UNIVERSIDAD
DE CHILE

Reporte de Trabajo

CC5905-1 Trabajo Dirigido

Profesor: Jocelyn Simmonds
Alumna: Valeria León

1. Proyecto y Objetivos específicos

El sistema intervenido corresponde a la plataforma para organización y gestión de eventos deportivos estudiantiles masivos, específicamente para los JING, Juegos de Ingeniería, que se realizan períodicamente entre las facultades y escuelas de ingeniería de distintas universidades en Chile.

Para este trabajo en específico, los objetivos establecidos fueron tres:

- Implementar manejo de usuarios
- Carga de archivos de datos
- Implementar sistema de notificación

De los cuales, solo los primeros dos fueron abordados parcialmente. Los detalles de lo realizado se presentan a continuación.

2. Trabajo Realizado

2.1. Manejo de Usuarios

En un inicio, este objetivo se enfocaba en establecer los permisos de usuarios de acuerdo a los roles establecidos dentro de la organización de un evento deportivo, pero durante las semanas de trabajo se realizó también el *login* de usuarios provisorio, para poder extraer los permisos desde las *request* entre front y backend. Se presentarán los módulos agregados y funciones modificadas para cada parte.

2.1.1. Backend

- **Authentication:** Este módulo tiene que ver con la autenticación de usuario en el sistema. Este módulo tenía algunos archivos pero se agregaron y modificaron funciones para implementar el login provisorio y el sistema de permisos.
  - **models.py , admin.py :** Se agregaron los modelos para guardar los roles asociados a cada persona en tablas dentro de la base de datos. Son cinco clases: AdminRole, EventCoordinatorRole, UniversityCoordinatorRole, SportCoordinatorRole y TeamCoordinatorRole. Estas clases fueron implementadas junto a Sebastián Cisneros.
  - **permissions.py :** Se definen las clases de permisos para cada rol, las cuales son utilizadas para consultar si el usuario contenido en la request posee el rol. Sin embargo, existe algún error en el user enviado por la request o en el evento, ya que siempre se retorna falso, incluso aunque el usuario logeado tenga el rol consultado y sea una instancia de *Person.* Esto es, cuando son llamadas, el *try* no se ejecuta exitosamente (la llamada a la base de datos).

- **serializars.py :** Se agrega la clase *TokenObtain*, que contiene la función *getToken*, la que al llamarla con un usuario obtiene el token asignado a dicho usuario; se utilizó la librería *simpleJWT* de Rest Framework. Finalmente, se agregó la clase *RegisterSerializer* que se utiliza al registrar nuevos usuarios, esta clase no se utiliza actualmente, pero puede ser útil para futuras iteraciones.
- **urls.py :** Se agregan las urls de los *endpoints* para el registro, acceso a token, actualización de token y obtención de permisos.
- **views.py :** Se agregan las clases *TokenObtainView* y *RegisterView* que utilizan los serializers descritos anteriormente. Además se define la función *get_permissions*, que al recibir una request, retorna todos los permisos que tiene el usuario autenticado en una *Response*.

- **backend:** Este módulo contiene configuración para poder correr el backend del proyecto y el registro de las urls a utilizar. Específicamente solo se modificó el archivo *settings.py*, en lo referente a CORS y los validadores de autenticación.

2.1.2. Frontend

Todas las modificaciones hechas en este módulo están dentro de la carpeta *src*, por lo que se listarán los módulos y archivos dentro de ella que fueron modificados.

- **components:**
  - **navbar:** Se modificó el archivo *NavBarMidSection.jsx*, que controla la sección media de la barra de navegación superior de la plataforma, que permite acceder a las diferentes herramientas y páginas disponibles. Se agrega un botón de navegación en caso de que la persona logeada tenga algún permiso de administración para llevar al panel de administración.
- **contexts:** Se crea un contexto de React que permite guardar la información relativa a la autenticación y autorización de usuario, esto en el ardhivo *UserContext.jsx*. Se guardan en contexto y además en el *local storage* el nombre de usuario, el token de autenticación y los permisos asociados a ese usuario. Se define la función *loginUser* que toma como argumento un usuario y contraseña y envía una request al backend para recibir un token asociado a ese usuario. Aquí debiese modificarse tanto esa request como la contraparte del backend para que se inicie sesión en el sistema con el usuario y se reciba también el toke, en lugar de solo recibir el token. Es por ese error de diseño que si no se tiene la misma sesión iniciada en el panel administrador de Rest Framework no funcionan el resto de las funciones como deberían, ya que una vez enviada la request de inicio de sesión y recibida la respuesta, se envía una segunda request para solicitar los permisos que posee un usuario y se guarda toda la información en local. Se tiene además una función para registro de usuario, que no fue utilizada en este trabajo y una función de logout que solo borra la memoria local, no cierra la sesión en el backend.
- **pages:**
  - **Admin:** en el archivo *index.jsx* se tiene la función admin que verifica si xiste un usuario autenticado en el contexto para redirigir a la función *AdminIndex()*, la cual se encuentra en un archivo dentro de pages del mismo nombre (debiese refactorizarse a la carpeta Admin). Esta función renderiza la página de administración y muestra el contenido de acuerdo a los permisos específicos del usuario. Sin embargo, esto no funciona de manera correcta al no poder extraer los permisos de buena forma desde el back, explicado anteriormente. Este archivo, además, está programado en HTML con React embebido, no modularizado como otras secciones actualizadas de la plataforma. En el archivo *NavBarAdmin.jsx* se tiene una barra de navegación para poder navegar entre las distintas pestañas de administración y que debiese mostrar diferentes opciones de acuerdo a los permisos del usuario.

- **Login:** en el archivo *index.jsx* dentro de esta carpeta, se encuentran las funciones que renderizan y muestran la página de login, además del llamado a las funciones del contexto para efectivamente autenticar a un usuario.

- **utils:** en el archivo *PrivateRoute.jsx* se encuentra la función que verifica que, al visitar la url de administración, si el usuario no está autenticado o no tiene permisos, redirija a la página de login. En el archivo *useAxios.jsx* se tienen funciones que interceptar las request enviadas desde front a back para verificar que el token de autorización guardado esté aún vigente. En caso de no estarlo, se refresca el token y se guarda, cambiándolo además en la request interceptada.

2.2. Carga de datos

Este objetivo se enfocaba en permitir a los usuarios con algún permiso de administración poder cargar datos desde archivos excel, pero evitar la carga manual de miles de datos. Se realizó una primera aproximación de solución para este problema, ya que por ejemplo se podrían cargar datos en diferentes formatos de archivos o en uno solo generalizado. La solución en este nivel, asume lo último, que se tiene un archivo de carga de datos con ciertas personas y datos que pide el modelo de Persona, pero podría modificarse para aceptar también la información de los Equipos y Deportes a los que pertenece.

2.2.1. Backend

- **Administración:**

  - **urls.py :** se habilita endpoint en *upload-data* para recibir los datos enviados desde el frontend.
  - **views.py :** se crea la función *_uploadPersonData* que recibe una request con un archivo JSON que contiene la data y la inserta en la base de datos de Persona. Al ser esta función una primera aproximación, solo pide que traiga en el archivo los datos obligatorios que pide el modelo de persona, además de requerir que la data venga ya parseada desde el frontend como un archivo JSON con personas y sus datos contenidos. Sin embargo, esta función aparece como si no fuese accesada, ya que si bien la request se recibe desde frontend, devolviendo un status 200, los datos no son ingresados realmente en la base de datos de la plataforma.

2.2.2. Frontend

- **pages/Admin :** en el archivo *uploadFile.jsx* se define la función *FileUploader* que agrega el componente a la página de Administración que permite seleccionar un archivo para ser cargado, lo procesa y lo guarda en un estado. La función *processData* procesa las hojas del archivo excel subido y les da un formato JSON. Al ser enviado el formulario, se llama a la función *sendData*, que crea la request con el archivo contenido para poder enviarla al backend. Esta función envía la request, pero es en el trayecto desde esta función al backend en *_uploadPersonData* en que se está perdiendo la data enviada.

A continuación se entregan las principales referencias utilizadas para este trabajo. Cabe destacar que se utilizaron otras fuentes para búsqueda de errores y debugging, como *Stack Overflow*, pero que no fueron agregadas por simplicidad (eran muchos links).

3. Referencias

- **Autenticación y Manejo de Usuarios**
  - *Django Rest Framework + React — Authentication workflow 2022 (Part 2)*, Sushil Kamble.
    https://blog.devgenius.io/django-rest-framework-react-authentication-workflow-2022-
  - *Django and React Integration*, Priyanshu Gupta.
    https://priyanshuguptaofficial.medium.com/django-and-react-integration-b712321a5232
  - *Implementing Authentication in React using React Context API — Part 1 (React Context API)*, Khushal Agarwal.
    https://khushal87.medium.com/implementing-authentication-in-react-using-react-contex
  - *Role based authorization in React*, Carl Vitullo.
    https://blog.vcarl.com/role-based-authorization-react/
  - *How To Add Login Authentication to React Applications*, Joe Morgan.
    https://www.digitalocean.com/community/tutorials/how-to-add-login-authentication-to-
- **Carga de Datos**
  - *How to upload files in React using Axios*, Suraj Sharma.
    https://surajsharma.net/blog/react-upload-file-using-axios
  - *React js Transformar datos de excel a json.*, Leopoldo Ramon Montesinos.
    https://medium.com/@leopoldoramonmontesinos/react-js-transformar-datos-de-excel-a-j