

# Documentation

## High Level Frontend Documentation

For details specific to each file, please see the CRC cards in the system design documentation.

The root of the frontend is the `index.tsx` file in `src/`. This component hosts global configuration details for the application. Furthermore, it hosts the `App` component.

## Component Library

We use Material-UI as our component library. Docs for all components are available online.

## Auth

To understand how the frontend works, it is necessary to understand how it handles auth. There are a few key players in this system, `App`, `UserAuthContext`, `PrivateRoute`, `Register` and `Login`.

## App

The `App` component has 2 key responsibilities. Firstly, it is responsible for rendering the route/page that the user navigates to. We use `react-router` for this. However, many of the routes in our application are private and so the `App` component will not allow a route/page to be rendered until the global auth state of the user has been determined. This is done using the `UserAuthContext`.

## UserAuthContext

`UserAuthContext` is responsible for determining and holding the global auth state. It is hosted by `index.tsx` and any subcomponent can access the related auth state. This context is in particular used by the `PrivateRoute` component. The context works by making a request to the `/api/accounts/me` endpoint which returns user specific details

such as whether they are a student or a professor. The request will only be successful if the user already has a session with the backend. If the request is unsuccessful we can conclude that the user is not logged in.

## PrivateRoute

This component is the primary component that enforces auth. Any route that should only be accessible to logged in users or users with a specific role (such as prof or student), will be wrapped in this component within `App`. This component will use the context provided by `UserAuthContext` to check whether the user is logged in. If not, the component will redirect the user to the login page.

## Login and Register

These two pages are the only public routes of the application. The user can access these if they are not logged in. Furthermore, if they login or register, these components will set the context so that the user can access other routes.

## Logout

All private routes render a navbar that hosts the logout button so users can destroy their session.

## API Calls

We use react query to handle all state related to API calls in our components. React query will take a function that makes the API call and will handle different states of the API call such as error, loading and success. All API calls are defined in `/api/api-client.tsx`. These calls are made using an Axios client. Furthermore, the calls all use the request and response types defined in `/api/api-client-types.tsx`.

## Pages

All pages are defined in `src/pages`. Each page is grouped with its respective components. We currently have a couple of main pages; `createProject`, `Projects` (this renders differently based on whether a professor or user accesses the page), `Login`, `Register` as well as a `navbar` page which is rendered on all private routes and allows the user to logout.

## Hooks

At the moment we are only using one hook called `useOnMount` which runs an effect only once on mount.

## index.tsx

Apart from the root level `index.tsx` file; there are multiple such files in the repository. These are used to make imports into other components less verbose (no need to worry about the individual filename).

## High Level Backend Documentation

The backend is structured into separate “apps” each with its own folder. All the urls for each app are located in its respective url.py file. Urls are hooked up to class based views defined in the views.py

## Accounts

Manage user account creation for the backend, views create an Account, validate user login and register, and organize students and professors.

## Projects

Models create the projects and join codes necessary for adding students and professors to projects.

## API

Container for all REST endpoints.

## API Endpoints

### Auth

Register: POST /register

- Accepts: Name, Email, Password, user type (student, prof)

- Business Logic
  - Store password hash
  - Create user entity with no classes
- Returns:
  - 200: User entity and session cookie
  - 400: Could not create user <state reason>
- Sample Request/ Response

```
//Request
{
  "name": "Mohammad Anwar",
  "email": "ma.rizvi.anwar@gmail.com",
  "password": "avada kedrava",
  "user_type": "student" //user_type is ["student", "professor"]
}
//Response 200
{
  "name": "Mohammad Anwar",
  "email": "ma.rizvi.anwar@gmail.com",
  "user_type": "student"
}
//Response 400
{
  "reason": "Could not create user" // Should be implemented in future if not now
}
```

- Login: POST /login
  - Accepts: Email, password
  - Business Logic
    - Compare password hashes for the input email
  - Return
    - 200: user entity and session cookie
    - 400: on invalid credentials
  - Sample Request/Response

```
// Request
{
  "email": "ma.rizvi.anwar@gmail.com",
  "password": "avada kedrava",
}
// Response 200
{
  "name": "Mohammad Anwar",
  "email": "ma.rizvi.anwar@gmail.com",
  "user_type": "student"
}
// Response 400
{
  "details": "Email or password is incorrect"
}
```

- Me: GET /me
  - Explanation
    - FE needs to know user **auth state** before loading any page to know whether to display it to them
      - For separation of concerns; pages should not have to deal with redirects due to unauthorized user; wrapper component will deal with that
    - Fe needs to know whether to display prof/student view
  - Accepts: nothing (request will be empty)
  - Business Logic
    - Check if user is authorized (has session cookie)
  - Return:
    - 200: User entity
  - Sample Request/Response

```
// No request body
// Response 200
{
  "name": "Mohammad Anwar",
  "email": "ma.rizvi.anwar@gmail.com",
  "user_type": "student"
}
```

## Projects

- View projects: GET /projects
  - Accepts: ^
  - Business Logic:
    - Return all projects that user is a part of
  - Return:
    - 200: list of classes
    - 403: not logged in request
  - Request/Response

```
//Request
{}
//Response 200, returns LIST of Project objects (linked to prof!)
{
  [{"project_name": "CSC01 Final Projects",
    "min_group_size": 3,
    "max_group_size": 5,
    "join_code": "hjhh454h3kh5",
    "end_date": "" //FE to standardize to UTC +0, BE to store as UTC +0
    "description": "CSC01 Projects", "professor": {"name": "Iilir Dema"}}, {...}]
}
```

- Join/View project: GET /projects/<project\_id>
  - Accepts: ^
  - Business Logic:
    - Has user joined project?
      - No?
        - Add user to project
    - Return project details (& student state wrt project)
  - Request/Response

```
//Request
{}
```

```
//Response 200
{
  "project":
  {
    "project_name": "CSC01 Final Projects",
    "min_group_size": 3,
    "max_group_size": 5,
    "join_code": "hjhh454h3kh5",
    "end_date": "" //FE to standardize to UTC +0, BE to store as UTC +0
    "description": "CSC01 Projects",
    "professor": {"name": "Iilir Dema"}"
  },

  "newly_joined" : false or true //false implies that user was already apart of the class
}
//Response 404
{
  "reason": "Project not found" // Should be implemented in future if not now
}
```

- Return:
  - 400: user doesn't exist
  - 403: not logged in request
- Create a project: POST /projects
  - Accepts: Project name, min/max group size, Group close date, description
  - Business Logic:
    - Store date as UTC +0
    - Create entity, return success
  - Return:
    - 403: on non-prof request
    - 400: bad request
    - 200: Created project
  - Request/Response

```
//Request
{
  "project_name": "CSC01 Final Projects",
  "min_group_size": 3,
```

```
"max_group_size": 5,  
"end_date": "" //FE to standardize to UTC +0, BE to store as UTC +0  
"description": "CSC01 Projects",  
}  
//Response 200  
{  
"join_code": "2318921798213",  
}  
//Response 400  
{  
"reason": "Invalid name" // Should be implemented in future if not now  
}
```