Backend Docs, se03-grp-7

Subigya Paudel, Tuan Pham

The backend of our application runs on django and consists of a single application called server which will serve appropriate frontend templates, and provide user creation and authentication services. The database that is being used is MySQL. The file structure of the backend (including only the important files and folders) are as follows:

|- manage.py
|- assets (contains all the static files)
|-beer_game
    |- settings.py (contains configuration settings related to location of static files, database the database settings, allowed server domains, CORS policy etc.)
    |- urls.py (contains the routing information that the server uses to forward requests to the appropriate django application, all the requests at this point are forwarded to the server application)
|- server
    |- models.py contains all the classes/ORMs that we use in this app
    |- forms.py: contains all the forms that we use to validate the data that has been sent to to the server. Contains only CreateUserForm up to this point.
    |- urls.py: contains all the valid routes that are handles by this application and the functions to be invoked once these routes have been visited.
    |- views.py: contains all the functions that encode the application logic of the backend. functions within this file are used in urls.py
    |- admin.py: contains all the models that can be “visually” seen in the admin site of the Backend
    |-tests: Contains the source code for backend testing
    |- templates: contains the frontend react templates (both their source codes and compiled counterparts). The compiled react js source code is served by the server.
        |- game_screen: The template for the game screen
            |- plain javascript/html/css files
        |- index: The template for the index page
            |- build: The compiled react code for the index page
            |- src: The source code (in react) for the index page
        |- instructor: The template used when an instructor logs in
            |- build: The compiled react code for the instructor page
            |- src: The source code (in react) for the instructor page
        | - player: The template used when a player logs in
            |- build: The compiled react code for the player page
            |- src: The source code (in react) for the player page

Routes:

The routes that our application handles up to this point are:

Routes	Methods Supported/ Intended	Purpose	Things required in the request	Response Type
“”	GET	Get the landing page of the application, or redirect users of “welcome/” route if they are signed in	Nothing	Compiled React JS bundle
“create_new_user/”	POST	Create a new user, provide validation errors in the frontend simultaneously	Details for creating new user in JSON, and a csrf_token	Status text of “OK” when the user has been created and authenticated , or validation errors
“welcome/”	GET	Serve the appropriate signed-in view to users	Nothing	Compiled React JS bundle
“sign_in/”	POST	Signing in to the application	Username and password in JSON format	Status text of “OK” or “NO” depending on whether the authenticatio-n was successful or not
“logout/”	GET	Logging out of the application	Nothing	Compiled React JS bundle
"play_games/<str:id>/"	GET	Serves the game screen to the player	Game id as a url parameter	text/HTML, since the game_screen is made using plain HTML, CSS, and JavaScript

Models:

Player: A class that represents a player. Upto this point only player only has a user_object (which is a general user used for authentication) and a primary key

Instructor: A class that represents an Instructor. Upto this point the instructor only has a user_object (which is a general user used for authentication) and a primary key.

Mapping: A class that is used to check whether a general, authenticated used is a player of an instructor.

Security and POST requests:

To make sure that the application is safe from CSRF attacks, all the routes in the website upto this point are csrf protected, meaning that one cannot make POST requests without providing a CSRF token that was previously provided by the backend. So, when additional functionalities are added in the application that requires communication with the server, then the frontend will have to extract the csrf_token and provide it in the subsequent request. This has been done in the following manner in the index template

At the end of the compiled index.html of the template for the landing page in templates/index/build (generated after running frontend_compiler.sh)

```
    <script src="/static/js/2.3b8f7c63.chunk.js"></script>
    <script src="/static/js/main.829aed13.chunk.js"></script>
  </body>
</html>
<form id="csrf_cup">
        {% csrf_token %}                    <--- csrf_token is rendered by the server here
</form>
```

So, in order to make asynchronous POST requests, one needs to extract the value of the csrf_token and place it in the header of the request. For instance, while using fetch API:

```
fetch(some_url, {
        method: “POST”,
        headers:{
                “X-CSRFToken”: document.querySelector(“#csrf_cup > input”).value
                        csrf verification passed when this is present in the header

        }
}
```