

CoderDojo website

Proyecto final - Programación Web 2

Resumen

“Este proyecto está destinado a ser usado por los estudiantes del evento CoderDojo, para que puedan encontrar los recursos de aprendizaje y hacer el envío de sus tareas.”

Integrantes

- Alumno: Arias Quispe Jhonatan David
- Alumno: Chambilla Perca Ricardo Mauricio
- Alumno: Carbajal Gonzales Diego Alejandro

URL a los repositorios:

- Frontend: <https://github.com/JhonatanDczel/coder-dojo-front>
- Backend: <https://github.com/rikich3/coderDojoBack>

Coder Dojo y la IEEE CS Unsa

Aplicación para el evento de Coder Dojo que está organizando la **IEEE Computational Society rama Perú** por parte de la **UNSA**.

Requerimientos

Las especificaciones que recibimos de los coordinadores fueron:

1. Tipos de Usuario:

- **Profesor:** Puede crear salones virtuales y asignar tareas a los estudiantes.
- **Estudiante:** Puede unirse a los salones y acceder a las tareas asignadas.
- **Administrador:** Gestiona la plataforma.

2. Salones Virtuales:

- Los profesores pueden crear salones y asignar tareas a los estudiantes dentro de esos salones.

- Los salones contienen a estudiantes y un profesor.
- Los profesores pueden publicar material de estudio y otros recursos que los estudiantes pueden ver.

3. **Usuarios Objetivo:**

- Estudiantes de secundaria.
- La plataforma debe incluir elementos de gamificación para aumentar la atención y el compromiso.
- Utiliza un framework moderno desarrollado por Octolasy Group para mejorar la experiencia del usuario.

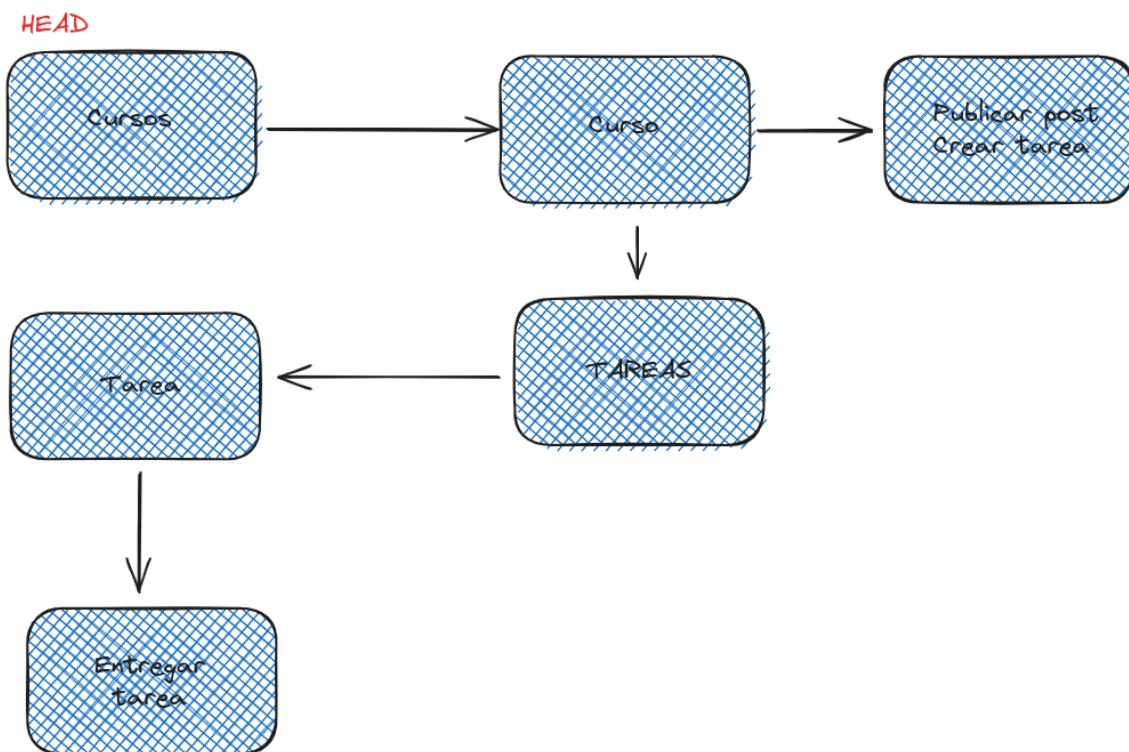
Planificación

Una vez recibidas las especificaciones, procedimos a crear los mockups en una herramienta de diseño: [excalidraw](#).

REQUERIMIENTOS:

Estudiante	Profesor	Interfaz
Cursos <ul style="list-style-type: none"> - Vista de cursos generales - Vista de un curso, con todos los post/tareas/recursos de este Tareas <ul style="list-style-type: none"> - Vista de tareas pendientes y deadlines - Vista en detalle de la tarea - Entrega de la tarea solo un link 	Cursos <ul style="list-style-type: none"> - Vista de cursos generales - Vista de un curso, con todos los post/tareas/recursos de este y posibilidad de publicar algo Tareas <ul style="list-style-type: none"> - Vista de tareas por curso - Vista en detalle de la tarea Quienes entregaron y quienes no - Creación de tareas (a modo de post) 	Sidebar <ul style="list-style-type: none"> - Menú vertical para ir a cursos, tareas, foro - Menú de información, para dar una guía al estudiante Foro <ul style="list-style-type: none"> - Vista general de los post - Redactar un post - Comentar / reaccionar a un post

NAVEGACIÓN DE PÁGINAS



Desarrollo

Una vez entendida la dinámica de la interacción entre las páginas, se procede con el desarrollo, que está dividido en dos partes: [frontend](#) y [backend](#).

Stack de desarrollo

Las tecnologías que se eligieron para el proyecto, fueron:

- **Django**: Para el lado del backend.
- **Django REST Framework**: Usado para hacer las API que consumira el frontend.
- **React**: Para crear las vistas y aprovechar el aspecto reactivo de react.
- **Tailwind**: Para manejar los estilos de una mejor manera a css puro.

Frontend

Para el desarrollo del lado del frontend, se dividió en las siguientes tasks:

- **Login** - que debería tener la autenticación de los usuarios, así como el minigame [DojoType](#).
- **Dashboard** - que debería tener el acceso a las vistas como los cursos, vista por curso, y las tareas que se tienen.

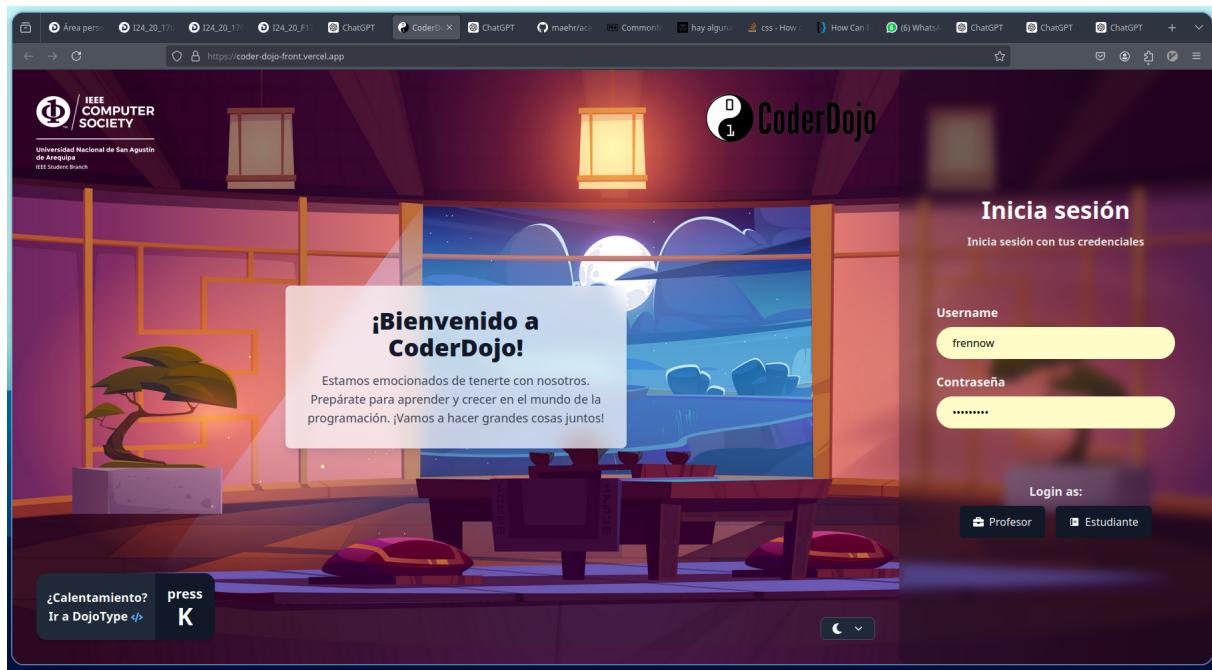
Login

El login se hizo en `react` con un diseño basado en componentes, se enfatizó el concepto de [gamificación](#) de la interfaz para hacerla más amigable para nuestro público objetivo (estudiantes de secundaria).

Dark - Light - System modes

Con ese objetivo en mente, se optó por una interfaz sencilla, clara y concisa, que destaque los elementos importantes y que tenga un toque fresco, esta es una vista de la interfaz:

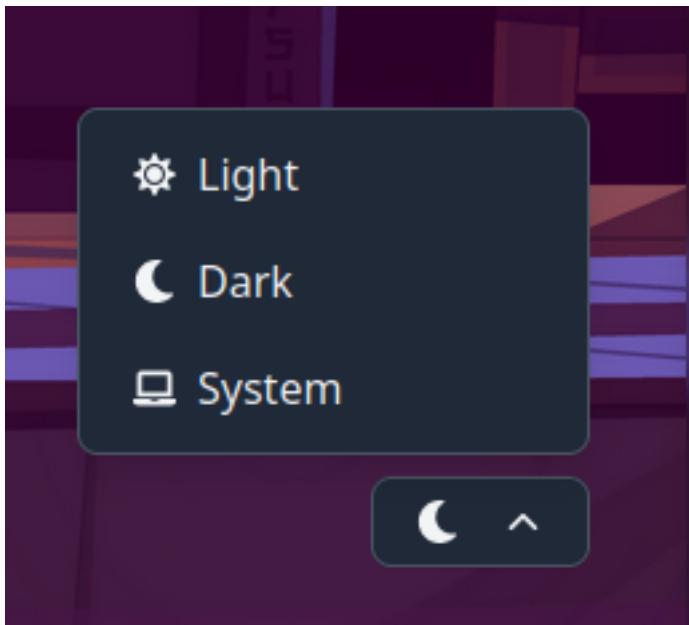
Modo dark



Modo light



Para poder hacer el cambio entre modos dark, light o system se usó el siguiente componente:



que es un componente de react:

```
1 // src/components/common/ThemeSwitcher.js
2 const ThemeSwitcher = () => {
3   const [theme, setTheme] = useState("system");
4   const [isOpen, setIsOpen] = useState(false);
5
6   useEffect(() => {
7     const root = window.document.documentElement;
8     if (theme === "dark") {
9       root.classList.add("dark");
10      localStorage.setItem("theme", "dark");
11    } else if (theme === "light") {
12      root.classList.remove("dark");
13      localStorage.setItem("theme", "light");
14    } else {
15      root.classList.remove("dark");
16      if (window.matchMedia("(prefers-color-scheme: dark)").matches) {
17        root.classList.add("dark");
18      }
19      localStorage.removeItem("theme");
20    }
21  }, [theme]);
22
23 const handleThemeChange = (newTheme) => {
24   setTheme(newTheme);
25   setIsOpen(false); // Close the dropdown after selection
26 };
```

DojoType

Junto al concepto de gamificación viene esta idea: hacer un centro de entrenamiento previo, en el que los alumnos puedan ejercitarse sus habilidades de tipeo antes de hacer las lecciones de CodeDojo.

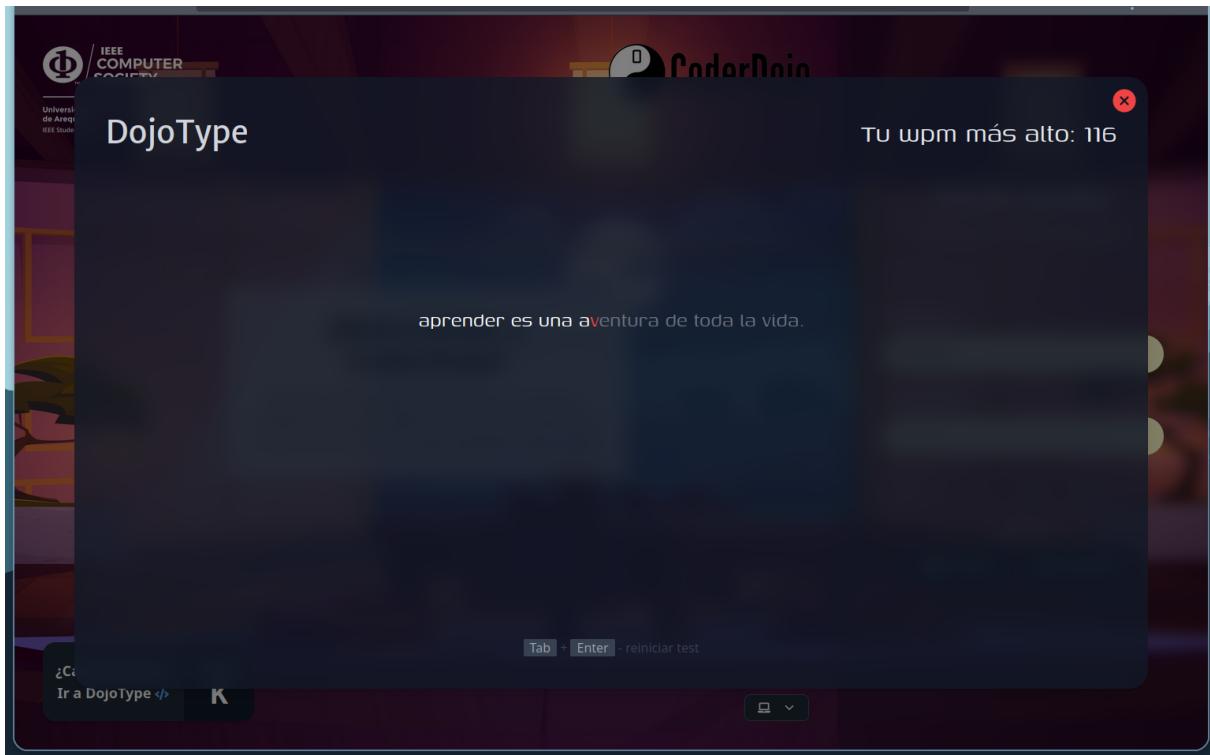
Para esto, hicimos DojoType!, que es un minigame a modo de test de mecanografía que mide tu velocidad y la guarda para mantener un record que romper.

El diseño inicial pensado fue este:

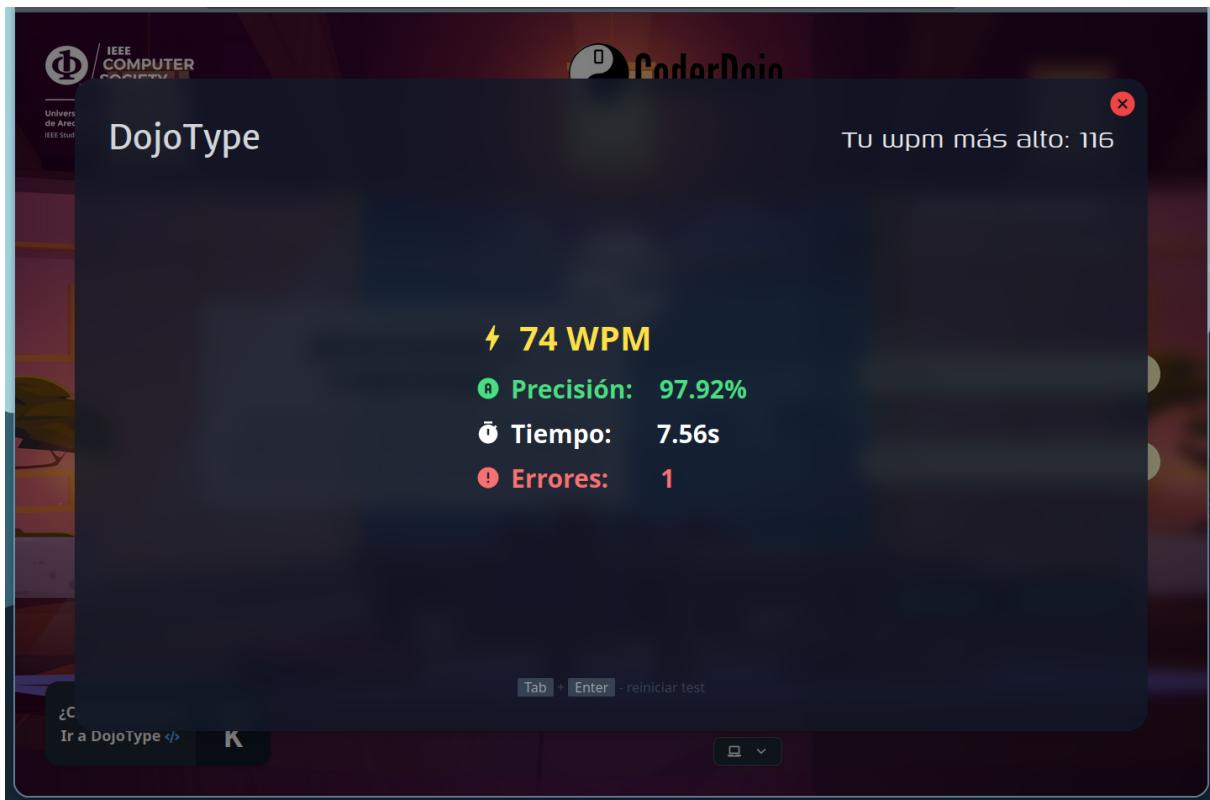


Para su implementación, y en pos de la modularización de código, se hizo un nuevo componente, que llamamos dentro de un PopUp (también componente) que contendrá el juego, esto nos permite, por ejemplo, escalar el minigame si así lo queremos hasta convertirlo en un proyecto independiente, la vista final fue esta:

DojoType



Visualización de los datos del test



Composición de la interfaz

La interfaz esta compuesta siguiendo reglas de código limpio y buenas prácticas de programación, a continuación tenemos el código del componente [HomePage](#) que se encarga del renderizado de la página principal:

```
1 //src/pages/HomePage.jsx
2     <div className="...">
3         <div className="...">
4             <div className="...">
5                 <h1 className="...">
6                     ¡Bienvenido a CoderDojo!
7                 </h1>
8                 <p className="text-lg text-gray-700">
9                     Estamos emocionados de tenerte con nosotros. Prepárate
10                    para
11                    aprender y crecer en el mundo de la programación.
12                    ¡Vamos a hacer
13                    grandes cosas juntos!
14                </p>
15            </div>
16        </div>
17        <div className="absolute top-5 left-5 p-4">
18            <Logo path={IEELogo} size="h-28"/>
19        </div>
20        <a href="https://coderdojo.com/en/" target="_blank" className
21            ="inset-1">
22            <div className="absolute top-5 right-5 p-4">
23                <Logo path={coderDojoLogo} size={"h-[4.5rem]"} />
24            </div>
25        </a>
26        <div className="absolute bottom-5 left-5 p-4">
27            <DojoTypeButton onClick={handleDojoTypeButtonClick} />
28        </div>
29        <div className="absolute bottom-5 right-5 p-4">
30            <ThemeSwitcher />
31        </div>
32    </div>
33    <LoginForm />
```

Vistas de los cursos

Las vistas de cursos y de cada curso individual una vez que el usuario se autentifica son las siguientes:

The screenshots illustrate the CoderDojo website's user interface. The top screenshot shows the 'Clases' (Classes) page, which lists three courses: 'Estructuras de datos' (Data Structures), 'Algoritmos' (Algorithms), and 'Bases de datos' (Database Systems). Each course card includes the author's name (Edson Luque) and a 'Ver curso' (View course) button. The bottom screenshot shows the detailed view for the 'Estructuras de datos' course, displaying a list of five pending tasks (Tarea 1 through Tarea 5) and two buttons: 'Ver Tareas' (View Tasks) and 'Ver Calificaciones' (View Grades).

Para los cursos se hacen peticiones al backend, que devuelven una lista de cursos en los que el docente/estudiante forma parte, todos estos datos se usan para dibujar la interfaz en el front:

```
1  [
2    {
3      "id": 1,
4      "docente": {
5        "id": 1,
6        "name": "Ricardo"
7      },
8      "name": "Frameworks Web",
9      "entregas": [
10        {
11          "id": 1,
12          "file": "/entregas/GITHUB.txt",
13          "submitted_at": "2024-07-24T08:41:23.819654Z",
14          "estudiante": {
15            "id": 1,
16            "name": "pepe"
17          },
18          "asignacion": {
19            "id": 1,
20            "title": "Generar paginas con Javascript",
21            "description": "Manipulando el dom con javascript"
22          }
23        }
24      ],
25      "publicaciones": [
26        {
27          "id": 1,
28          "content": "Ejemplo de publicacion para alumnos"
29        }
30      ],
31      "estudiantes": [
32        {
33          "id": 1,
34          "name": "pepe"
35        },
36        {
37          "id": 2,
38          "name": "Raquel"
39        }
40      ]
41    }
42  ]
```

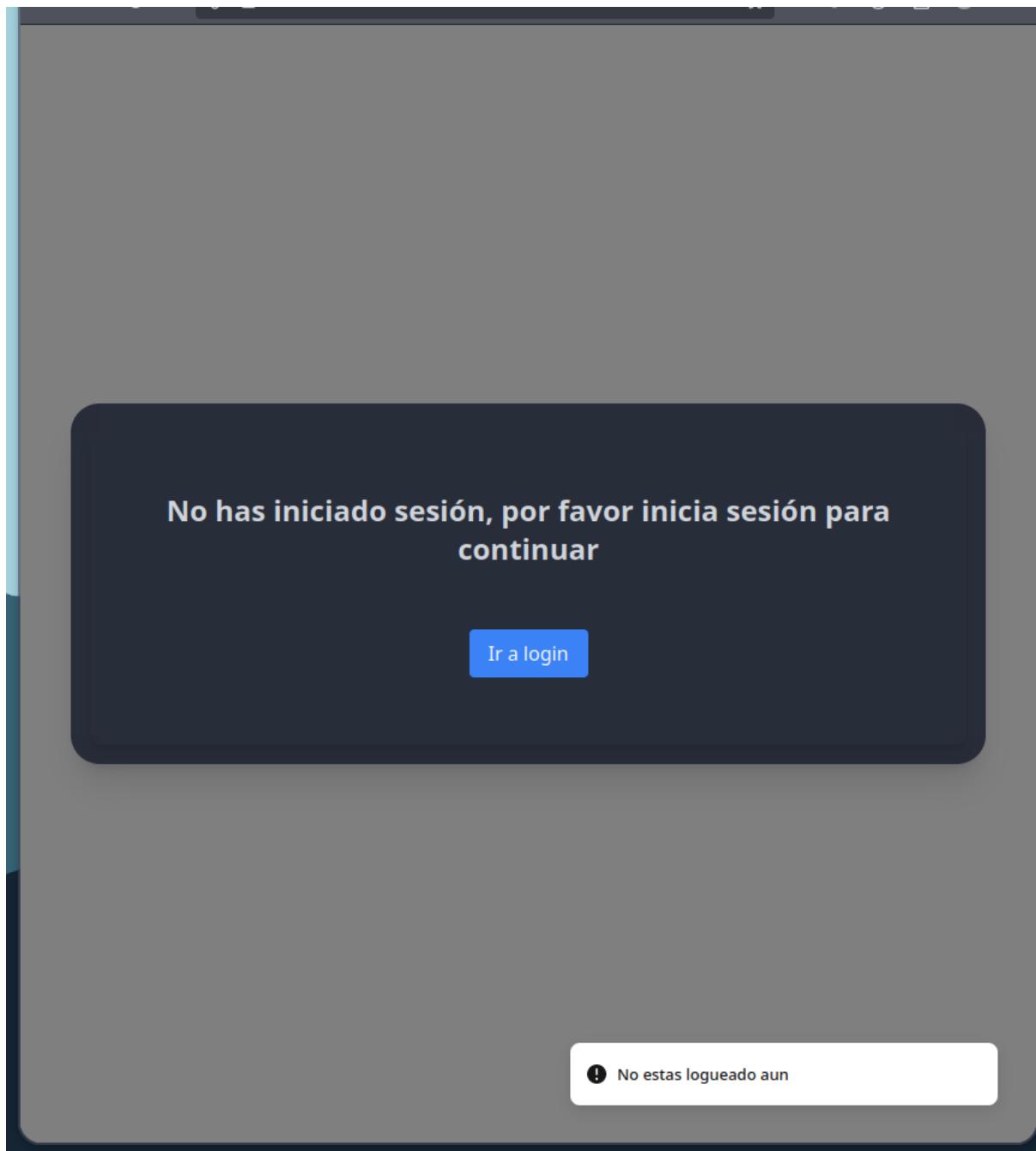
Verificación de inicio de sesión

Para evitar que cualquier persona intente entrar con el link directo, se desarrolló un sistema de verificación, que hace un llamado a la API para ver si una `sessionId` esta autenticada o no.

La llamada a la API nos da algo como esto:

```
1 // el endpoint es /api/is_authenticated
2 HTTP 200 OK
3 Allow: GET, OPTIONS
4 Content-Type: application/json
5 Vary: Accept
6
7 {
8     "authenticated": true
9 }
```

En caso de no estar autenticado el usuario, se muestra un [PopUp](#) y un mensaje emergente invitándolo a registrarse:



Composición:

La composición para estas vistas fué la siguiente:

```
1 return (
```

```
2   <div className="flex">
3     <Sidebar />
4     <div className="ml-64 w-full">
5       <Navbar data={data} />
6       <div className="pt-16">
7         <StudentRoutes />
8       </div>
9       <CoursesList />
10    </div>
11  </div>
12 );
```

Para el ruteo de la web se usó la biblioteca `react-router-dom`, la estructura es la siguiente:

```
1 <Routes>
2   <Route path="/" element={<HomePage />} />
3   <Route path="/clase/1" element={<DashBoard rol={1} />} />
4   <Route path="/clase/0" element={<DashBoard rol={0} />} />
5 </Routes>
```

Backend

Introducción

Ahora se procederá a describir la implementación del backend del proyecto CoderDojo, una plataforma educativa diseñada para facilitar la gestión de clases, publicaciones y asignaciones para estudiantes y docentes.

Los servicios API del back y la lógica están desarrollados en Django y utiliza Django REST Framework para la creación de APIs. A continuación, se detalla la estructura de los modelos, vistas y serializadores, así como la configuración de las rutas y permisos.

Modelos

Los modelos representan la estructura de la base de datos y las relaciones entre los diferentes elementos de la aplicación.

A continuación, se describen los modelos más importantes implementados en este proyecto.

Modelo para los Usuarios El modelo `AppUser` extiende `AbstractBaseUser` y `PermissionsMixin` para crear un sistema de autenticación personalizado.

Este modelo define diferentes tipos de usuarios (estudiantes y docentes) y gestiona la creación de usuarios y superusuarios a través del `AppUserManager`.

```

1 class AppUserManager(BaseUserManager):
2     def create_user(self, email, password=None):
3         if not email:
4             raise ValueError('Se requiere un correo electrónico.')
5         if not password:
6             raise ValueError('Se requiere una contraseña.')
7         email = self.normalize_email(email)
8         user = self.model(email=email)
9         user.set_password(password)
10        user.save()
11        return user
12
13    def create_superuser(self, email, password=None, **extra_fields):
14        user = self.create_user(email, password, **extra_fields)
15        user.is_superuser = True
16        user.is_staff = True
17        user.save()
18        return user

```

El `AppUserManager` maneja la creación de usuarios y superusuarios, asegurando que se proporcionen un correo electrónico y una contraseña.

```

1 class AppUser(AbstractBaseUser, PermissionsMixin):
2     user_id = models.AutoField(primary_key=True)
3     email = models.EmailField(max_length=50, unique=True)
4     username = models.CharField(max_length=50)
5     is_staff = models.BooleanField(default=False)
6     is_student = models.BooleanField(default=False)
7     is_teacher = models.BooleanField(default=False)
8
9     USERNAME_FIELD = 'email'
10    REQUIRED_FIELDS = ['username']
11    objects = AppUserManager()
12
13    def __str__(self):
14        return self.username

```

El modelo `AppUser` define los atributos básicos del usuario y extiende funcionalidades de `AbstractBaseUser` y `PermissionsMixin`.

Modelo para los Salones El modelo `Clase` y sus relaciones con `Estudiante` y `Docente` son fundamentales para la organización de las clases y la asignación de tareas. También se incluyen modelos para gestionar publicaciones y entregas de asignaciones.

```

1 class Clase(models.Model):
2     name = models.CharField(max_length=100)
3     estudiantes = models.ManyToManyField(Estudiante, related_name='clases')

```

```
4     docente = models.ForeignKey(Docente, on_delete=models.CASCADE,  
                                related_name='clases')
```

El modelo `Clase` define una clase con un nombre, una relación muchos-a-muchos con `Estudiante`, y una relación muchos-a-uno con `Docente`.

```
1 class Publicacion(models.Model):  
2     content = models.TextField()  
3     clase = models.ForeignKey(Clase, on_delete=models.CASCADE,  
                                related_name='publicaciones')
```

El modelo `Publicacion` gestiona el contenido de las publicaciones dentro de una clase específica.

```
1 class Asignacion(models.Model):  
2     title = models.CharField(max_length=100)  
3     description = models.TextField()  
4     due_date = models.DateTimeField()  
5     clase = models.ForeignKey(Clase, on_delete=models.CASCADE,  
                                related_name='asignaciones')
```

El modelo `Asignacion` define las tareas con un título, descripción, fecha de entrega y la relación con una clase.

```
1 class Entrega(models.Model):  
2     asignacion = models.ForeignKey(Asignacion, on_delete=models.CASCADE  
                                     , related_name='entregas')  
3     estudiante = models.ForeignKey(Estudiante, on_delete=models.CASCADE  
                                     , related_name='entregas')  
4     file = models.FileField(upload_to='entregas/')  
5     submitted_at = models.DateTimeField(auto_now_add=True)
```

El modelo `Entrega` gestiona las entregas de las asignaciones, incluyendo el archivo entregado y la fecha de entrega.

Vistas y Serializadores

Vistas Las vistas en Django REST Framework gestionan las solicitudes HTTP y devuelven las respuestas adecuadas. En este proyecto, se implementaron vistas para gestionar el inicio y cierre de sesión, así como para listar, crear, actualizar y eliminar clases, publicaciones y asignaciones.

```
1 class LoginView(APIView):  
2     permission_classes = (permissions.AllowAny,)  
3  
4     def post(self, request):  
5         username = request.data.get('username')  
6         password = request.data.get('password')
```

```
7     user = authenticate(request, username=username, password=password)
8     if user is not None:
9         login(request, user)
10        return HttpResponseRedirect('/clases')
11    return Response({'error': 'Credenciales inválidas'}, status=status.HTTP_401_UNAUTHORIZED)
```

La [LoginView](#) permite a los usuarios autenticarse con sus credenciales y redirige a la página de clases si la autenticación es exitosa.

```
1 class ClaseListView(generics.ListCreateAPIView):
2     serializer_class = ClaseSerializer
3     permission_classes = [IsAuthenticated]
4
5     def get_queryset(self):
6         user = self.request.user
7         if user.is_estudiante:
8             estudiante = get_object_or_404(Estudiante, user=user)
9             return estudiante.clases.all()
10        elif user.is_docente:
11            docente = get_object_or_404(Docente, user=user)
12            return docente.clases.all()
13        return Clase.objects.none()
14
15     def perform_create(self, serializer):
16         user = self.request.user
17         if user.is_docente:
18             docente = get_object_or_404(Docente, user=user)
19             serializer.save(docente=docente)
```

La [ClaseListView](#) maneja la lista y creación de clases. Filtra las clases según el tipo de usuario (estudiante o docente) y asegura que solo los docentes puedan crear clases.

Serializadores Los serializadores convierten los modelos de Django en formatos JSON para que puedan ser enviados como respuestas HTTP. Aquí se presentan los serializadores implementados para [Clase](#), [Publicacion](#), y [Asignacion](#).

```
1 class ClaseSerializer(serializers.ModelSerializer):
2     publicaciones = PublicacionSerializer(many=True, required=False)
3     asignaciones = AsignacionSerializer(many=True, required=False)
4     estudiantes = serializers.ListField(child=serializers.IntegerField(),
5                                         required=False)
6
7     class Meta:
8         model = Clase
9         fields = ['id', 'nombre', 'docente', 'publicaciones',
10                  'asignaciones', 'estudiantes']
```

```
9
10    def create(self, validated_data):
11        publicaciones_data = validated_data.pop('publicaciones', [])
12        asignaciones_data = validated_data.pop('asignaciones', [])
13        estudiantes_data = validated_data.pop('estudiantes', [])
14
15        clase = Clase.objects.create(**validated_data)
16
17        for publicacion_data in publicaciones_data:
18            Publicacion.objects.create(clase=clase, **publicacion_data)
19
20        for asignacion_data in asignaciones_data:
21            Asignacion.objects.create(clase=clase, **asignacion_data)
22
23        for estudiante_id in estudiantes_data:
24            estudiante = Estudiante.objects.get(id=estudiante_id)
25            estudiante.clases.add(clase)
26
27        return clase
```

El `ClaseSerializer` convierte los datos de las clases en formato JSON y maneja la creación de clases junto con sus publicaciones, asignaciones y estudiantes asociados.

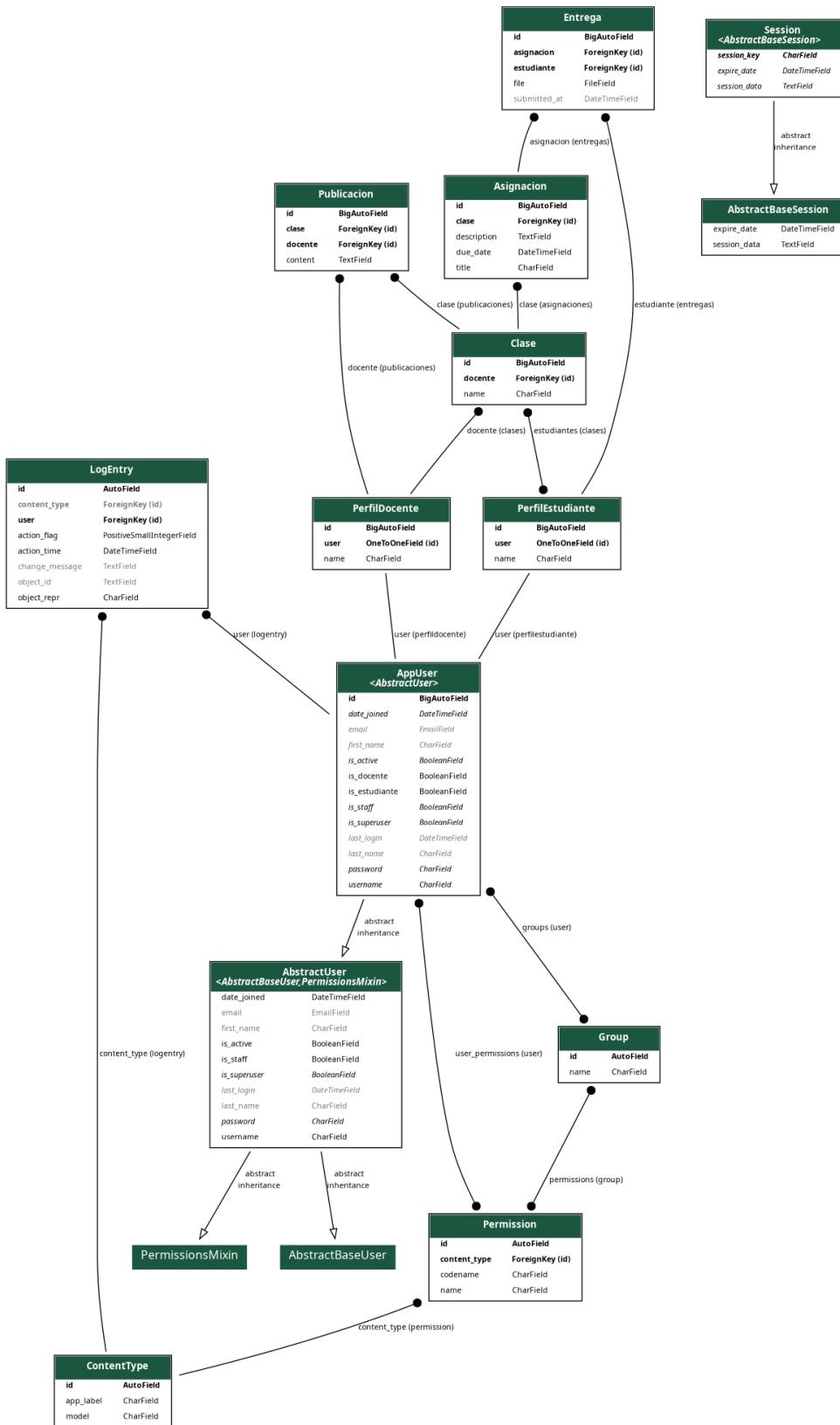
URLs

Las URLs definen los puntos de entrada para las vistas de la API. A continuación, se presenta la configuración de las rutas para las operaciones de autenticación, gestión de clases, publicaciones y asignaciones.

```
1 from django.urls import path
2 from .views import LoginView, LogoutView, ClaseListView,
3                   ClaseDetailView, PublicacionListView, AsignacionListView
4
5 urlpatterns = [
6     path('login/', LoginView.as_view(), name='login'),
7     path('logout/', LogoutView.as_view(), name='logout'),
8     path('clases/', ClaseListView.as_view(), name='clase-list'),
9     path('clases/<int:pk>', ClaseDetailView.as_view(), name='clase-
10       detail'),
11    path('publicaciones/', PublicacionListView.as_view(), name='
12       publicacion-list'),
13    path('asignaciones/', AsignacionListView.as_view(), name='
14       asignacion-list'),
15]
```

Diagrama

Después de hacer las implementaciones, se tiene el siguiente diagrama que representa la estructura interna de nuestra base de datos:



Desplegando en la web

Para hacer el despliegue del proyecto se dividieron los servicios por dos lados.

El front estaría desplegado en [vercel](#), como un servicio estático, aprovechando el empaquetamiento que nos hace vite con el comando

```
1 npm run build
```

El back está desplegado en [railway](#), un servicio cloud que nos permite hostear proyectos en [django](#) de una manera fácil y rápida.

Al momento de llevar nuestro backend a producción, necesitamos sustituir [db.sqlite3](#) con una base de datos real, es así que decidimos usar [postgresql](#), servicio que podemos integrar en railway para tener nuestra base de datos guardada ahí.

Dashboard de railway

The screenshot shows the Railway dashboard for a project named "courteous-passion". The top navigation bar includes "Architecture", "Observability", "Logs", "Settings", and a "TRIAL" button with a price of "\$ 4.99". On the left, there's a sidebar with a "Create" button and a "Postgres" section showing deployment history via Docker Image and Postgres disk. The main area displays the "Activity" log, which lists several successful deployments for the "web" service over the past 5 hours, each triggered by GitHub. A summary at the bottom indicates "1 change in web" made by "Jhonatandczel" 5 hours ago. The overall interface has a clean, modern design with a dark mode option.

Prueba

Para la prueba del backend se puede acceder a <https://web-production-79e8a.up.railway.app> y probar el inicio de sesión con estas credenciales:

```
1  {
2    "username": "frennow",
3    "password": "coderdojo",
4    "rol": "Docente"
5 }
```

Para probar puede rutear el endpoint `/api/loginUser`, o puede entrar directamente desde este enlace deberia ver algo como:

The screenshot shows a browser window displaying a Django REST framework API endpoint for logging in a user. The URL is `https://web-production-79e8a.up.railway.app/api/loginUser`. The page title is "Django REST framework" and the user is identified as "frennow". The main content area shows the "Login User" endpoint. A "OPTIONS" button is visible. Below it, a "GET /api/loginUser" section displays an error message: "HTTP 405 Method Not Allowed" with allowed methods "OPTIONS, POST", content type "application/json", and vary header "Accept". The response body is: { "detail": "Method \"GET\" not allowed." }. Below this, a "POST" section is shown with "Media type: application/json" selected. The "Content" field contains a JSON payload: { "username": "frennow", "password": "coderdojo", "rol": "Docente" }. A "POST" button is located at the bottom right of this section.

Haciendo click en **POST** debería darle un token a la sesión actual del navegador.

Para comprobar que está logueado puede entrar a `/api/` que direcciona a `/api/clase/1`, puede acceder mediante este enlace