Zip Manufacturing Training & Certification Manager

Object Oriented Design Document

Author: Roni Diwan

zipline

Table of content

About the project

This program is designed to manage all training and certifications for the zip manufacturing team in Zipline. With the company rapidly growing, and FAA strict training requirements there is a need to track all team members qualifications. This app is designed for ziplines current manufacturing team, but with an eye to the future, as the company and the team grow.

Technologies used in this project

- Python using Django framework
- ORM - Django REST
  - Data is exported to a file in sqlite3.
- React JS
- Bootstrap framework
- React main Packages:
  - Axios for communication with API
  - Redux toolkit for state management

Writing Standards

The project follows the standard style guides for the languages used.

Python – PEP 8 style guide: https://www.python.org/dev/peps/pep-0008/

JavaScript – Google JS style guide: https://google.github.io/styleguide/jsguide.html

Assumptions

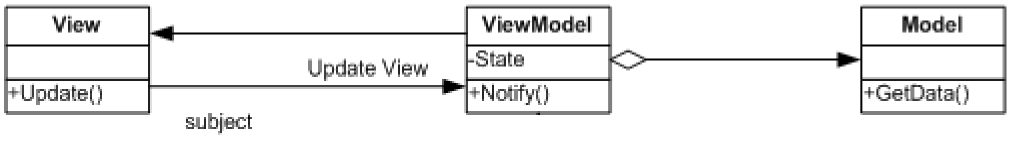
- A new user can only be created by an admin, there is no self-registration. The admin will create an initial password, the user can change this password once logged in.
- If a user forgot the password, the admin could change that password for this user, but at no point an admin can view another user's password.
- A user is identified by an email. This property must be unique and is not changeable.
- All training files are source from google drive, since this is the file management system the company currently uses.

Project Architecture

The project will be built using MVVM architecture:

1. Model – controls all interactions with the data base and business logic.
2. View – controls the UI.
3. View Model– controls the interaction between the view and the model.

MVVM promises a level of separation between the view and the model, making them independent and there for the design of the system flexible.



MVVM architecture UML

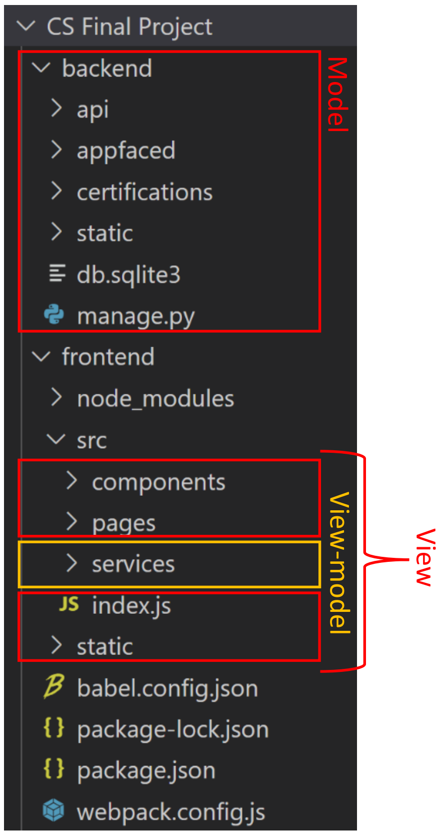
<u>Files and apps:</u>

<u>Backend</u>

1. **Model** – the model is represented by the backend folder. This includes the model mapping and business logic in the models and signals files, and the API logic and views under the 'serializers' file and the 'views' folder.

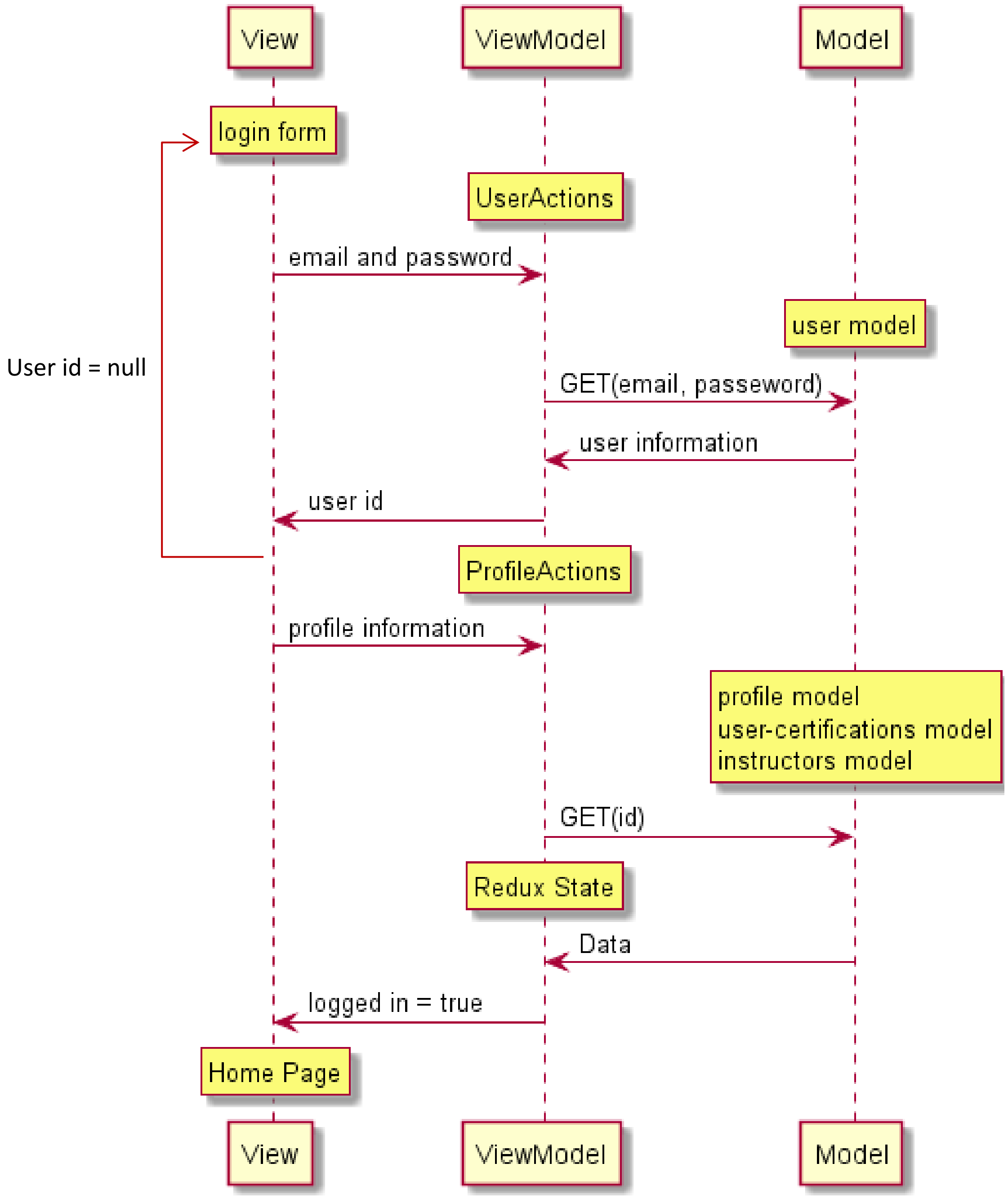
<u>Frontend</u>

1. **View** – this part of the program is managed in the frontend file and includes all pages, components and static files.
2. **View Model** – The view model is represented by the 'services' folder. It includes state management services using Redux, and API services. Each page or component in the app has a reference to the services it needs and uses it to send and receive data.

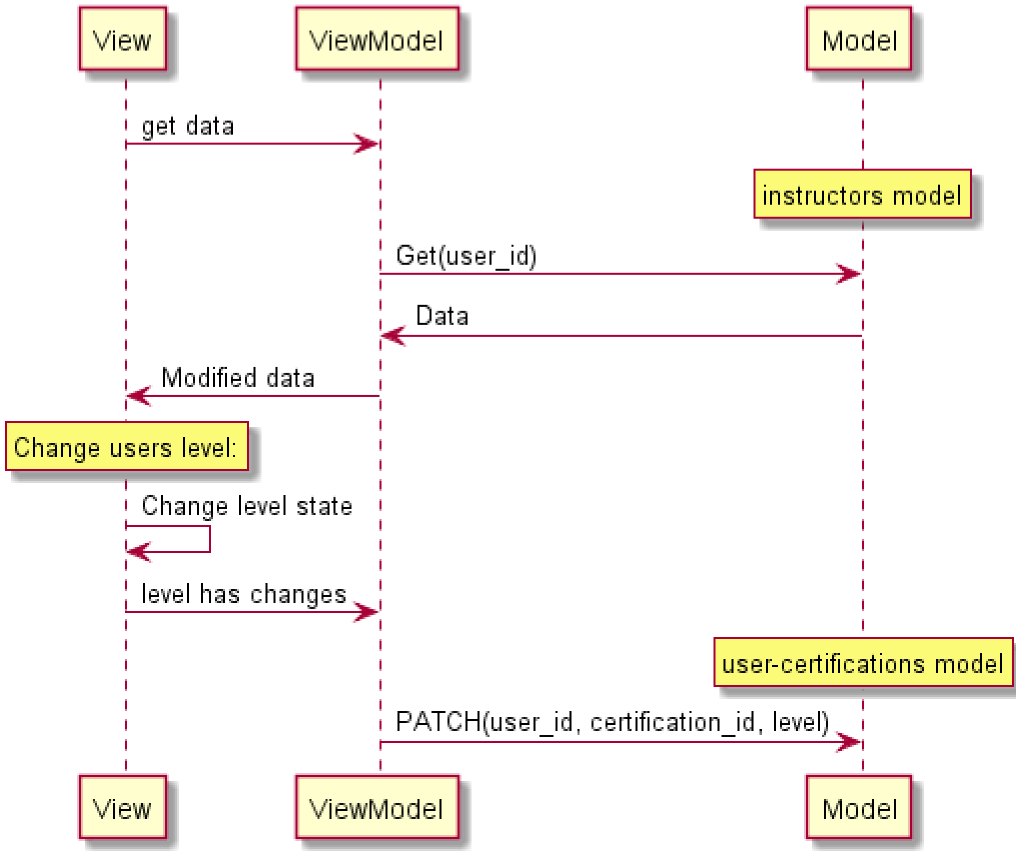


Main Processes

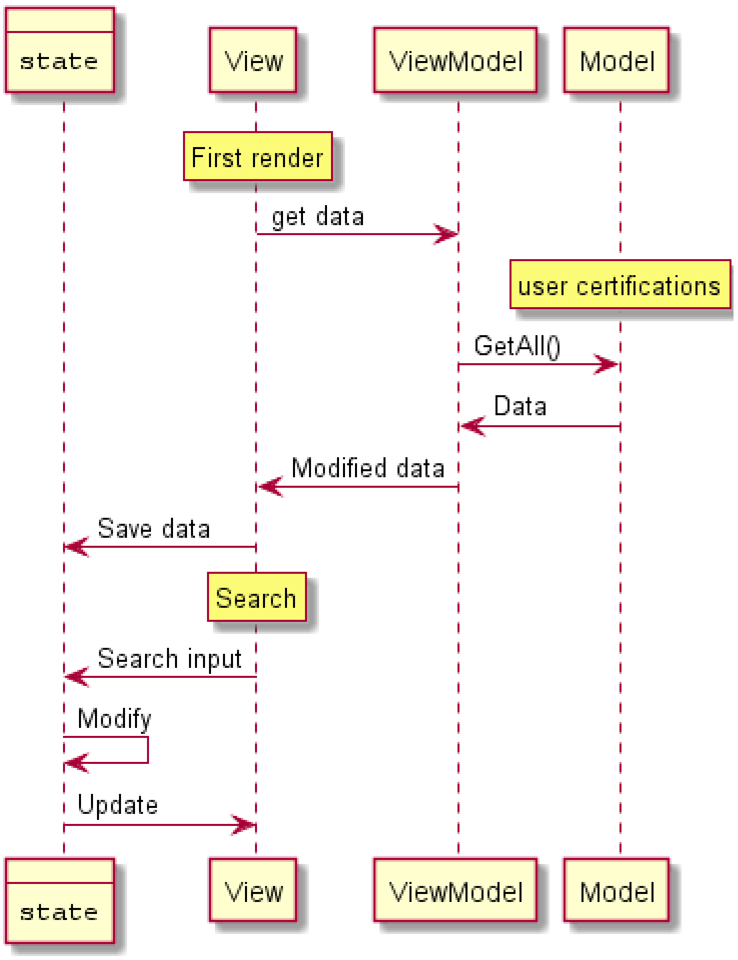
Login Sequence diagram



Train (Practical) Sequence diagram



Search My Team Table Sequence Diagram



Future Changes

As the team grows, the application might need to be modified. Here are some examples of possible changes and their implementation:

1. **Adding more scales**
   Even through the requirement was to implement two scales, I chose to create a database entry for this information instead of hard coding it.
   When a new scale will be added, it and its levels will need to be entered in the data base. This is currently only accessed by adding information directly to the API but can be easily modified to be a feature in the application. The next step would be to modify the functionality for the level attribute under the user certification table, to match the new required logic.

2. **Trainer can add fail status to a training**
   In this case a trainer would mark a specifics users training as "Fail", and the trainee will need to re-do the training the pass. To implement this change, on the Train page, there will be a component similar to the practical training component. This will be a list view of trainings will all of the trainees and their level. The trainer will have the permission to change a user's level.
   This could be implemented with a mix of trainings that have an auto pass, and their level is not editable (the current implementation), and trainings with an editable level. To make this change a Boolean field will be added to the training database table.

Backend

This part of the project represents the model in the MVVM design pattern and includes the data base, business logic and functionality for reading and writing data.

Data Base

Data Base Models Tables:

User *Each instance represents a single user*			
Name	**Type**	**Key**	**Explanation**
Id	Int	Primary Key	
Username	String		Equals email. Mandatory field in Django User model.
Email	Email		
password	String		

Profile *Each instance represents a single profile. A Profile is an extension of a user.*			
Name	**Type**	**Key**	**Explanation**
User id	Int	Primary + Foreign key	Same id as user id.
User name	String		The users name
Is admin	Boolean		True if user is an admin.
Role	String		
image	Image		

Certification *Each instance represents a single certification*			
Name	**Type**	**Key**	**Explanation**
Id	Int	Primary Key	
Name	String		Certification name
Practical	Boolean		True if a practical certification, False if theoretical
Level scale	String	Foreign key	Scale used to measure certification level
Days valid	Int		For how many days is this certification valid from the time associated to a user.

Training Module *Each instance represents a single Training module.*			
Name	**Type**	**Key**	**Explanation**
Id	Int	Primary Key	
Certification id	Int	Foreign key	The certification this model is associated to.
Name			
Link			Link to training file (Google Drive)

Certification Scales *Each instance represents a single Scale*			
Name	**Type**	**Key**	**Explanation**
Id	Int	Primary Key	
Scale name	String		

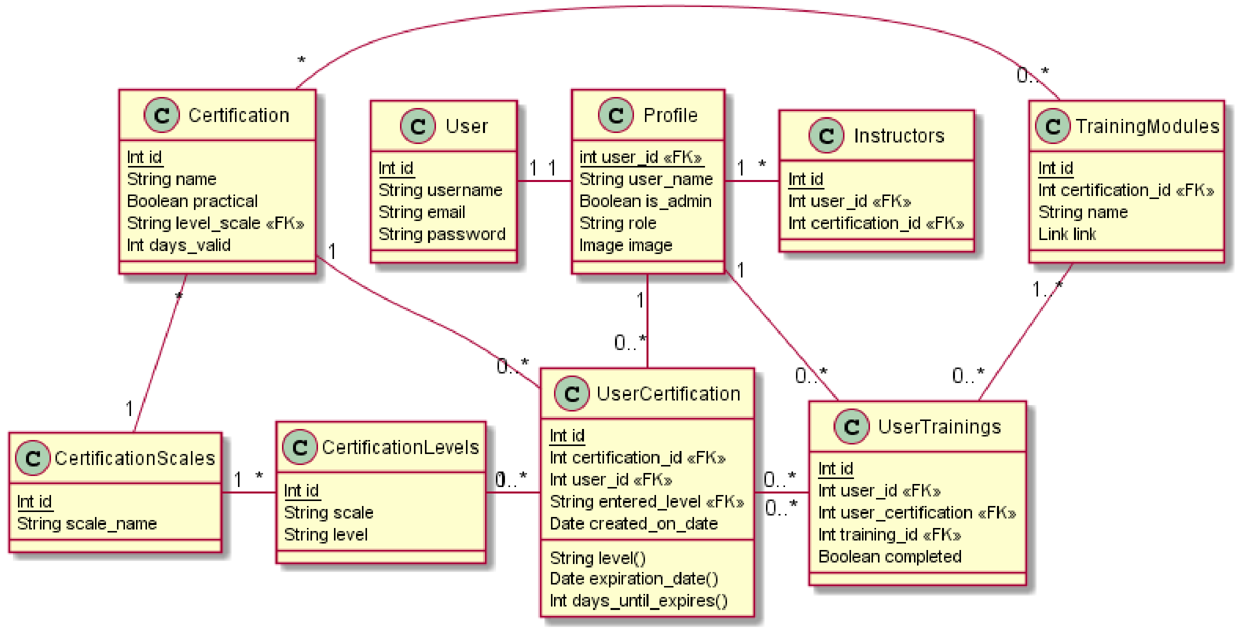
Certification Levels *Each instance represents a single level of a scale.*			
Name	**Type**	**Key**	**Explanation**
Id	Int	Primary Key	
Scale	String	Foreign key	The scale this level is associated to
Level	String		Levels name

Instructors *Each instance represents a single Instructor for a specific certification*			
Name	**Type**	**Key**	**Explanation**
Id	Int	Primary Key	
User id	Int	Foreign key	Id of user defined as an instructor
Certification id	Int	Foreign key	Id of certification the user is an instructor for

User Certifications *Each instance represents a single certification associated to a user.*			
Name	**Type**	**Key**	**Explanation**
Id	Int	Primary Key	
Certification id	Int	Foreign key	Certification associated to user
User id	Int	Foreign key	
Entered level	String		User input level -> if doesn't match the certifications scale, will be overwritten in Level attribute
Created on	Date		Date this instance was created
Level	String		Actual attribute for reading. Will override illegal entered level or automate Pending-Pass status for certifications with a pass/fail scale.
Expiration date	Date		Created on date + certifications days valid
Days until expires	Int		Days from today until expiration date.

User Trainings *Each instance represents a single training associated to a user*			
Name	**Type**	**Key**	**Explanation**
Id	Int	Primary Key	
User id	Int	Foreign key	User with this training
User certification	Int	Foreign key	User-certification that generated this instance.
Training id	Int	Foreign key	Training associated to user.
Completed	Boolean		True if user completed the training.

Data Base Relations Diagram:



The Models in the Data Base have a high dependency on one another, some will be created automatically when others are created. All instances of a model will be deleted if one of their foreign keys is deleted.

Models with multiple foreign keys (Instructors, user Certifications and User Training) require a unique combination of those keys.

User Certification is unique because it has 3 fields that are automatically calculated (level, expiration date and days until expires). This promises accuracy of data, and it being updated when an attribute of a foreign key is updated.

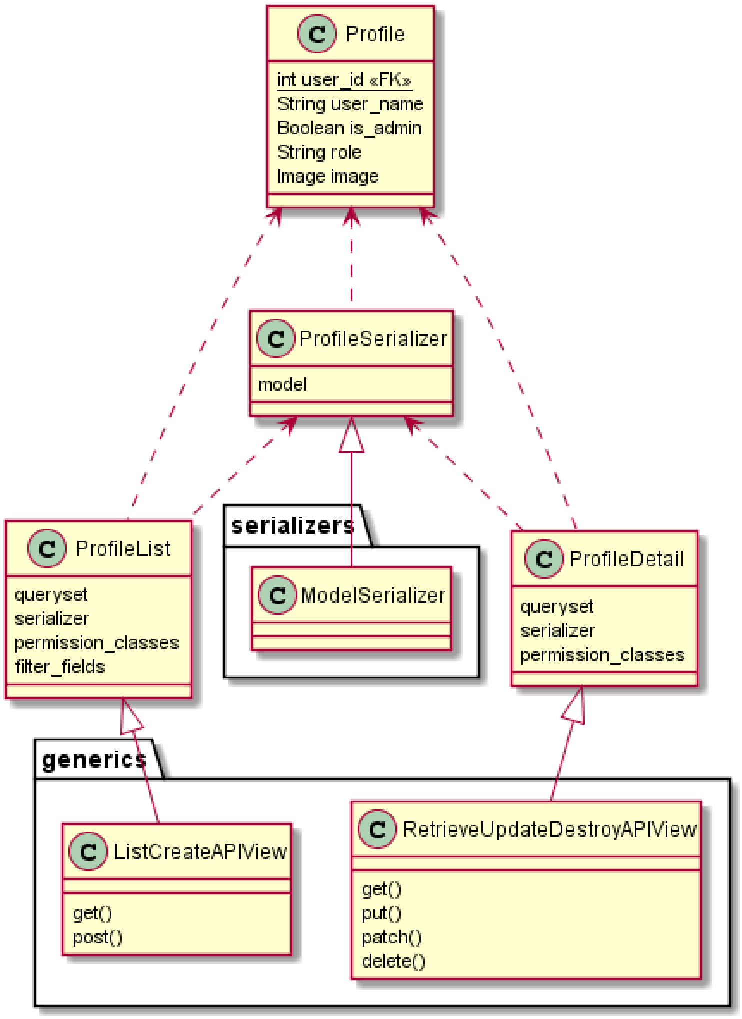
Model to API

Each model in the date base is serialized to JSON format using a serializer class. Each model is represented by 2 views:

1. A List view for reading all data in the model and creating a new instance
2. A Detail view for actions on a specific instance of the model. This view allows reading, updating and deleting the instance.

Then an API endpoint (URL path) is created for each of the views.

Example of Profile model to API:



Design patterns

<u>Observer Pattern:</u>

Some models in the data base have a high dependency and should be created as a result of the creation of their dependency. To assure this happens, I implanted the observer design pattern in these classes using Django's built-in observer: Signals.

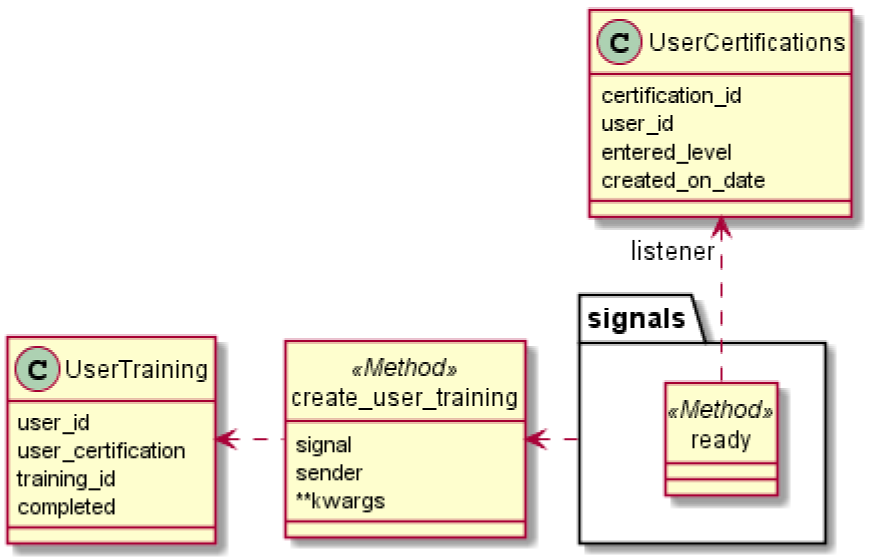
Once an instance of a class is created, a signal is sent to the dependent class (receiver), if that class doesn't have an appropriate object a new one will be created.

Signals are sent from the *User* class to the *Profile* class, and from *UserCertification* to the *UserTraining* class:

1. *Profile* is an extension of *User*, when a user is created an empty profile will be created using the same *id* from *User* as the *user_id* attribute in *Profile*.
2. Some certifications have associated trainings. When a new *UserCertification* is created, a *UserTraining* will be initialized for each training under that certification using *user_id*, *user_certification_id* and *training_id*.

<u>Signal example - UserTraining signal:</u>

Structure:



Code snippet:

```
    def ready(self):
```

```
        from . import signals
```

The *ready()* method is defined in the *apps.py* file. This method “creates the listener” by connecting the receiver to the signal.

```
@receiver(post_save, sender=UserCertifications)
def create_user_training(sender, **kwargs):
    if kwargs.get('created', False):
        for training in
(Certification.objects.filter(id=kwargs.get('instance').certification_id_i
d).values("trainings")):
            if (not training.get("trainings")):
                break

            UserTraining.objects.get_or_create(
                user_certification=kwargs.get('instance'),
                user_id=kwargs.get('instance').user_id,
                training_id=TrainingModule.objects.filter(id=training.get(
'trainings'))[0])
```

In the *Signals.py* file, the receiver decorator defines that the function *create_user_training* will run as a result of the signal *post_save* sent by the *UserCertification* class. This means that every time the *save()* function in the UserCertification class is complete This function will be called.

Then the code will look at certifications with the same id as the senders, and check if the trainings that are associated with the certification exist. If not – a new user training will be created using the information provided by the sender.

Frontend

The Front end is implemented in React-JS and includes the view and view-model of MVVM architecture. The view model uses class base components, to improve design and help keep the code clean. The view is implemented using functional based components, which is the current standard. State is passed between the component using either hierarchy or redux state management.

View-Model

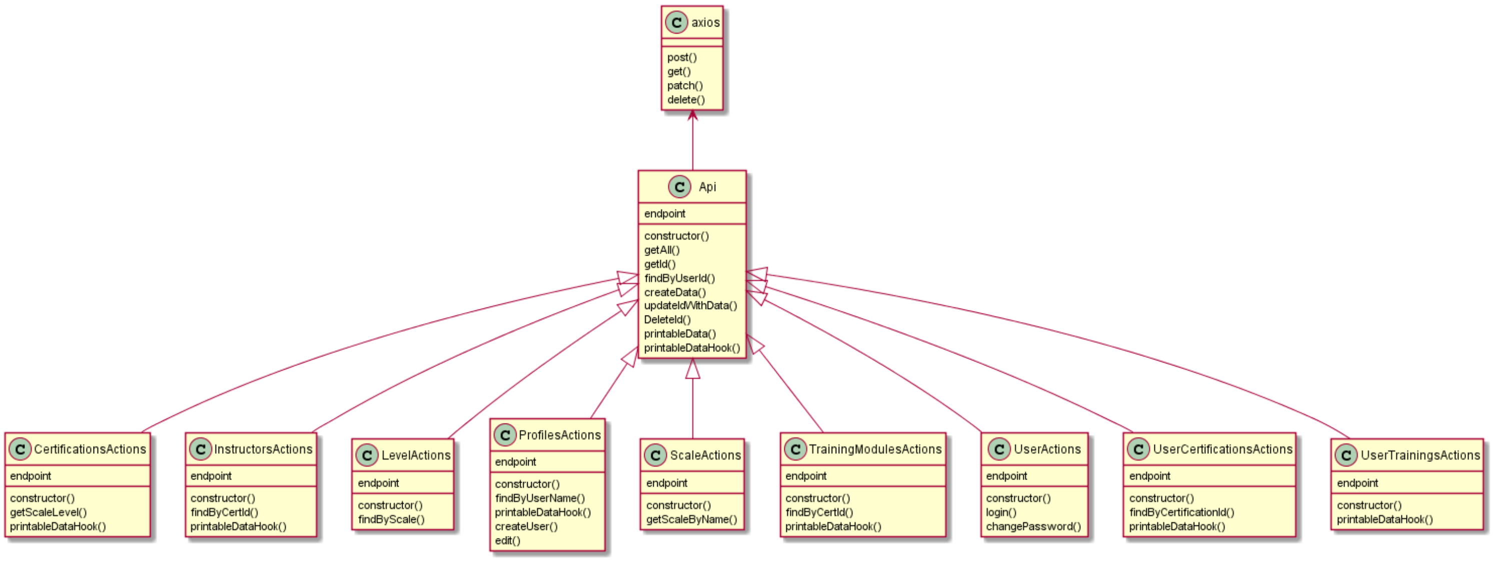
Structure

The View-Model is constructed of 2 main parts:

1. API – used for communication with the model. In charge of transferring data between the view and the model using get, post, patch and delete requests.
2. Redux – used for storing the state of this session. When a user logs-in, an API call is made and all relevant data for this user's session is stored in the state. This data is then reachable throughout the application, without making any other calls to the server.

API class diagram

The API folder contains the main *Api* class that implements all the general and most common actions. All other *APIActions* classes inherit *Api* but override the *endpoint* variable with their unique API-endpoint. Some classes have extra methods that are unique to the data they handle.



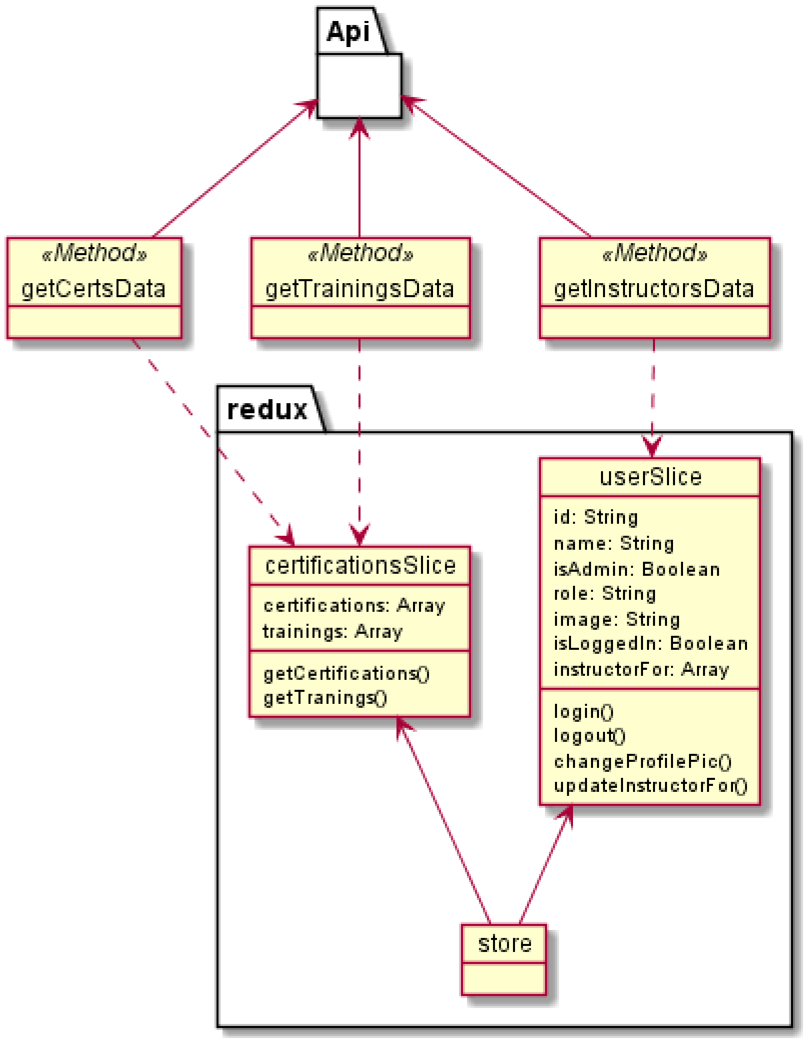
Redux folder diagram

Redux toolkit is used to manage the state throughout the whole React app, meaning the information is accessible from all components of the system. The sate is divided into two sections:

1. UserSlice: includes all of the information about the user logged in as well as methods for overwriting this information.
2. CertificationSlice: includes a list of the certifications and a list of the trainings associated with the user logged in.

Both of the 'slices' are connected to the redux store, that is the root of the applications component tree, assuring this information is reachable in all child components.

Some supporting methods are added to retrieve information from the data base and update the state when needed.



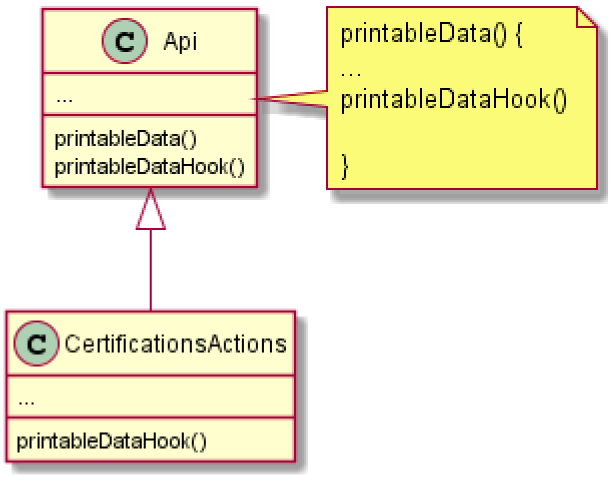
Design patterns

<u>Template Method Pattern:</u>

API class and subclasses are designed using template method design pattern. This part of the project is written in JavaScript, which doesn't have an abstract class notation, there for I used a regular class.

The *printableData* method in the *Api* class implements the parts of code that are similar upon all inheriting classes. The method calls the *printableDataHook* which has been overwritten in all the classes and assures a result based on the data that class represents.

<u>Template Method example - CertificationActions:</u>



View

Structure:

The View is written in React-JS, using functional based components. A component usually has a state and will re-render when the state is changed. States can be passed between parents and children, meaning a change in one components state can trigger re-rendering in the other component. This increases the apps speed by only rendering the components that have changed.

General Structure

React applications are usually built as single page apps – there is a single HTML file in the project (*index.html*), This page renders the *index.js* component.

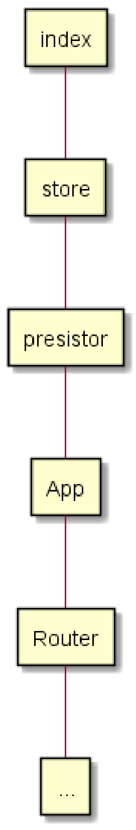
Next there is the *store* that holds the apps internal state (explained under redux store diagram section).

Then *Presistor*, which stores the current state in local storage, assuring the data isn't lost if the connection is cut (for example when the page is refreshed).

Next the tree contains the *App* component, this is the entrance to the visual part of the application.

Finally, under the app there is the *Router* component, that routs the relevant URLs to the website's pages.

This part of the hierarchy tree lives above all "true visual" components.

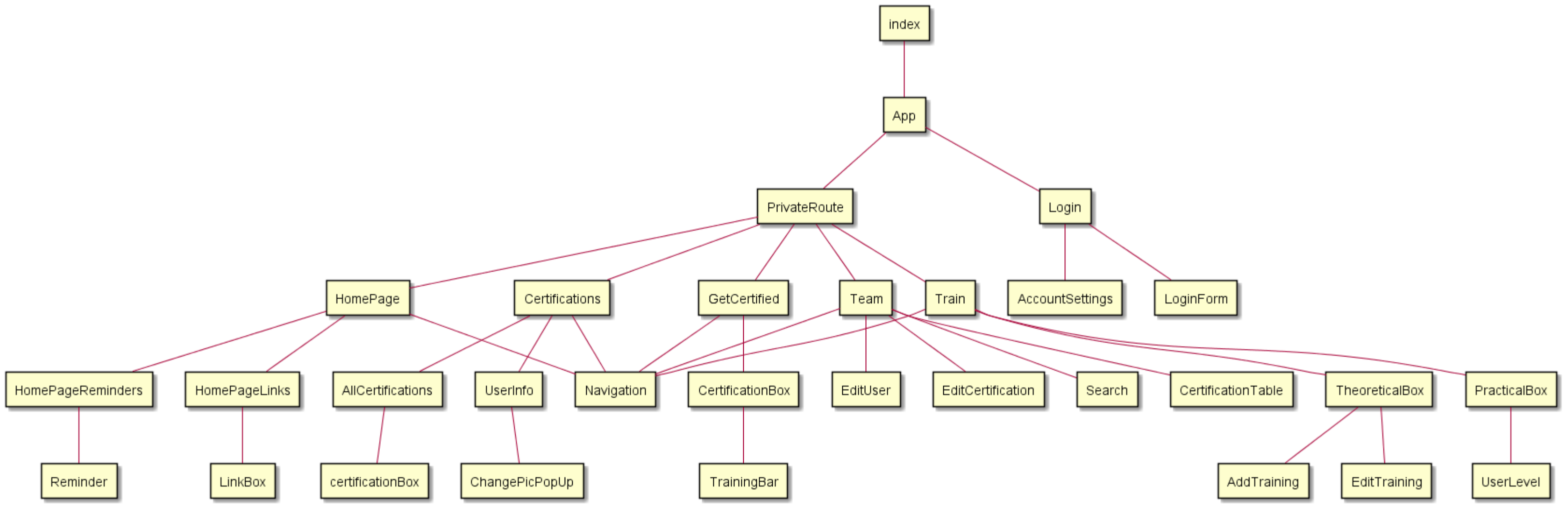


Components hierarchy diagram

A simplified version of the root from the graph above, connected to all the app's subcomponents.

The *App* has two children – *PrivateRoute* and *Login*. The PrivateRoute branch is only accessible for logged-in users, unidentified users will be routed to the login page.

Under the *PrivateRoute* are all the applications pages, and the components that compose them.

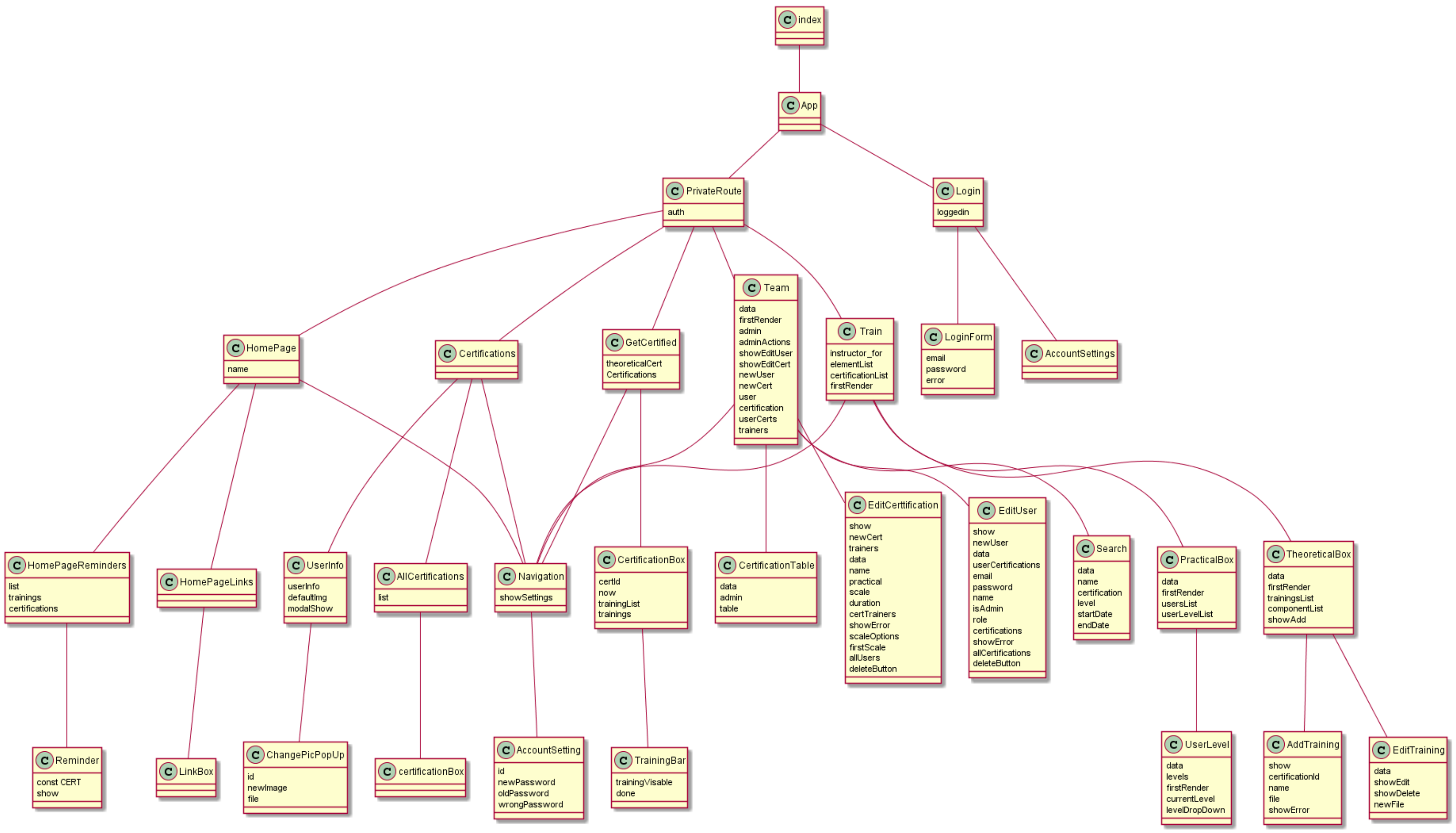


State hierarchy diagram

The diagram shows the component hierarchy with the state variables of each component.

A state oversees the data passed to the component, either for design and visibility purposes or for the information that component is currently displaying.

The state is managed using the use state hook, a state can be gotten by calling that state or edited using the setState call. Once a state is set, the component will re-render.



Design patterns

React apps naturally implement two main design patterns:

1. **Template design Pattern**
   The view is built as a single page application. Meaning, the single HTML page it the base, or the template, and all other components are placed on this template.

2. **Observer design Pattern**
   Reacts components state variables are a classic case of the observer design pattern. A component observes its own state, if the state changes – the component will re-render in order to reflect the change to the user.