Object`

  contains a style for the table

*Used components*

- `ScrollLayout`

*Slots*

- `header`

  slot for the table header

- `body`

  slot for the table body

### Vue component `ScrollLayout`

This component handles the scrolling of a table.

*Props*

- `public verticalScrollBar: boolean`

  indicates if `default` is vertically scrollable

- `public horizontalScrollBar: boolean`

  indicitas if `default` is horzontally scrollable

*Slots*

- `default`

  slot for the scrollable tables

### 3.2.8. Package frontend.components.views.landingpage

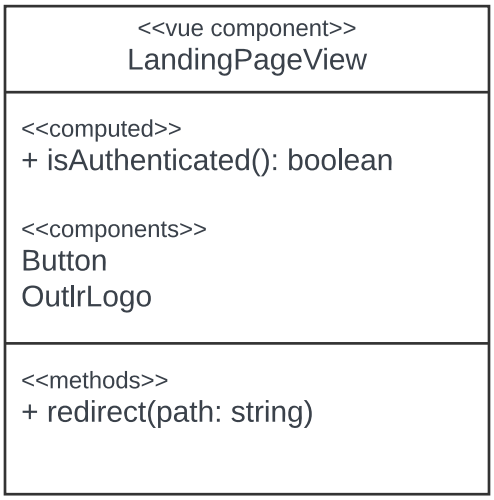


Figure 18: `LandingPageView`

#### Vue component `LandingPageView`

The main purpose of the landing page view is to allow the user to welcome the user to our web application and provide easy access to common features like creating an account or navigating to the dashboard.

*Computed*

- `private isAuthenticated: boolean`

  Indicates whether the user is logged in to an account, which is derived from the Vuex auth module state.

*Used components*

- `Button`

  If the user is authenticated a sign up button is shown, where the user can be directed to the sign up page.

  Else, a dashboard button is shown, which redirects the user to the dashboard page.

- `OutlrLogo`

  The Logo is shown in the middle of the page.

*Methods*

- `public redirect(path: string): void`

Redirects the user to the component linked to the path passed as a parameter if a valid route path was provided.

### 3.2.9. Package frontend.components.views.login

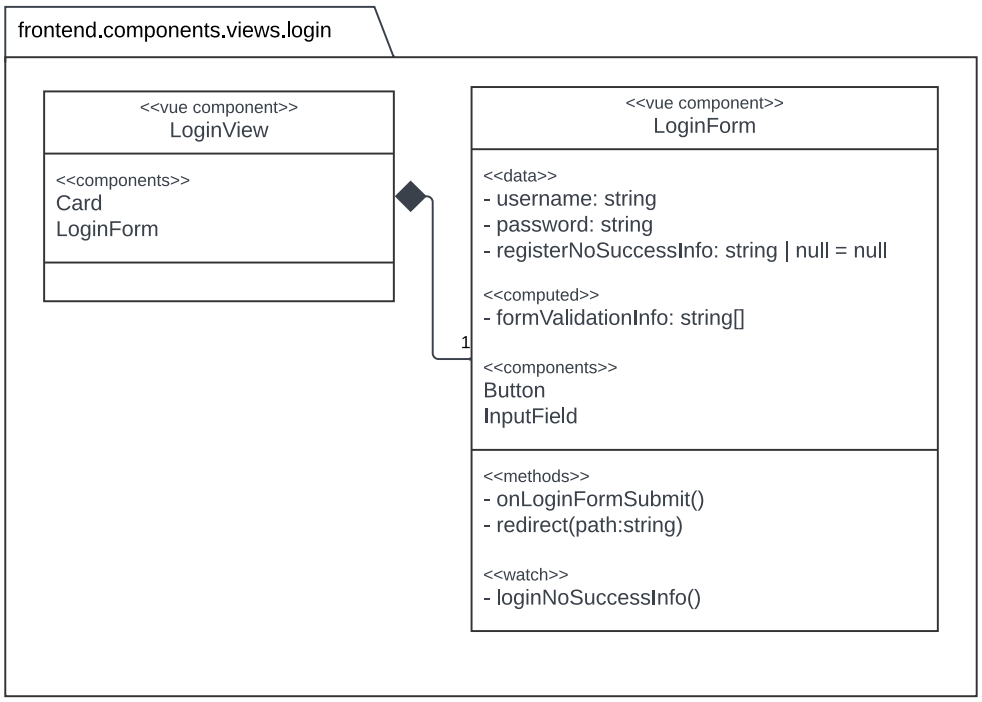


Figure 19: `LoginView` and `LoginForm`

#### Vue component `LoginView`

The main purpose of this view is to allow the user to log in to his account, which then when successfully logged in, can view his already created experiments on the dashboard page or create a new one on the `CreateExperimentView`The Login view is only accessible when a user is not authenticated, see 4.2 for more info on this.

*Used components*

- `LoginForm`

  Used, to let the user input data into a form.

- `Card`

  Used, to display inside a card.

#### Vue component `LoginForm`

Is a form with 2 text input fields, one for the username and another for the password

*Data*

- `private username: string`

  Is binded to the username text input field and always corresponds to the content in that input field

- `private password: string`

  Is binded to the password text input field and always corresponds to the content in that input field

- `private loginNoSuccessInfo: string = null`

  Carries the info provided by the API, when submitting the form data (username and password) was not successful

*Computed*

- `private formValidationInfo: string[]`

  Carries info on incorrectly passed form data for each input field

*Used components*

- `Button`

  Is used to submit the form with the provided input data

*Methods*

- `public onLoginFormSubmit(): void`

  Method executed when `buttonClick` event occurs. Submits the data passed by the user to the back-end. If valid user data was passed the user is logged into his account and redirects him to the dashboard for which the `redirect` method is used. Internally,the Vuex module auth is used for the login action.

- `public redirect(path: string): void`

  Redirects the user to the component linked to the path passed as parameter, if a valid route path was provided

### 3.2.10. Package frontend.components.views.register

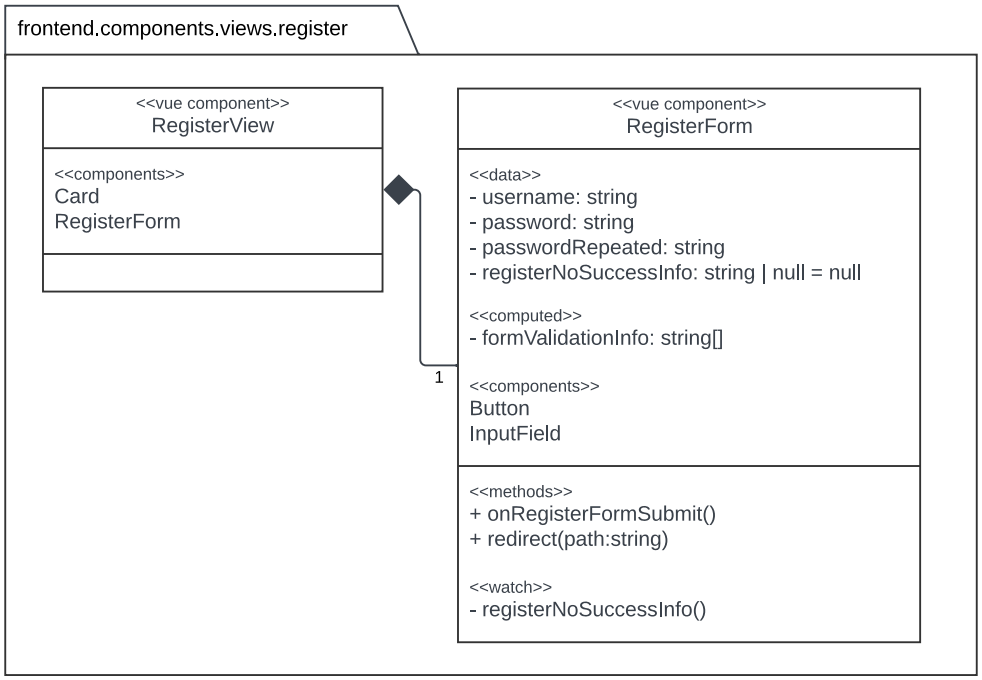


Figure 20: `RegisterView` and `RegisterForm`

#### Vue component `RegisterView`

The main purpose of this view is to allow a user to create an account. The Register view is only accessible when a user is not authenticated, see 4.2 for more info on this.

*Used components*

- `RegisterForm`

  Used, to let User input data into a form

- `Card`

  Used, to display the form on a card

#### Vue component `RegisterForm`

Is a form with 3 text input fields, one for the username, another for the password and the last for the password, which has to be reentered. Each text input field is bound to a data property.

*Data*

- `private username: string`

  The username data variable is bound to the username text input field and always corresponds to the content in that input field.

- `private password: string`

  The password data variable is bound to the password text input field and always corresponds to the content in that input field.

- `private passwordRepeated: string`

  Is bound to the password repeated text input field and always corresponds the content in that input field.

- `private registerNoSuccessInfo: string = null`

  Carries the info provided by the API, when submitting the form data was not successful

*Computed*

- `private formValidationInfo: string[]`

  Carries info on incorrectly passed form data for each input field

*Used components*

- `Button`

  Is used to submit the form with the provided input data

*Methods*

- `public onRegisterFormSubmit(): void`

  Method executed when `buttonClick` event occurs. It submits the data passed by the user to the back-end and. If valid user data was passed and if the username is not already used, creates an account for the user and redirects him to his (empty) dashboard, for which `redirect` is used.

- `public redirect(path: string): void`

  Redirects the user to the component linked to the path passed as parameter, if a valid route path was provided

### 3.2.11. Package frontend.components.views.dashboard

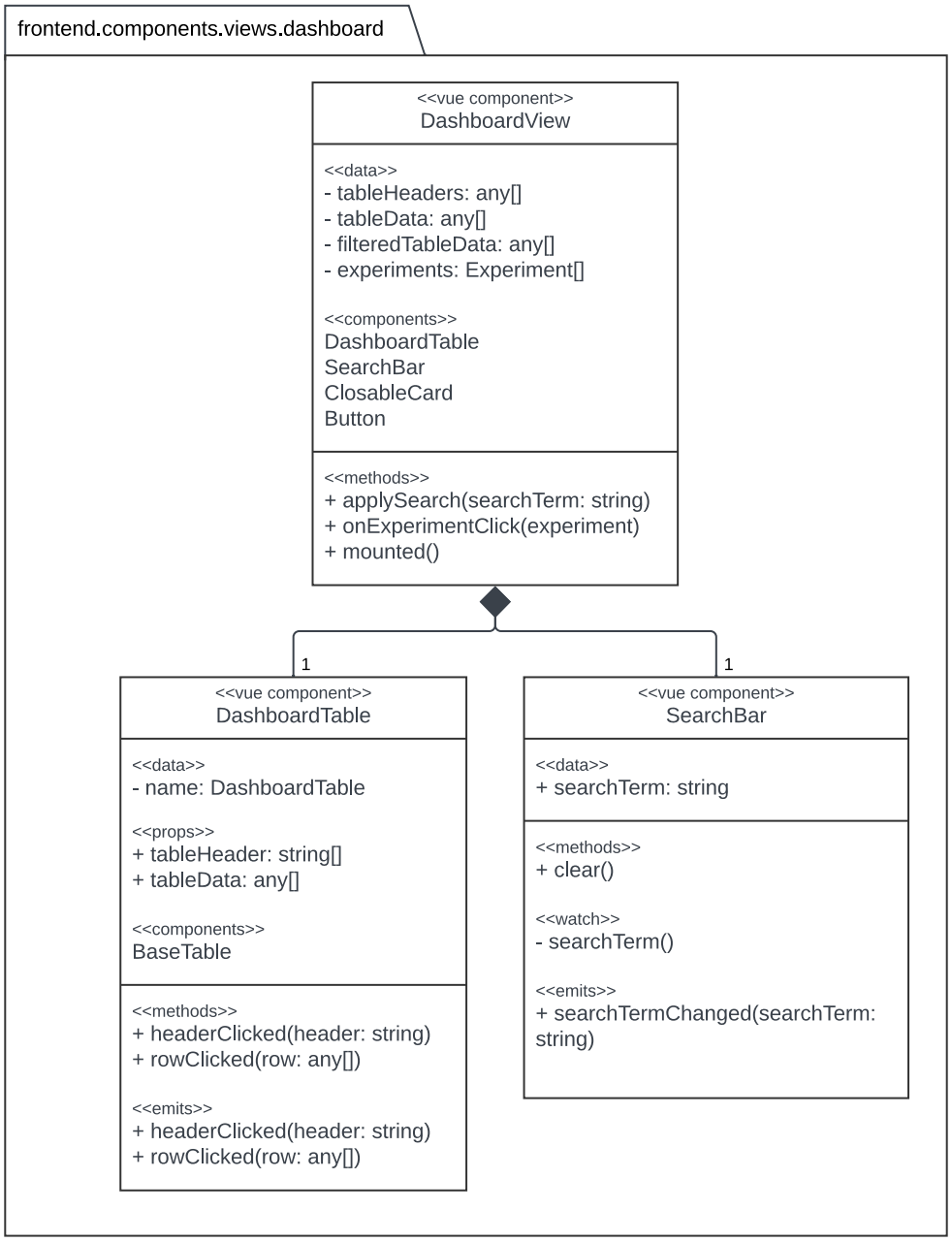


Figure 21: `DashboardView`, `DashboardTable` and `SearchBar`

#### Vue component `DashboardView`

This component gives the user an overview of all his experiments. The user can access the Experiments to review their results by clicking on the Experiment in the Table.

*Data*

- `private tableHeaders: any[]`

  Contains the Headers for the `DashboardTable`

- `private tableData: any[]`

  Contains the data for every experiment the user has created

- `private filteredTableData: any[]`

  is given to the `DashboardTable` to display the data The array is per default the same as the table data but gets altered when the user filters the table. When the user clears the filter the array is set back to the default.

- `private experiments: Experiment`

  Contains the experiments in its experiment class form. The array is used to extract the data from each experiment to then insert it into the `tableData` array.

*Used components*

- `DashboardTable`

  This component is used to display the Experiments of a user in a Table.

- `SearchBar`

  This component is an input field where the user can input a search term. The `DashboardTable` is then filtered to display only those experiments that contain the search term in some way.

- `ClosableCard`

  This component is used to display information to the User. It is intended to show a hint on how to select one or multiple rows in the `DashboardTable`.

- `Button`

  The clear search Button is used to reset the `searchTerm` and the `DashboardTable` to their default state.

*Methods*

- `public applySearch(searchTerm: string): void`

  The method filters the `tableData` array making it only contain rows that include the `searchTerm` in some way. The result then gets saved in `filteredTableData` array.

- `public onExperimentClick(experiment): void`

  The method redirects the user to the `ExperimentResultView` and sends the experiment id that the user clicked to the `ExperimentResultView` component.

- `public mounted(): void`

  This method requests all experiments of the user from the backend.6.

#### Vue component `DashboardTable`

This component is the table component for the dashboard. It contains the styling of the table and handles the functions that are executed when the user clicks on the table.

*Props*

- `public tableHeader: string[]`

  Contains the headers of the table

- `public tableData: any[]`

  Contains the data of the table

*Used components*

- `BaseTable`

  This component is used for the basic table structure and style of the DashboardTable

*Methods*

- `public headerClicked(header: string): void`

  This method emits the `headerClicked` with the clicked header.

- `public rowClicked(row: any[]): void`

  This method emits the `rowClicked` with the clicked row.

*Emits*

- `public headerClicked(header: string): void`

  Allows the parent component to specify what happens when the header is clicked.

- `public rowClicked(row: any[]): void`

  Allows the parent component to specify what happens when the row is clicked.

#### Vue component `SearchBar`

This Component is an input field for the user. It is used for filtering the `DashboardTable`.

*Data*

- `public searchTerm: string`

  contains the user input of the input field

*Methods*

- `public clear(): void`

  This method resets the `searchTerm` to an empty string.

*Watches*

- `private searchTerm(): void`

  watches the `searchTerm` and emits the `searchTerm` on change

*Emits*

- `public searchTermChanged(searchTerm: string): void`

  Allows the parent component to specify what happens on `searchTermChanged` .

### 3.2.12. Package frontend.components.views.createexperiment

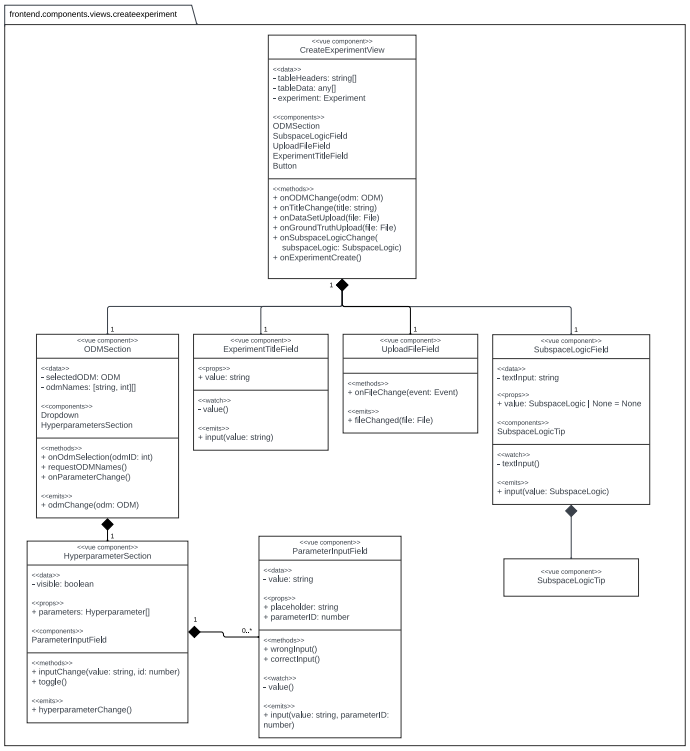


Figure 22: `CreateExperimentView`

#### Vue component `CreateExperimentView`

This component is the main component to create an Experiment

*Data*

- `private tableHeaders: string[]`

  Contains the headers of the dataSetPreview table

- `private tableData: any[]`

  Contains the data of the dataSetPreview table

- `private experiment: Experiment`

  Contains the experiment and is updated every time the user inputs something

*Used components*

- `ODMSection`

  Contains the input fields for the selection of the ODM.

- SubspaceLogicField
- UploadFileField
- ExperimentTitleField
- Button

  Button to create the experiment

*Methods*

- `public onODMChange(odm: ODM): void`

  This method sets the ODM of the experiment when the user selects something and enters hyperparameters.

- `public onTitleChange(tilte: string): void`

  This method sets the name of the experiment when the user inputs something.

- `public onDataSetUpload(file: File): void`

  This method is executed when the user uploads a dataset. It then sends the file to the backend for verification.

- `public onGroundTruthUpload(file: File): void`

  This method is executed when the user uploads a ground thruth file. It then sends the file to the backend for verification.

- `public onSubspaceLogicChange(subspaceLogic: SubspaceLogic): void`

  This method inserts the SubspaceLogic into the experiment attribute.

- `public onExperimentCreate(): void`

  This method is executed when the create experiment button is pressed. It calls the corresponding api request function to send the created experiment to the backend.

### Vue component ODMSection

This Component contains the functions and components that are necessary for the ODM selection.

*Data*

- `private selectedODM: ODM`

  contains the ODM element

- `private odmNames: [string, int][]`

  contains a tuple array of all available ODM names with their corresponding ids

*Used components*

- `Dropdown`

  This component lets the user select and `ODM` from the `odmNames` array.

- `HyperparameterSection`

*Methods*

- `public onOdmSelection(odmID: int): void`

  This method requests the `ODM`, the user selected, from the backend.

- `public requestODMNames(): void`

  This method requets the list of all available `ODM` from the backend

- `public onParameterChange(): void`

  This method emits `odmChange` .

*Emits*

- `public odmChange(odm: ODM): A`

  llows the parent component to specify what happens on odmChange.

### Vue component `HyperparameterSection`

This component is for the hyperparameter selection. It is visible when the user selected a specific `ODM`.

*Data*

- `private visible: boolean`

  indicates, if parameters are visible or not

*Props*

- `public parameters: Hyperparameter[]`

  contains the hyperparameters of the selected `ODM`

*Used components*

- `ParameterInputField`

  These components allow the user to input the hyperparameters.

*Methods*

- `public inputChange(value: string, id: number): void`

  This method emits the `hyperparameterChange` with the value as string and the parameterID as number.

- `public toggle(): void`

  This method toggles the visibility of the `HyperparameterSection`

*Emits*

- `public hyperparameterChange(): void`

  Allows the parent component to specify what happens when the hyperparmeters change.

#### Vue component `ParameterInputField`

This component is an input field for an parameters with a specific type. It shows some kind of indicatin when the user trys to insert a wrong type.

*Data*

- `private value: string`

  contains the user input

*Props*

- `public placeholder: string`

  contains the name of the parameter

- `public parameterID: number`

  contains the id of the parameter the input field is for

*Methods*

- `public wrongInput(): void`

  This method shows the user some indication when the `value` has the wrong type that is needet for the parameter.

- `public correctInput(): void`

  This method shows the user some indication when the `value` is in the correct type.

*Watches*

- `private value(): void`

  This method emits the changed `input`

*Emits*

- `public input(value: string, parameterID: number): void`

  This method allows the parent component to specify what happens when the input changes.

### Vue component `ExperimentTitleField`

This component is an Input field specify for the task of changing the experiment name.

*Props*

- `public value: string`

  contains the user input

*Watches*

- `private value(): void`

  This method emits the changed `input` .

*Emits*

- `public input(value: string): void`

  This method allows the parent component to specify what happens when the input changes.

### Vue component `UploadFileField`

This component is for the upload of a File.

*Methods*

- `public onFileChange(event: Event): void`

  This method gets an event when the user uploads a file. It then emits the `fileChanged` to the parent component.

*Emits*

- `public fileChanged(file: File): void`

  This method allows the parent component to specify what happens when the file changes.

### Vue component `SubspaceLogicField`

This component is an input where the user can input a subspace logic. When the input is correct the parsed string gets emitted.

*Data*

- `private textInput: string`

  contains the user input

*Props*

- `public value: SubspaceLogic | None`

  can contain a subspace logic when provided by the parent component

*Used components*

- `SubspaceLogicTip`

*Watches*

- `private textInput(): void`

  This method trys to parse the subspace logic when the user inserts something. When the subspace logic can be parsed the `input` is emitted.

*Emits*

- `public input(value: SubspaceLogic): void`

  This method allows the parent component to specify what happens when a correct subspace logic is inserted.

### Vue component `SubspaceLogicTip`

This component contains some tips for the user on how the subspace logic syntax works.

### 3.2.13. Package frontend.components.views.experimentresult

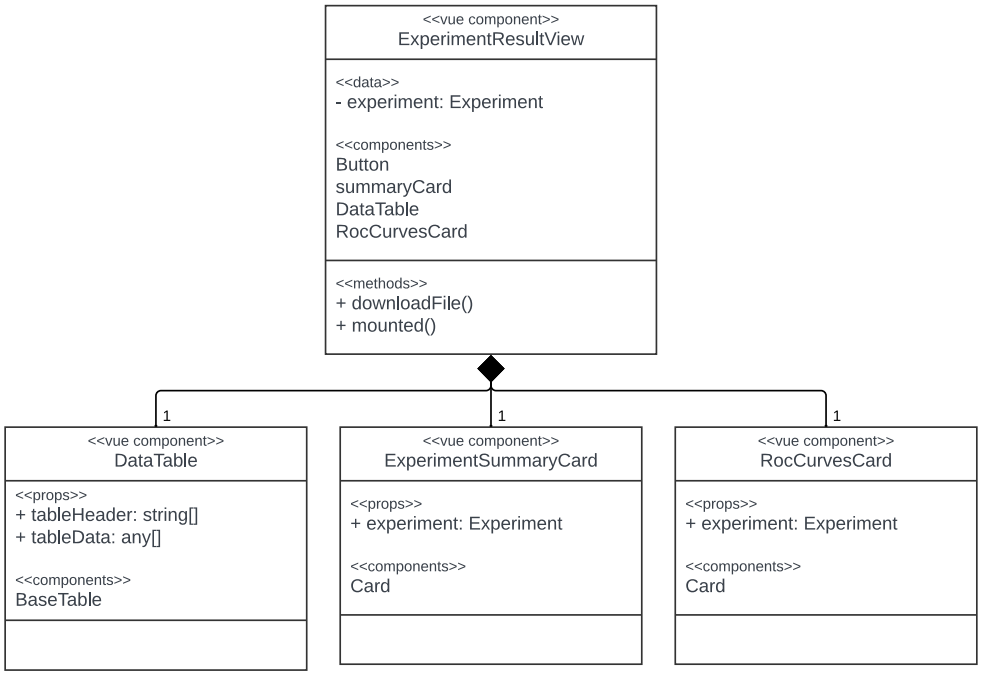


Figure 23: `ExperimentResultView`

#### Vue component `ExperimentResultView`

This component gives the user an detailed overview of an experimnt.

*Data*

- `private experiment: Experiment`

  contains the experiment and all of its data

*Used components*

- `Button`

  The download button can be used to download data of the experiment.

- `ExperimentSummaryCard`
- `DataTable`

  This component is a table for showing data of the experiment

- `RocCurvesCard`

*Methods*

- `public downloadFile(): void`

  This method starts a download to a file.

- `public mounted(): void`

  This method requests the experiment from the backand an inserts it into the `experiment` attribute.

#### Vue component `DataTable`

This component is a table that can show the data that is provided.

*Props*

- `public tableHeader: string[]`

  contains the strings for the headers

- `public tableData: any[]`

  contains the data that is shown in the table

*Used components*

- `BaseTable`

#### Vue component `ExperimentSummaryCard`

This component is a card that shows a summary of the experiment result.

*Props*

- `public experiment: Experiment`

  contains the experiment to display the summary

*Used components*

- `Card`

#### Vue component `RocCurvesCard`

This component is a card that shows the ROC-Curves.

*Props*

- `public experiment: Experiment`

  contains the experiment to display the ROC-Curves.

*Used components*

- `Card`

## 3.3. Backend

The back-end mainly relies on the libraries Flask, PyOD, and SQLAlchemy. Flask is for creating the API, PyOD provides ODMs, pandas is for dataset handling and SQLAlchemy allows easy database access and ORM. In the following, the usage and interaction of these libraries will be illustrated.

### 3.3.1. Package backend

#### Class `Application`

This class is the entry point of the backend and defines a global state. It sets up the ODMs in the database and starts the API.

*Attributes*

- `private experiment_scheduler: ExperimentScheduler`

  Defines which scheduler should be used for experiment execution.

- `private odm_providers: dict[ODMType, ODMProvider]`

  Contains a mapping for every `ODMType` to their provider. Allows for scraping ODMs from multiple sources. If an `ODMType` is mapped to a provider, each `ODM` of this type is marked as deprecated. The provider then returns which ODMs of the given type are available from now on. The provider is not allowed to modify ODMs of any other type.

*Methods*

- `private run_odm_providers(): None`

  Collects all ODMs using the `odm_providers`

- `private start_api(): None`

  Starts the API.

### 3.3.2. Package backend.api

#### Class `API`

This class manages the routes of the API.

*Methods*

- `public status(): Response`

  Always returns status code "200 OK". Can be used by clients to check if the API is reachable.

#### Class `ExperimentAPI`

This class defines API endpoints which are experiment related.

*Methods*

- `public validate_dataset(): Response`

  Requires a jwt access token. Expects a dataset as a CSV file in the request. Returns status code "200 OK" if the dataset was valid, "400 Bad Request" with message "Invalid dataset." otherwise.

- `public validate_ground_truth(): Response`

  Requires a jwt access token. Expects a ground truth file as a CSV file in the request. Returns status code "200 OK" if the ground truth file was valid, "400 Bad Request" with message "Invalid dataset." otherwise.

- `public get_result(): Response`

  Requires a jwt access token. Expects the experiment id in the request. If the experiment was found, returns status code "200 OK" and the experiment results encoded as json. If there is no experiment with the given ID it returns "404 Not Found".

- `public get_all(): Response`

  Requires a jwt access token. Returns a list of all experiments the user has encoded as json and status code "200 OK".

- `public create(): Response`

  Requires a jwt access token. Expects an experiment encoded as json in the request. Inserts the experiment in the database and runs it.

- `public download_result(): Response`

  Requires a jwt access token. Expects the experiment id in the request. If the experiment was found, returns a CSV file with all the outliers from the given experiment and status code "200 OK". If there is no experiment with the given ID it returns "404 Not Found".

### Class `UserManagementAPI`

This class defines API endpoints which are user management related.

*Methods*

- `public register(): Response`

  Expects a username and a password in the request. Inserts a new user into the database if username and password are valid and returns a jwt access token.

- `public login(): Response`

  Expects a username and a password in the request. If username and password were correct a jwt access token and the status code "200 OK" is returned, status code "401 Unauthorized" otherwise.

- `public check_token(): Response`

  Requires a jwt access token. Returns status code "200 OK" if the provided access token is still valid.

### Class `ODMAPI`

This class defines API endpoints for retrieving available ODMs.

*Methods*

- `public get_all(): Response`

  Requires a jwt access token. Returns a list of all ODMs available to the user encoded as json and status code "200 OK".

- `public get_parameters(): Response`

  Requires a jwt access token. Expects an ODM id. Returns a list of all parameters the given ODM has encoded as json and status code "200 OK" if the ODM was found, status code "400 Bad Request" otherwise.

### 3.3.3. Package backend.database

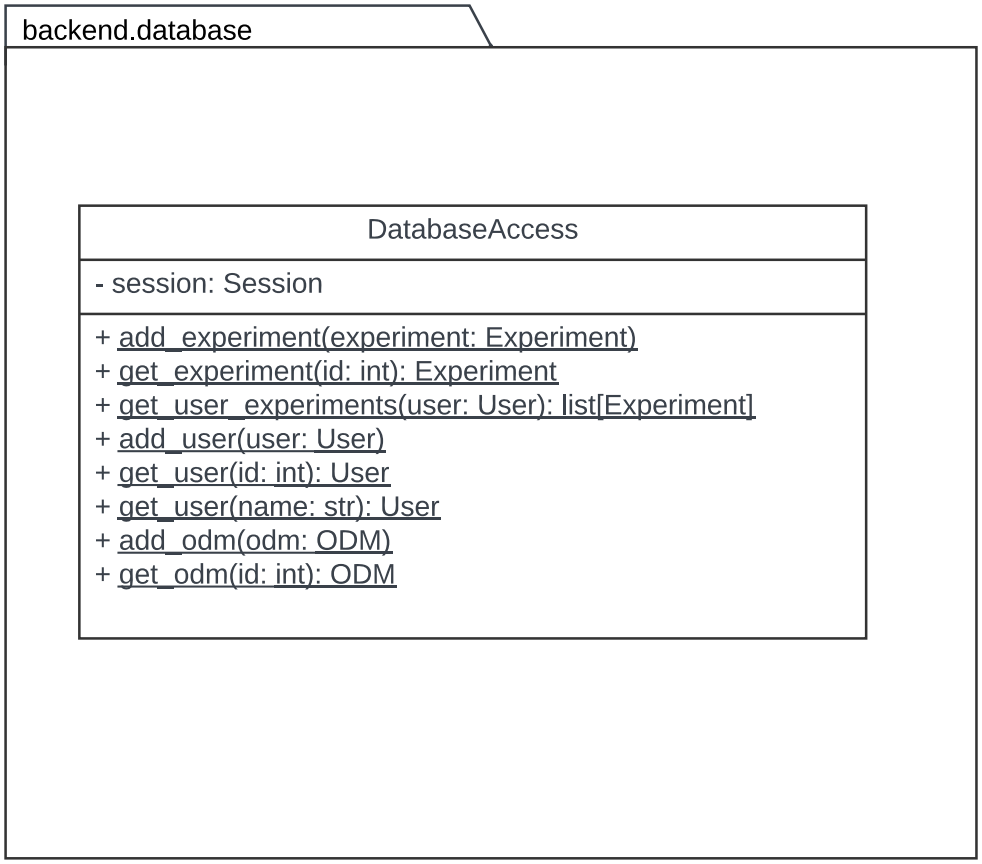


Figure 24: database package with `DatabaseAccess` class

#### Class `DatabaseAccess`

This class handles all database access methods.

*Attributes*

- `private session: Session`

*Methods*

- `public add_experiment(experiment: Experiment): void`

  Write an experiment to the database

- `public get_experiment(id: int): Experiment`

  Read the experiment with the given `id` from the database

- `public get_user_experiments(user: User): list[Experiment]`

  Read a list of all experiments created by a user from the database

- `public add_user(user: User): void`

  Write a user to the database
- `public get_user(id: int): User`

  Read the user with the given `id` from the database
- `public get_user(name: str): User`

  Read the `User` with the given `id` from the database
- `public add_odm(odm: ODM): void`

  Write an `ODM` to the database
- `public get_odm(id: int): ODM`

  Read the `ODM` with the given `id` from the database

### 3.3.4. Package backend.models.experiment

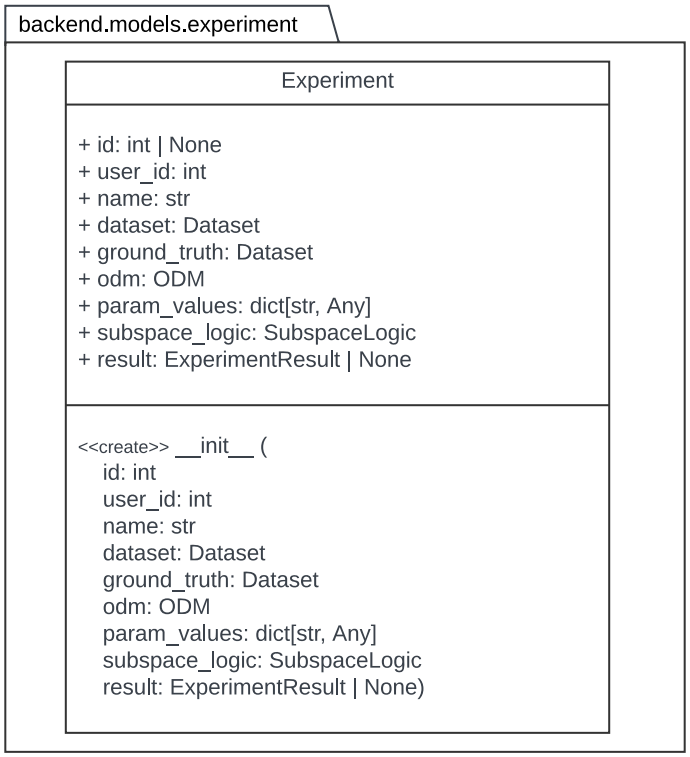


Figure 25: experiment package

#### Class `Experiment`

This class represents the experiment table.

*Attributes*

- `public id: int | None`
- `public user_id: int`
- `public name: str`
- `public dataset: Dataset`
- `public ground_truth: Dataset`
- `public odm: ODM`
- `public param_values: dict[str, Any]`
- `public subspace_logic: SubspaceLogic`
- `public result: ExperimentResult | None`

*Methods*

- ```
  public __init__(
      id: int
      user_id: int
      name: str
      dataset: Dataset
      ground_truth: Dataset
      odm: ODM
      param_values: dict[str, Any]
      subspace_logic: SubspaceLogic
      result: ExperimentResult | None
  ): void
  ```
  ```

### 3.3.5. Package backend.models.dataset

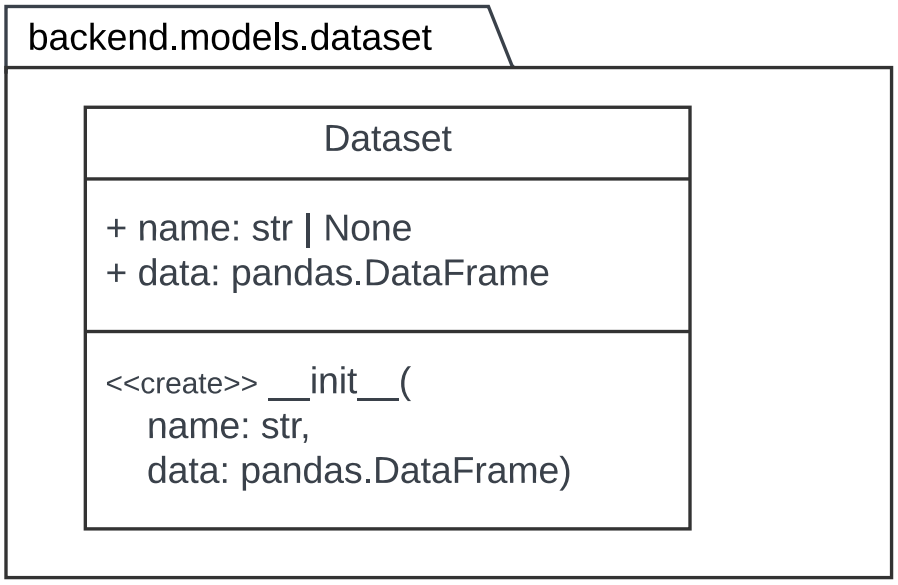


Figure 26: dataset package

#### Class `Dataset`

This class represents the dataset table.

*Attributes*

- `public name: str | None`
- `public data: pandas.DataFrame`

*Methods*

- `public __init__(name: str, data: pandas.DataFrame): void`

### 3.3.6. Package backend.models.user

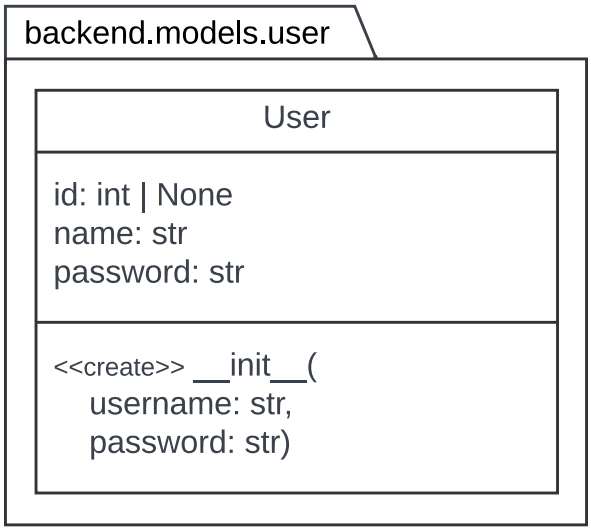


Figure 27: `User`

#### Class `User`

This class represents the user table.

*Attributes*

- `public id: int | None`
- `public name: str`
- `public password: str`

*Methods*

- `public __init__(username: str, password: str): void`

### 3.3.7. Package backend.models.odm

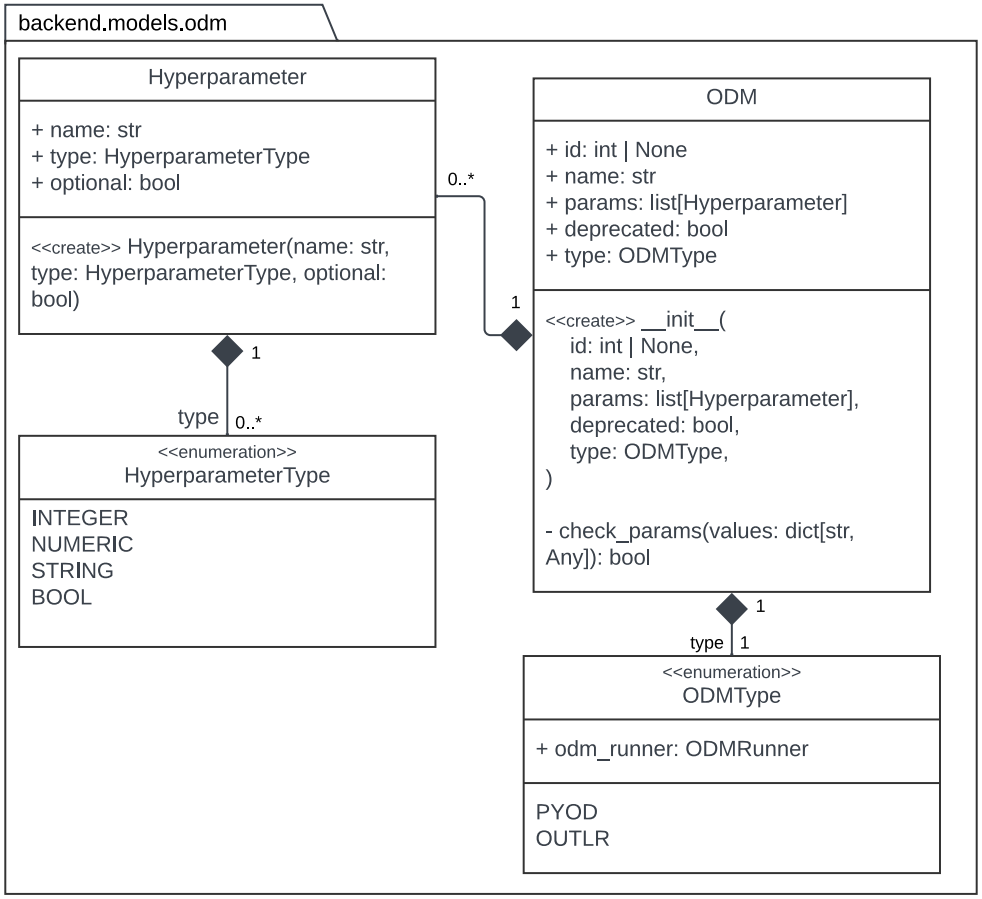


Figure 28: Contains classes and enumerations, which have to do with the ODM. These are: `ODM`, `Hyperparameter`, `HyperparameterType`, `ODMType`

#### Class `ODM`

Models an ODM on the backend

*Attributes*

- `public id: int | None`

  Assigns the ODM an id, which is also used in the database

- `public name: str`
- `public params: list[Hyperparameter]`
- `public deprecated: bool`
- `public type: ODMType`

*Methods*

- `public __init__(id: int | None, name: str, params: list[Hyperparameter], deprecated: bool, type: ODMType): void`
- `private check_param(values: dict[str, Any]): bool`

  Checks whether the values passed, which should be used for the parameters, would be valid parameters for the ODM

### Class `Hyperparameter`

Models a hyperparameter for an `ODM`

*Attributes*

- `public name: str`
- `public type: HyperparameterType`
- `public optional: bool`

*Methods*

- `public __init__(name: str, type: HyperparameterType, optional: bool): void`

### Enum `HyperparameterType`

Defines what type a hyperparameter has

- `INTEGER`
- `NUMERIC`

  A number, which can also be a float

- `STRING`
- `BOOL`

### Enum `ODMType`

Defines of what type the ODM is, that is e.g. if the ODM is from the PyOD library

- `PYOD`

  ODM is from the PyOD library

- `OUTLR`

  ODM from us

### 3.3.8. Package backend.models.subspacelogic

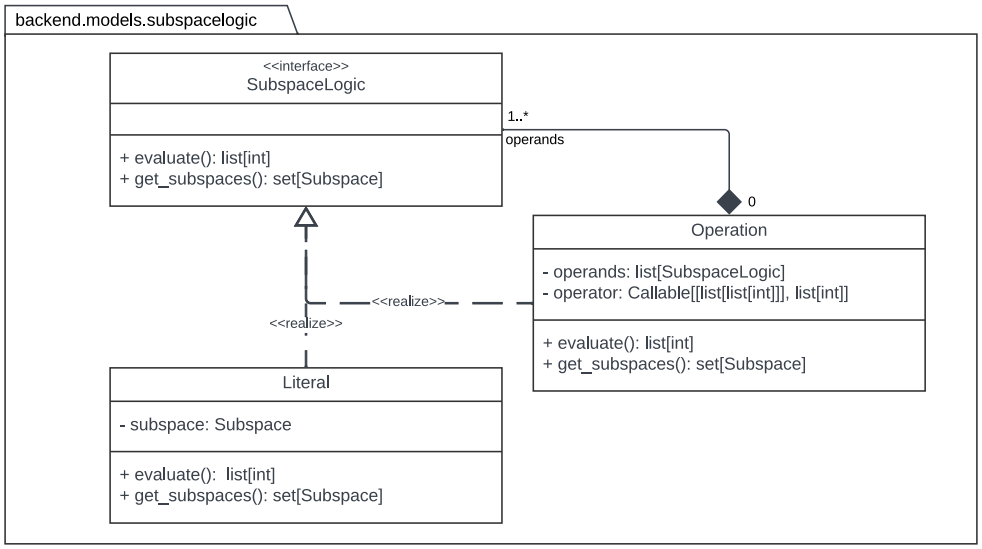


Figure 29: `SubspaceLogic`, `Literal` and `Operation`

#### Interface `SubspaceLogic`

This interface represents the subspace logic using a composite pattern.

*Methods*

- `public evaluate(): list[int]`

  Evaluates the subspace logic

- `public get_subspaces(): set[Subspace]`

  Returns the subspaces that are contained in the subspace logic

*Implemented by* `Literal` `Operation`

#### Class `Literal`

This class represents a literal in the subspace logic, which is a leaf in the tree.

`implements SubspaceLogic`

*Attributes*

- `private subspace: Subspace`

  Contains a single subspace as part of the subspace logic

*Methods*

- `public evaluate(): list[int]`

  Evaluates the subspace logic. It is required that all contained `Literal.subspace` have a valid result specified in `Subspace.outliers`.

- `public get_subspaces(): set[Subspace]`

  Returns the subspaces that are contained in the subspace logic

### Class `Operation`

This class represents an operation in the subspace logic.

`implements SubspaceLogic`

*Attributes*

- `private operands: SubspaceLogic`

  Contains the operands of the operation.

- `private operands: Callable[[list[list[int]]], list[int]]`

  A function that defines the combination of integer lists where each integer is either `1` or `0`. An easy example would be a logical operation that acts pointwise on its inputs.

*Methods*

- `public evaluate(): list[int]`
- `public get_subspaces(): set[Subspace]`

### 3.3.9. Package backend.models.results

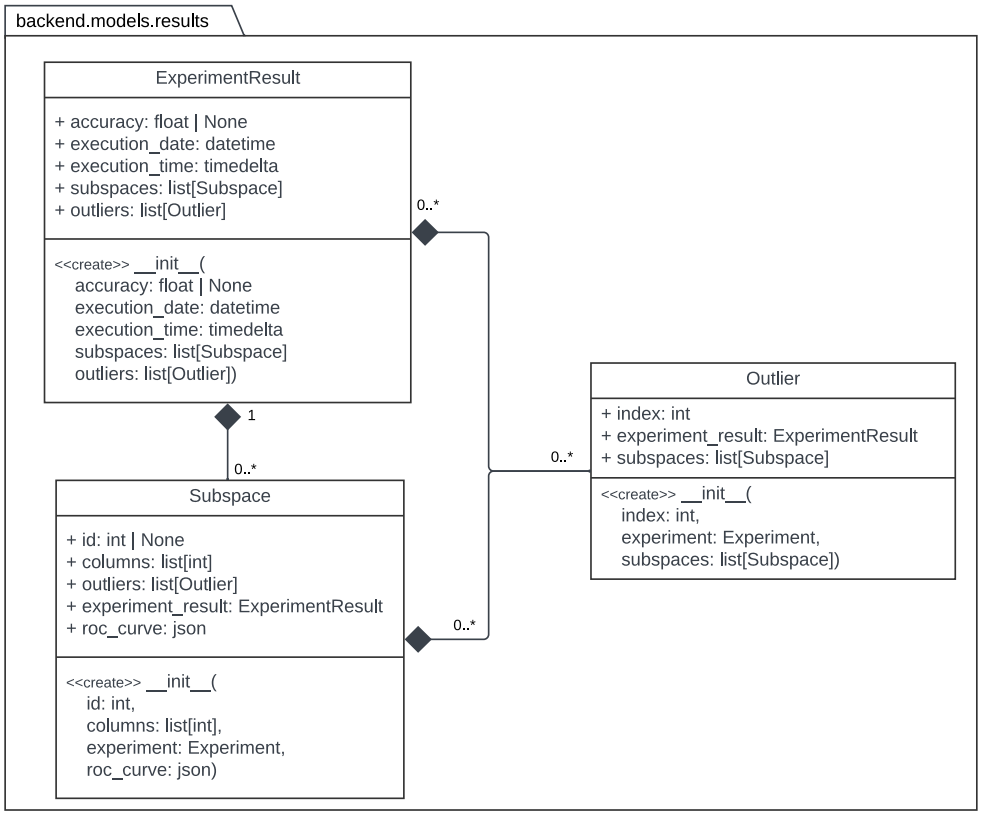


Figure 30: `ExperimentResult`, `Subspace` and `Outlier`

#### Class `ExperimentResult`

This class represents the ExperimentResult table in our database, where each row is mapped to its corresponding Python object by sqlAlchemy. It contains information about the result of the experiment.

*Attributes*

- `public accuracy: float | None`
- `public execution_date: datetime`
- `public execution_time: timedelta`
- `public subspaces: list[Subspace]`

  A list of all subspaces which have been specified by the subspace logic.

- `public outliers: list[Outlier]`

A list of data points which have been detected as outliers in any of the subspaces.

*Methods*

- `public __init__(accuracy: float | None, execution_date: datetime, execution_time: timedelta, subspaces: list[Subspace], outliers: list[Outlier], roc_curve: json): void`

### Class `Subspace`

This class represents the Subspace table in our database, where each row is mapped to its corresponding Python object by sqlAlchemy. It contains information of a subspace and outliers, which lie in this subspace.

*Attributes*

- `public id: int | None`
- `public columns: list[int]`

  A list of column numbers the subspace consists of.

- `public outliers: list[Outlier]`

  All outliers which have been found on the subspace when the experiment was run.

- `public experiment_result: ExperimentResult`

  A reference to the `ExperimentResult` this subspace belongs to.

- `public roc_curve: json`

  Saves the ROC-Curve data. Stored for each subspace, to allow for scalability.

### Class `Outlier`

This class represents the outlier table in our database, where each row is mapped to its corresponding Python object by sqlAlchemy. It contains information, about a datapoint, which is an outlier in at least one subspace.

*Attributes*

- `public index: int`
- `public experiment_result: ExperimentResult`

  A reference to the `ExperimentResult` this outlier belongs to.

- `public subspaces: list[Subspace]`

  A list of subspaces this outlier is in.datapoints

### 3.3.10. Package backend.experimentexecution

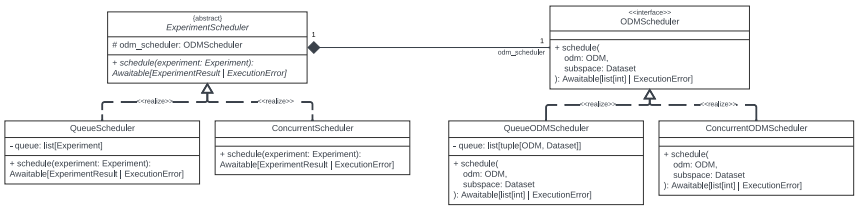


Figure 31: `ExperimentScheduler`, `QueueExperimentScheduler`, `ConcurrentExperimentScheduler`, `ODMScheduler`, `QueueODMScheduler`, `ConcurrentODMScheduler`

#### Class `abstract ExperimentScheduler`

This abstract class represents a scheduler that can schedule the execution of an experiment.

*Attributes*

- `protected odm_scheduler: ODMScheduler`

  Is used to schedule the individual ODMs. This uses the dependency injection design pattern.

*Methods*

- `public schedule(experiment: Experiment): Awaitable[ExperimentResult | ExecutionError]`

  Schedule an experiment

*Child classes* `QueueExperimentScheduler` `ConcurrentExperimentScheduler`

#### Class `QueueExperimentScheduler`

This class is a scheduler that schedules the execution of experiments in a single new thread where all incoming experiments are queued.

`extends ExperimentScheduler`

*Attributes*

- `private queue: list[Experiment]`

  The queue of experiments that need to be run

*Methods*

- `public schedule(experiment: Experiment): Awaitable[ExperimentResult | ExecutionError]`

### Class `ConcurrentExperimentScheduler`

This class is a scheduler that schedules the execution of experiments concurrently. Since concurrent execution is an optional feature the details of this class can be worked out later.

`extends ExperimentScheduler`

*Methods*

- `public schedule(experiment: Experiment): Awaitable[ExperimentResult | ExecutionError]`

### Interface `ODMScheduler`

This interface represents a scheduler that can schedule the execution of an ODM.

*Methods*

- ```
  public schedule(
      odm: ODM,
      subspace: Subspace
  ): Awaitable[list[int] | ExecutionError]
  ```

  Schedule an ODM

*Implemented by* `QueueODMScheduler` `ConcurrentODMScheduler`

### Class `QueueODMScheduler`

This class is a scheduler that schedules the execution of ODMs in a single new thread where all incoming experiments are queued.

`implements ODMScheduler`

*Attributes*

- `private queue: list[Experiment]`

  The queue of ODMs that need to be run

*Methods*
  ```

- `public schedule(`
  `odm: ODM,`
  `subspace: Subspace`
  `): Awaitable[list[int] | ExecutionError]`

  Schedule an ODM

### Class `ConcurrentODMScheduler`

This class is a scheduler that schedules the execution of ODMs concurrently. Since concurrent execution is an optional feature the details of this class can be worked out later.

`implements ODMScheduler`

*Methods*

- `public schedule(`
  `odm: ODM,`
  `subspace: Subspace`
  `): Awaitable[list[int] | ExecutionError]`

  Schedule an ODM

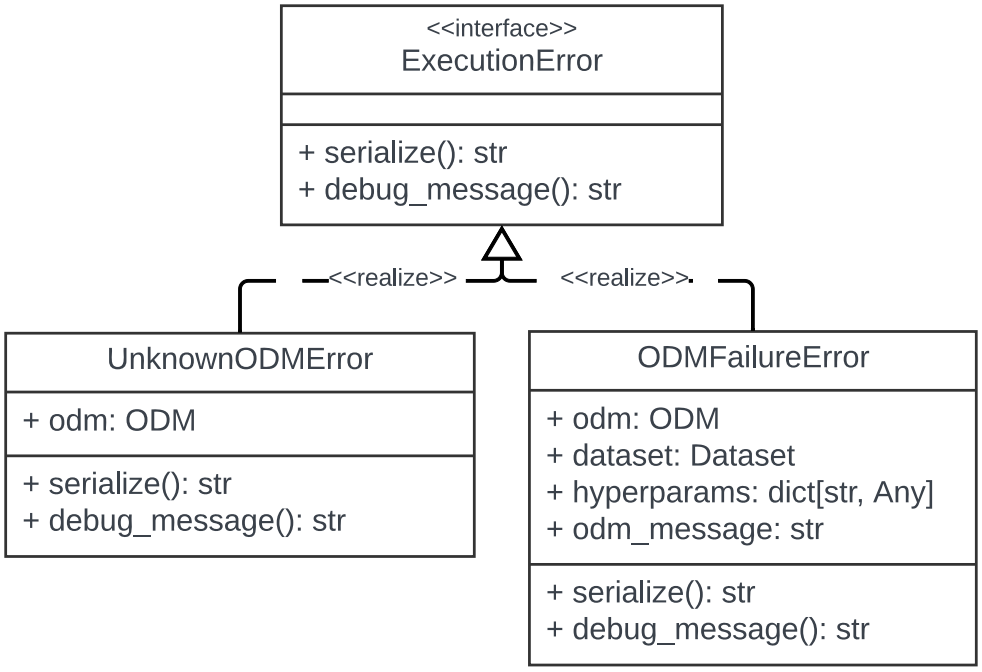


Figure 32: `ExecutionError`, `UnknownODMError` and `ODMFailureError`

#### Interface `ExecutionError`

This class describes an error that can occur during the execution of an experiment.

*Methods*

- `public serialize(): str`

  Returns a JSON representation of the error that can be sent to the frontend

- `public debug_message(): str`

  Returns a readable string representation that is meant to be displayed only for debugging purposes

*Implemented by* `UnknownODMError` `ODMFailureError`

#### Class `UnknownODMError`

This class describes the error case that the ODM is unknown to its `ODMRunner`. This error indicates a programming or database setup error and should not occur under normal conditions.

`implements ExecutionError`

*Attributes*

- `public odm: ODM`

  The `ODM` that was unknown to its `ODMRunner`

*Methods*

- `public serialize(): str`
- `public debug_message(): str`

### Class `ODMFailureError`

This class describes the error case that the ODM failed internally. This error cannot be fixed by the application and should be displayed to the user.

`implements ExecutionError`

*Attributes*

- `public odm: ODM`

  The `ODM` that was used

- `public dataset: Dataset`

  The `Dataset` that was used

- `public hyperparams: dict[str, Any]`

  The hyperparameters that were used

- `public odm_message: str`

  A message containing a readable representation of the internal ODM error

*Methods*

- `public serialize(): str`
- `public debug_message(): str`

### 3.3.11. Package backend.odmrunner

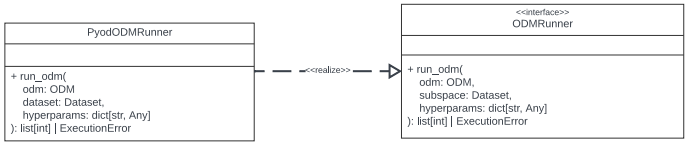


Figure 33: `PyODRunner` and `ODMRunner`

#### Interface `ODMRunner`

This interface provides the functionality to run an `ODM`. It represents an adapter pattern, which makes it possible to run ODMs from many different sources. An instance of this interface must not be able to run all `ODM`.

*Methods*

- ```
  public run_odm(
      odm: ODM,
      dataset: Dataset,
      hyperparams: dict[str, Any]
  ): list[int] | ExecutionError
  ```

  Run the ODM. At index $i$ in the returned list is a `1` if the $i$th datapoint of the dataset was identified as an outlier and a `0` if not. If the `ODM` cannot be run by this instance of `ODMRunner` the method returns `UnknownODMError`

*Implemented by* `PyODRunner`

#### Class `PyODRunner`

This class provides the functionality to run ODMs from the PyOD Python module.

`implements ODMRunner`

*Methods*

- ```
  public run_odm(
      odm: ODM,
      dataset: Dataset,
      hyperparams: dict[str, Any]
  ): list[int] | ExecutionError
  ```

  Run the ODM. `PyODRunner` can only run `ODM` of type `ODMType.PYOD`.

### 3.3.12. Package backend.odmprovider

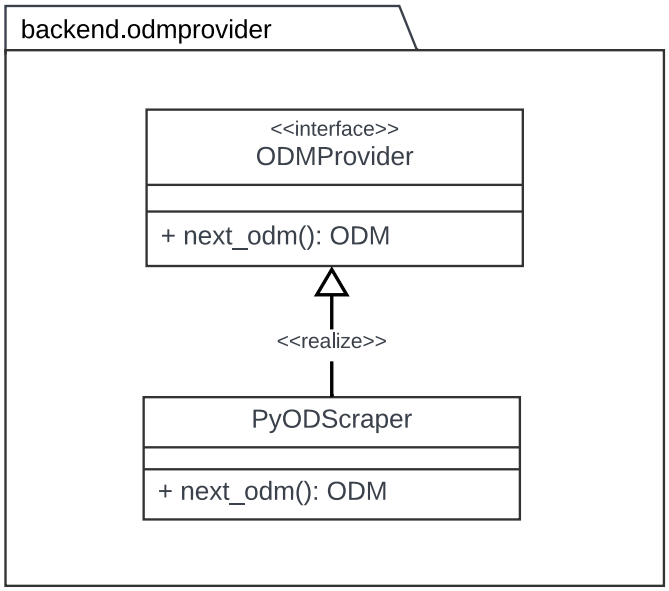


Figure 34: `ODMProvider` and `PyODScraper`

#### Interface `ODMProvider`

This interface provides the functionality to provide `ODM` instances.

*Methods*

- `public next_odm(): ODM`

  Return the next `ODM`. This method should be implemented as a coroutine that yields one `ODM` at a time.

*Implemented by* `PyODScraper`

#### Class `PyODScraper`

This class provides the functionality to scrape the PyOD Python module to retrieve the `ODM` instances that PyOD provides.

`implements ODMProvider`

*Methods*

- `public next_odm(): ODM`

### 3.3.13. Package backend.parsers

#### Class `DatasetCSVParser`

This class provides a method to parse a dataset.

*Methods*

- `public static parse(dataset: str): Dataset`

#### Class `GroundTrtuhCSVParser`

This class provides a method to parse a ground truth file.

*Methods*

- `public static parse(ground_truth: str): list[int]`

#### Class `OutlierCSVEncoder`

This class provides a method to encode an outlier list.

*Methods*

- `public static encode(outliers: list[int]): str`

# 4. Routing/URL overview

## 4.1. Front-end

Routing on the front-end side is implemented using the Vue router, which allows the definition of routes, which are key-value dictionaries containing (at the minimum) a path key and a component key. This way depending on the url of the browser Vue knows which component should be loaded. The passed component is rendered, when <RouterView /> is inserted into the template of a vue page/view. Navigation guards can also be set up for the router. For example, a beforeEach guard can be registered, which as the name implies, is run before each redirection. This way, an unauthenticated user can be redirected to login page when trying to access the dashboard page via "/dashboard". For this routes can be provided with meta fields such as "requiresAuth:true" to indicate that the route shall only be accessible, when the user is authenticated.

### URL overview:

- /home ⇒ HomeView loaded
- /login ⇒ LoginView loaded
- /register ⇒ RegisterView loaded

### Navigation guards:

.beforeEach:

- check whether "/login" or "/register" are attempted to be accessed by authenticated users and if that is the case redirect them to their dashboard
- redirect users to the login page if a route with the meta field "requiresAuth:true" is attempted to be accessed
- in the case that the "/experiment-result/..." is attempted to be accessed first check whether an experiment with the provided experiment id exists.

## 4.2. Back-end

With routing on the back-end side, the API calls/endpoints are meant, which are handled using Flask. Since some API calls such as the "/dashboard" api call only make sense, when a user is logged in they are only handled when a valid access token is passed in the HTTP request's header. For this, a python wrapper function"@jwt-required()" is used. All routes prefix "/api".

**API endpoints:**

User API:

- /user/login: Returns an access token if the username and password are correct.
- /user/register: If valid register data was passed a user account is created and an access token is returned.
- /user/logout: The user's access token is invalidated. Requires an access token.

Experiment API:

- /experiment/validate-dataset: Checks whether the specified dataset is valid. Requires an access token.
- /experiment/validate-ground-truth: Checks whether the ground truth file is valid. Requires an access token.
- /experiment/get-result/<int:exp-id>: Returns the result of an experiment if an experiment with id <exp-id> exists and has finished executing. Requires an access token.
- /experiment/get-all: Returns a list of all the experiments the user has run. Requires an access token.
- /experiment/create: Creates and runs a new experiment. Requires an access token.
- /experiment/download-result/<int:exp-id>: Checks if the user has an experiment with the specified <exp-id>. Downloads the outliers with the applied subspace logic as a CSV file. Requires an access token.

ODM API:

- /odm/get-all: Returns all of the availabe odms. Requires an access token.
- /odm/get-parameters/<int:odm-id>: Returns a list of all parameters the odm with id <odm-id> has including their type and whether they are optional. Requires an access token.

# 5. Authentication

Authentication is necessary in "Outlr." as user-specific data such as created experiments of a user are stored persistently and are shown on the dashboard page of the user. For this Outlr. implements token-based authentication using JSON web tokens (JWT). To ensure security a token expires after a certain duration.

How the user can retrieve such a token is depicted in the following diagram:

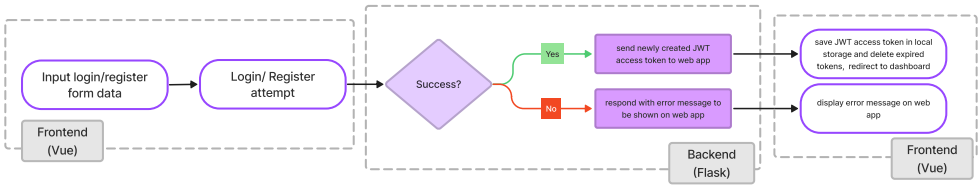


Figure 35: When a user successfully logs in or registers, the API responds with an access token, which is then saved in the local storage of the users browser.

### Front-end implementation:

As the display of the web app depends on whether a user is logged in, it must store the current authentication state of the user and switch the state when necessary. This will be implemented using vuex. When in authenticated state, the users access token and username are stored in local storage. When a token expires it is deleted from local storage and the state is changed to unauthenticated.

To prevent a user from navigating routes which should only be accessed when in a certain state, navigation guards are defined for the Vue router.

When accessing protected API endpoints, the token is read from local storage and passed in the authorization header of the http request.

### Back-end implementation

All API endpoints are protected except "/status", "/user/register" and "/user/login". When no valid access token is passed in the authentication header of the http request the protected endpoints deny the API call. If the access token is valid the user can be read from the token and the request can be handled in user context e.g. creating an experiment for the user or retrieving all experiments the user has run.

# 6. Sequence diagrams

## User Management:

**Initial state:** The user has entered his register data in the register form and pressed the submit button.

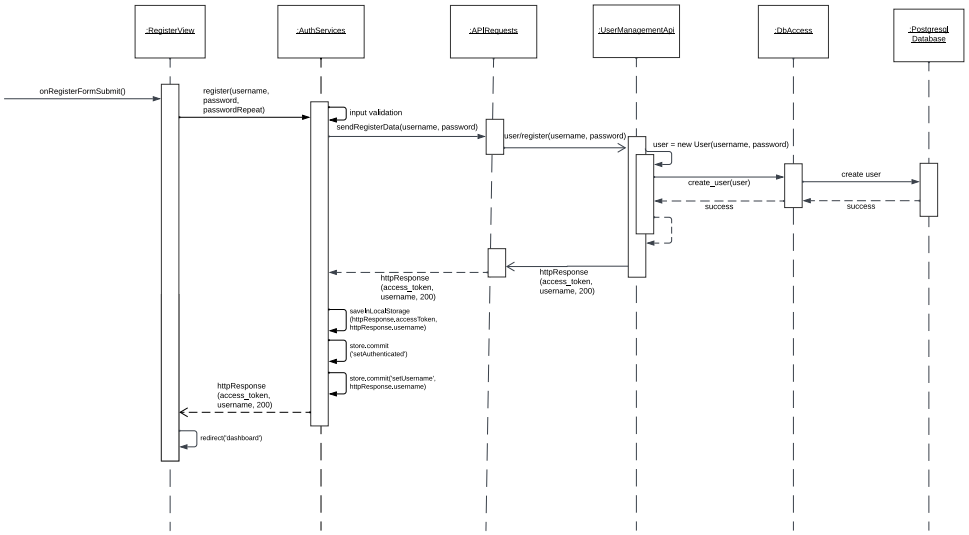


Figure 36: Creating an account when valid data is provided

**Initial state:** The user has entered his register data and entered a username, which is already taken by a different user in the register form and pressed the submit button.

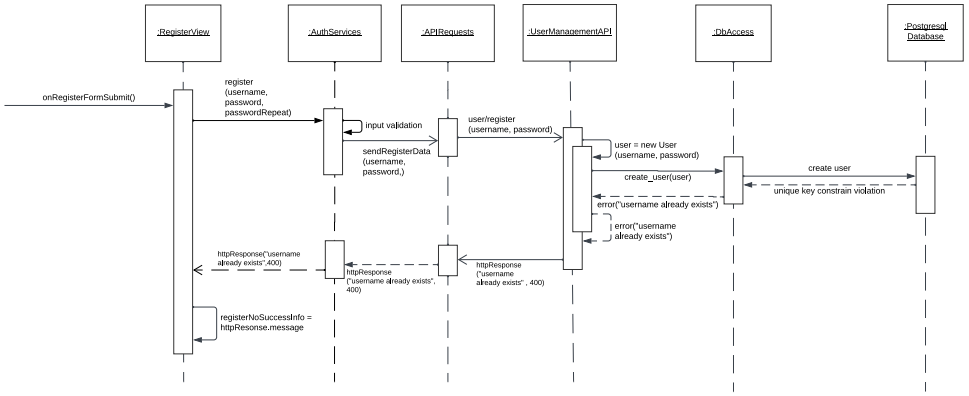


Figure 37: Trying to register with username, which is already taken

**Initial state:** The user has entered valid login data and pressed the submit button.

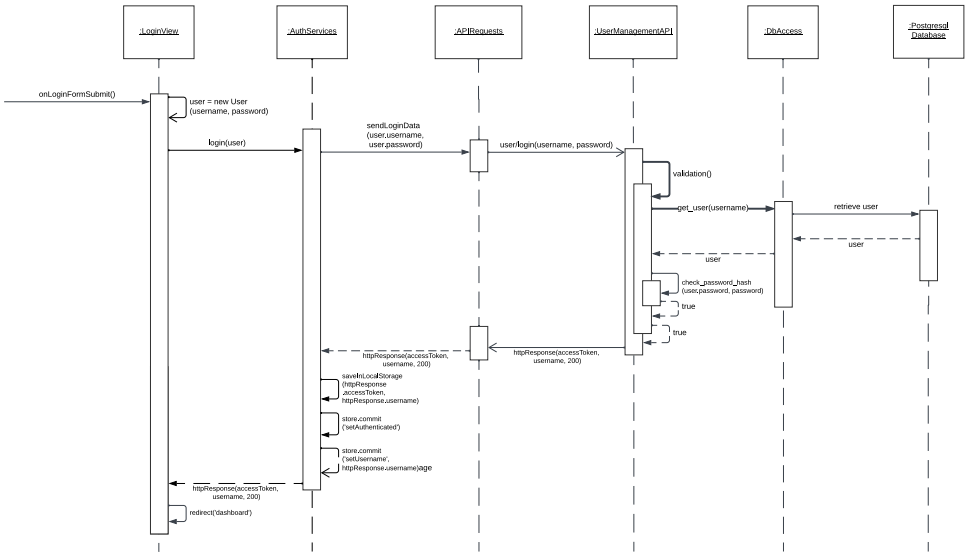


Figure 38: Logging in with valid data

**Initial state:** The user entered a wrong password in the form and pressed the submit button.

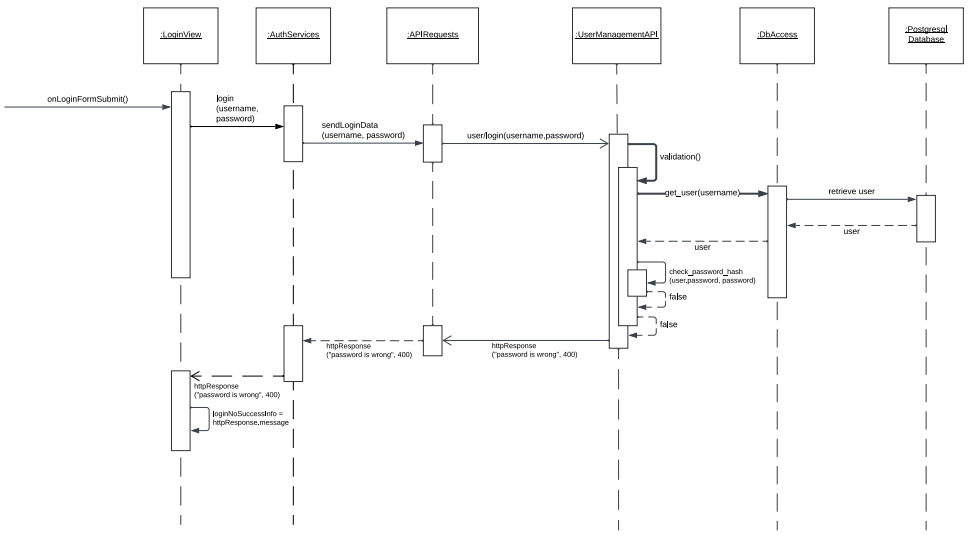


Figure 39: Trying to log in although provided password is wrong

**Initial state:** The user entered the logout button from the navigation bar.

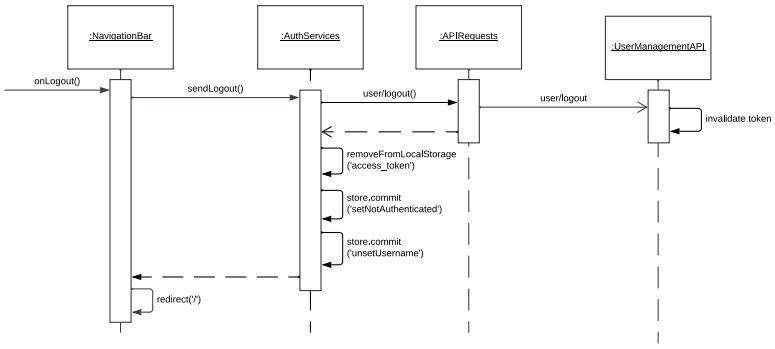


Figure 40: Logging out when logged in. This only works when the user is logged in, which is why the back-end checks whether an access token is sent with request, therefore the "@jwt-required"

## Dashboard:

**Initial state:** The user has clicked on navigate to dashboard and gets redirected to the `DashboardView`

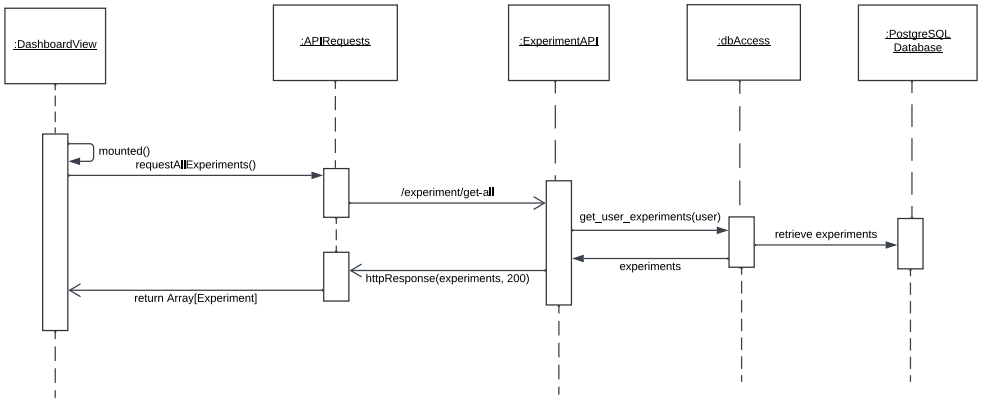


Figure 41: When the `DashboardView` is mounted to the DOM it requests all experiments the user has created from the backend and shows the data in the `DashboardTable`

**Initial state:** The user is on the Dashboard.

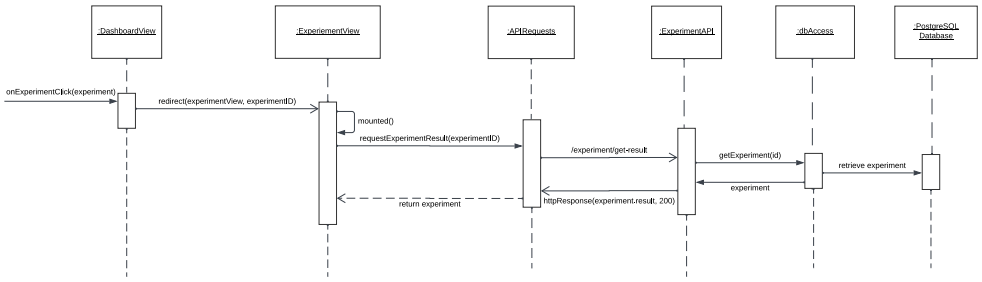


Figure 42: The user clicks on an experiment in the dashboard table. The user then gets redirected to the `ExperimentResultView`. The experiment id gets also passed to the `ExperimentResultView`. The `ExperimentResultView` then requests the experiment from the backend by id and shows the user the data

## Experiment creation:

**Initial state:** The user is on the `CreateExperimentView`

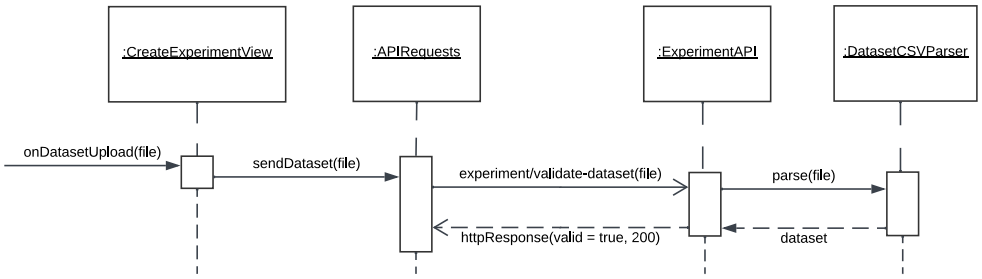


Figure 43: The user trys to upload a valid dataset.

**Initial state:** The user is on the `CreateExperimentView`

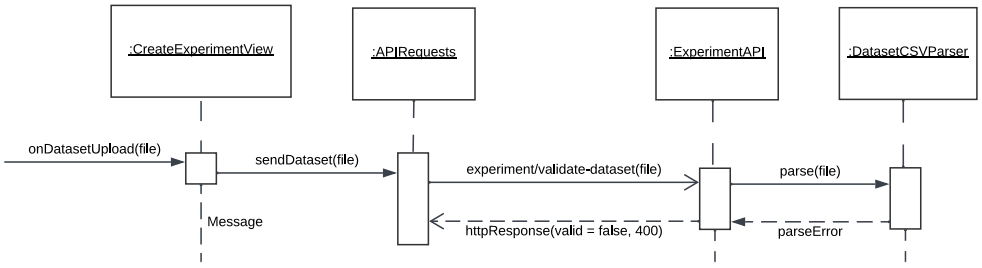


Figure 44: The user trys to upload a invalid dataset.

**Initial state:** The user is on the `CreateExperimentView`

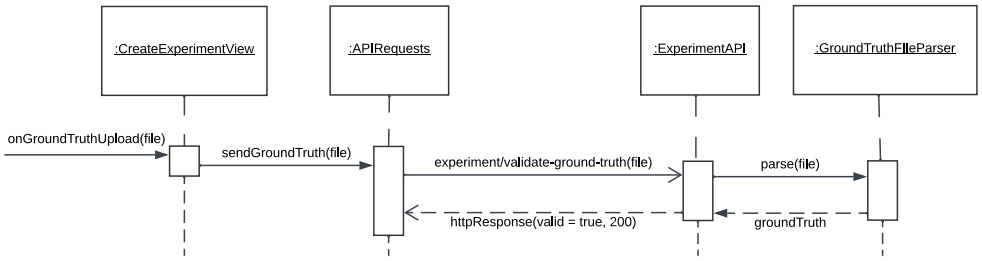


Figure 45: The user trys to upload a valid ground truth file.

**Initial state:** The user is on the `CreateExperimentView`

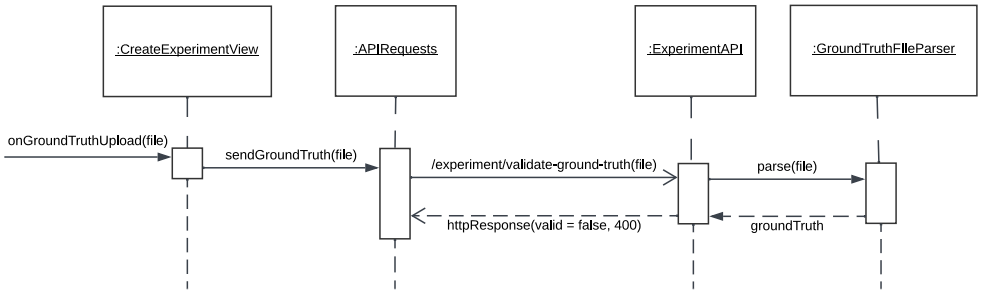


Figure 46: The user trys to upload a invalid ground truth file.

**Initial state:** The user is on the `CreateExperimentView`

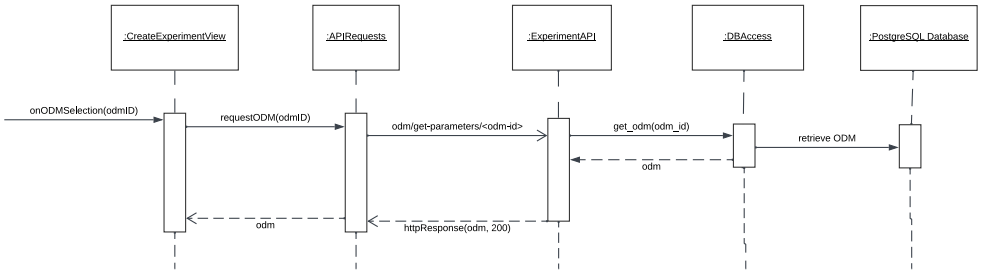


Figure 47: The user selects and odm from a dropdown menu.

**Initial state:** The user is on the `CreateExperimentView`

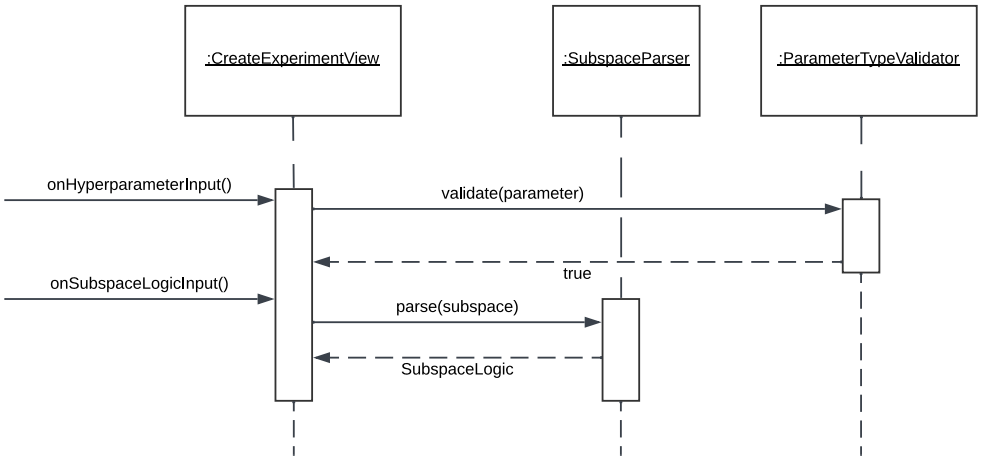


Figure 48: The user inserts correct hyperparameters and correct subspacelogic.

**Initial state:** The user is on the `CreateExperimentView`

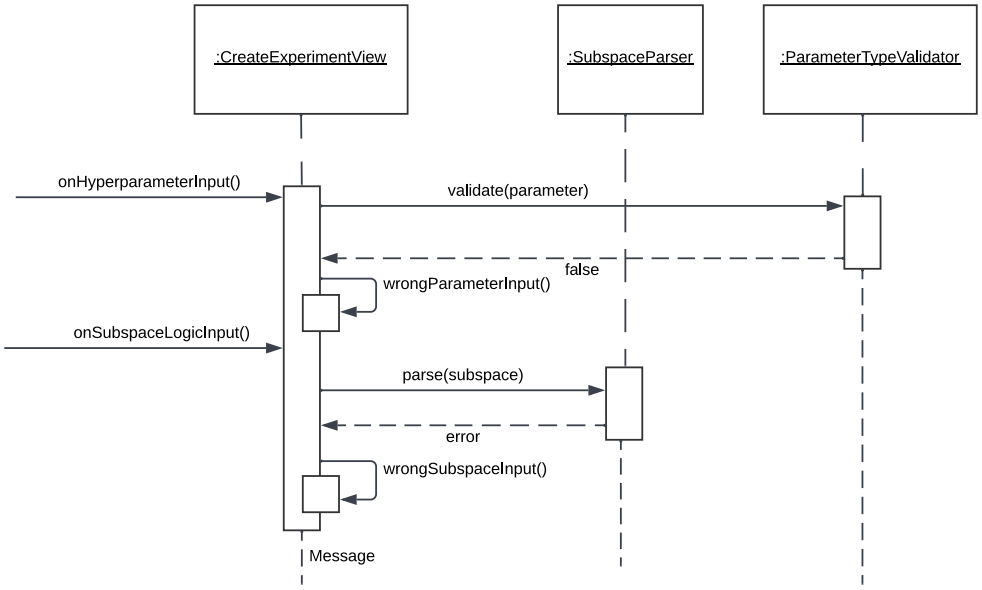


Figure 49: The user inserts incorrect hyperparameters or incorrect subspacelogic.

**Initial state:** The user is on the `CreateExperimentView`

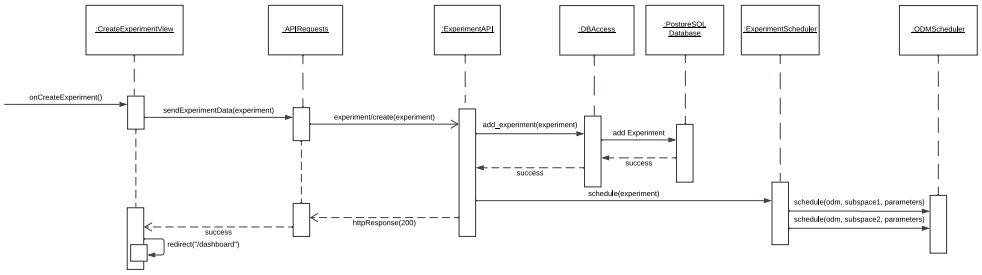


Figure 50: The user provide all data required for an experiment run.

**Initial state:** The user is on the `CreateExperimentView`

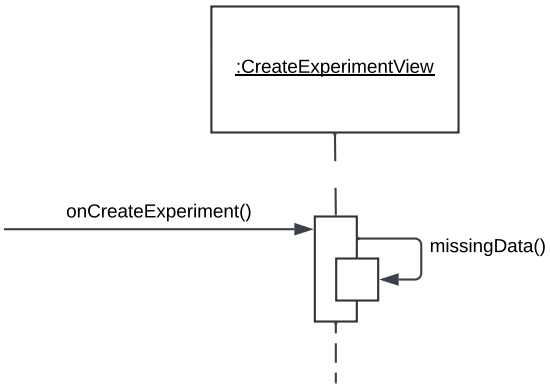


Figure 51: There is still required data missing.

# 7. Database

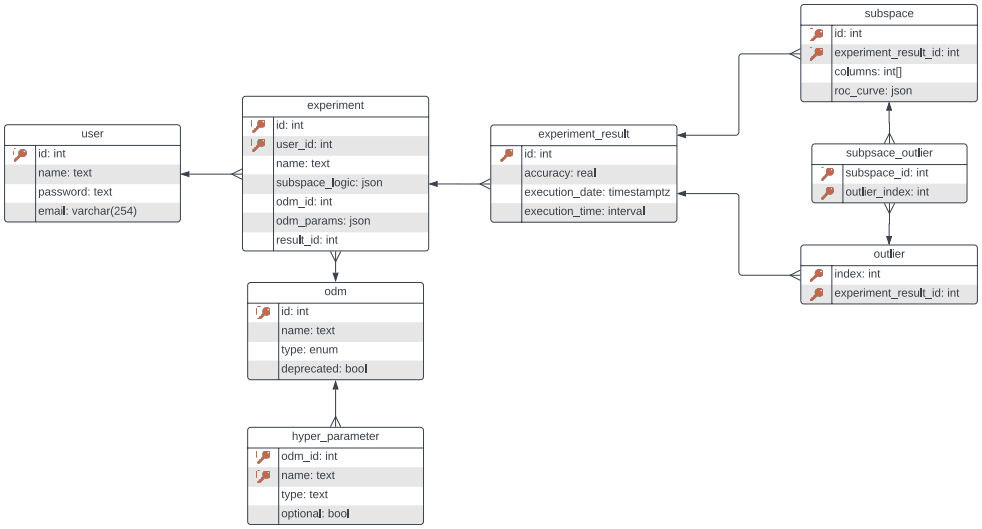


Figure 52: ER Diagram

# Appendices

## Glossary

**Experiment** A configurable execution of outlier detection methods on subspaces of a dataset. 70, 71, 73, 89

**Flask** Python web framework. 6, 12, 55

**Ground-truth file** CSV file containing which points of a dataset are actually outliers. 4

**Hyperparameter** Parameters that an ODM requires. 74

**numpy** Fundamental package for array computing in Python. 6

**Object-Relational Mapping** ORM (Object-Relational Mapping) is a programming technique that maps relational dactabases to objects in an object-oriented language. It provides an abstraction layer between the database and the application, and provide a set of tools to interact with databases using an object-oriented API.. 11, 89

**pandas** Python library used for data analysis. 6

**PostgreSQL** A database system. 6

**PyOD** Python library providing methods for outlier detection. 6, 55, 65, 75, 76

**Python** Programming language commonly used in the field of machine learning and outlier detection. 6, 75, 76, 89

**SQLAlchemy** Python library for working with relational databases. It provides a set of high-level API for interacting with databases in an object-oriented way, as well as a powerful ORM (Object-Relational Mapping) system for mapping Python objects to database tables.. 6, 11, 55

**Subspace** A subspace of the complete dataset. 20, 89

**Subspace logic** A logic that specifies in which subspaces a point of the dataset must be identified as an outlier in order to be contained in the experiment result. 19, 20, 25, 52, 66, 67

**TypeScript** Programming language that can be used for web development. 6, 13

**Vue.js** Front-end framework that can be used for building websites. 6, 13

## Acronyms

**CSS** Cascading Style Sheets, a language used to describe the appearance of a website. 6, 28

**CSV** Comma-separated values, a file format commonly used for storing datasets. 89

**HTML** HyperText Markup Language, a language used to describe the structure of a website. 6

**JSON** JavaScript Object Notation. 18, 25, 73

**ODM** Outlier detection method. 47, 49, 55, 64, 65, 70–72, 74, 75, 89

**ROC-Curve** Reciever operating characteristic curve, plots the true positive rate against the false positive rate. 54, 69

# A. Complete class diagram

## Frontend

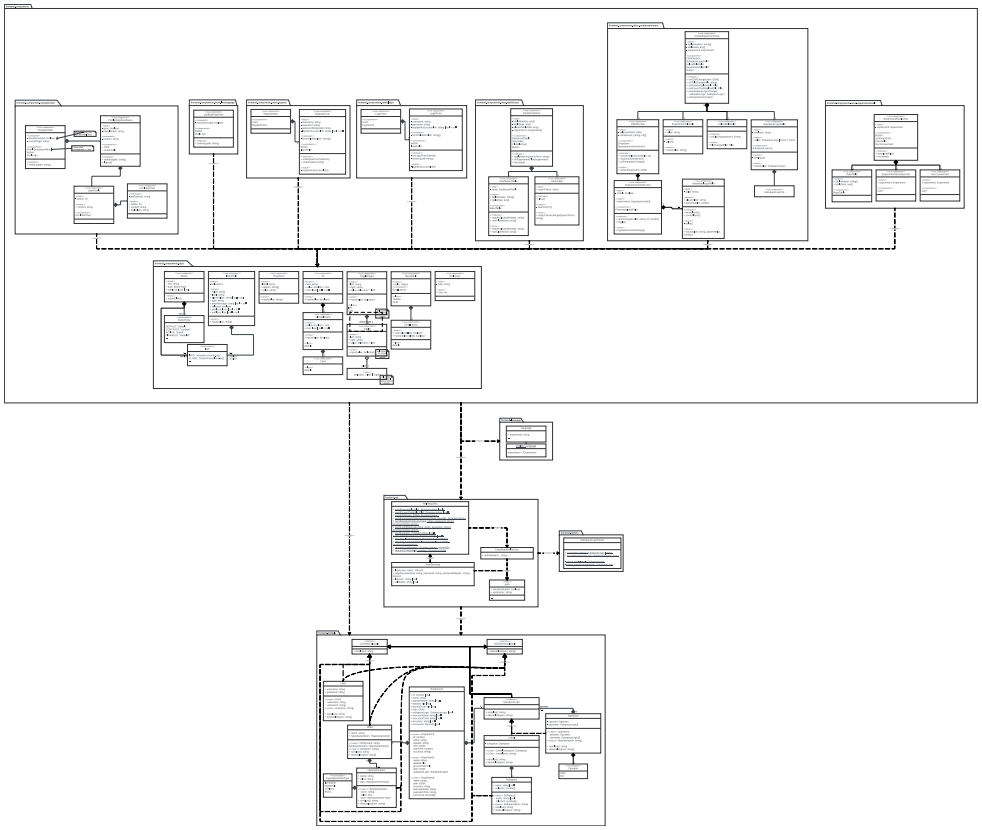

Figure 53: Complete frontend class and package diagram

## Backend

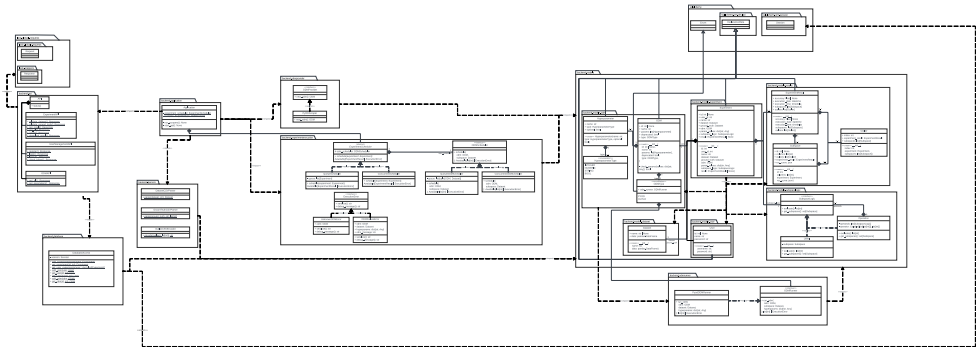

Figure 54: Complete backend class and package diagram