



# **ACTIVIDAD FINAL INGENIERIA WEB II**

Claudia Corona Blanco y Marta Pamies Merino

## **1. INTRODUCCIÓN**

## **2. ARQUITECTURA GENERAL DEL SISTEMA**

## **3. DISEÑO DE LA BASE DE DATOS**

## **4. BACKEND: API-REST (FASTAPI)**

## **5. FRONTEND: APP-WEB (FLASK + PLANTILLAS HTML)**

## **6. PROBLEMAS ENCONTRADOS**

## **7. CONCLUSIONES**

## 1. INTRODUCCIÓN

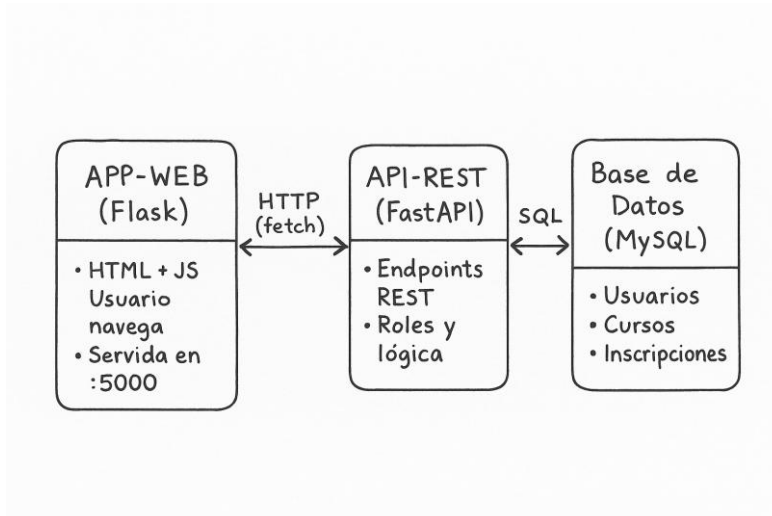
En este proyecto de Ingeniería Web II hemos desarrollado una aplicación web completa dividida en dos partes: por un lado, una **APP-WEB** hecha con Flask, y por otro, una **API-REST** construida con FastAPI que se conecta a una base de datos MySQL local.

El caso que hemos trabajado es el de una academia online de programación, donde los usuarios pueden registrarse, iniciar sesión, ver cursos disponibles, apuntarse a ellos, solicitar nuevos cursos o, si tienen permisos, gestionarlos. Hemos creado distintos tipos de usuarios (user, admin y super), y cada uno tiene acceso a funcionalidades diferentes según su rol.

La APP-WEB se encarga de mostrar las páginas (como el inicio, login, mis cursos, panel admin...) y se comunica con la API usando JavaScript y fetch. Por su parte, la API maneja toda la parte lógica: creación de cursos, gestión de usuarios, control de accesos, autenticación con tokens JWT, etc.

Durante el proyecto hemos aprendido a conectar correctamente una interfaz web con una API REST, a proteger rutas, a organizar el código backend usando routers, y a controlar qué puede hacer cada usuario según su rol. Todo el trabajo se ha hecho de forma local y la base de datos se ha montado en MySQL.

## 2. ARQUITECTURA GENERAL DEL SISTEMA



Dos partes principales:

**API-REST**, que es el backend. La hemos hecho con FastAPI, y es la encargada de conectarse a la base de datos, gestionar los cursos, los usuarios, las inscripciones, las solicitudes y controlar los roles. Esta parte funciona en el puerto 8000.

**APP-WEB** está hecha con Flask y es la parte visual de la aplicación, lo que el usuario ve y con lo que interactúa. En esta parte se encuentran las plantillas HTML que hemos diseñado y todo el código JavaScript que se encarga de pedir datos al backend. Esta parte corre en el puerto 5000.

Todo esto está conectado a una base de datos **MySQL** que guarda la información de los usuarios, los cursos, las inscripciones y las solicitudes. La base de datos se ejecuta en local.

### 3. DISEÑO DE LA BASE DE DATOS

La base de datos que hemos utilizado se llama academia, y está creada en MySQL Workbench local. En ella hemos definido las tablas necesarias para gestionar los cursos, usuarios, roles e inscripciones.

Tablas Principales:

**USUARIOS:** almacena todos los datos de los usuarios registrados. Incluye su nombre, apellidos, email, contraseña, fecha de registro, tipo de rol, y si están habilitados o no.

	id	nombre	apellidos	email	password	tipo	fecha_creacion	fecha_modificacion	habilitado	id_rol
▶	7	Laura	Martínez	laura@academia.com	1234	admin	2025-05-02 09:34:43	2025-05-06 13:58:32	0	2
	8	claudia	corona	claudia@gmail.com	\$2b\$12\$9rRGUy3mkGbvGtH/L1stHVOnYjAGEqSt...	admin	2025-05-04 15:33:11	2025-05-05 14:43:47	1	1
	9	diego	corona	claudia12@gmail.com	\$2b\$12\$41SL2HctUQVDvp/UMnOA2uvNoXq431...	user	2025-05-05 09:04:30	2025-05-06 13:58:49	1	3
	10	lucia	blanco	lucia@gmail.com	\$2b\$12\$RxtIeCgS9aWwxK5ONSS4s.rR3nWxbL...	user	2025-05-05 09:56:10	2025-05-06 13:58:50	1	3
	11	lucia	corona	jose122@gmail.com	\$2b\$12\$NAKYBaP4YThz9TFFB2jmuQRGaZL5I24...	admin	2025-05-05 14:09:23	2025-05-05 14:43:11	1	2
	12	diego	blanco	ccorobla@myuax.es	\$2b\$12\$7M3KGWkXMcQeQGjIiXvno.sdhbXt.Ear...	admin	2025-05-05 14:40:22	2025-05-05 14:43:12	1	2
	13	julia	luis	julia@gmail.com	\$2b\$12\$OwIjQhaxoTbiSkfmP21eR.oOB/WayGf...	admin	2025-05-05 14:42:28	2025-05-05 14:43:52	1	2
	14	marta	pamies	marta@gmail.com	\$2b\$12\$ohj2J.yblgJTqxwH8DPS8OfuIOrFbdUZ...	user	2025-05-05 14:50:04	2025-05-05 14:50:04	1	1

**ROLES:** Definimos los 3 tipos de roles que tenemos y el id que le corresponde.

	id	nombre
▶	2	admin
	1	super
	3	user

**CURSOS:** contiene información de los cursos: nombre, descripción, duración y si están disponibles o no para los usuarios.

id	nombre	descripcion	duracion	disponible
1	Curso de Python	Aprende Python desde cero con ejercicios práct...	20	1
2	Curso de JavaScript	Domina JS moderno con proyectos reales	30	1
3	Curso de JavaScript Avanzado	Incluye proyectos con React y Node.js	40	1
4	Curso de Python	Aprende Python desde cero con ejercicios práct...	20	1

**USUARIOS\_CURSOS:** es la tabla intermedia que relaciona los cursos con los usuarios. Representa las inscripciones, es decir, qué usuario está en qué curso.

id	id_usuario	id_curso	fecha_inscripcion
1	8	6	2025-05-06 09:16:17
2	19	1	2025-05-06 13:59:10

**SOLICITUDES:** propuestas de nuevos cursos hechas por usuarios.

id	nombre	descripcion	duracion	id_usuario
2	Curso de EXCEL	Curso de Excel básico	10	10
3	Curso de python	Curso de python básico	30	10
4	Curso de python	Curso de python básico	30	10

## 4. BACKEND: API-REST (FASTAPI)

El backend de nuestro proyecto ha sido desarrollado con FastAPI, un framework que permite construir APIs de forma rápida, clara y eficiente. Esta parte del sistema actúa como la lógica central de la aplicación, gestionando todo lo relacionado con usuarios, roles, autenticación, cursos, inscripciones y solicitudes.

Cada archivo tiene una función específica: conexión con la base de datos, definición de modelos, validación de datos, autenticación, control de permisos...

A través de esta API, la aplicación web puede enviar y recibir datos mediante peticiones HTTP (como GET, POST, PUT), comunicándose de forma segura gracias a la autenticación basada en tokens JWT. Además, cada endpoint está protegido según el rol del usuario (user, admin o super), lo que garantiza un control adecuado de permisos.

A continuación, explicamos cómo se estructura internamente el backend, archivo por archivo, y después detallamos los endpoints disponibles.

### - DATABASE.PY

La conexión entre la API y la base de datos la hicimos en el archivo database.py. Aquí usamos SQLAlchemy. Para mayor seguridad, configuramos todo a través de variables de entorno usando la librería dotenv, de modo que nunca ponemos el usuario, la contraseña ni la dirección del servidor directamente en el código.

Primero, cargamos las variables desde un archivo .env, luego, con esas variables formamos la DATABASE\_URL, que es la cadena que SQLAlchemy necesita para conectarse con MySQL. En nuestro caso, usamos el driver pymysql.

Una vez tenemos eso, creamos el engine, que es como el motor que abre la conexión real, y luego definimos SessionLocal, que es la clase que usamos para crear sesiones de base de datos.

Además, definimos una función get\_db() que usamos como dependencia en nuestros endpoints. Esta función nos asegura que cada vez que se llame a la base de datos, se cree una sesión, se use y luego se cierre correctamente, evitando fugas de conexión.

Por último, declaramos Base = declarative\_base(), que es la clase base sobre la que luego construiremos todos nuestros modelos (tablas). Todas las tablas que creamos en models.py heredan de esta base.

## - MODELS.PY

Aquí es donde definimos todas las tablas de la base de datos usando SQLAlchemy. Cada clase que aparece en este archivo representa una tabla, y cada atributo corresponde a una columna.

Primero, definimos la tabla de roles (Rol), que contiene los diferentes tipos de usuario: "user", "admin" y "super".

Después, pasamos al modelo Usuario, que representa a cada persona registrada en la plataforma. Esta tabla contiene campos básicos como nombre, apellidos, email, contraseña, el tipo de usuario, si está habilitado o no, y dos fechas: una de creación y otra de modificación.

Además, cada usuario tiene un `id_rol`, que lo vincula con la tabla de roles.

El modelo Usuario también tiene dos relaciones muy importantes:

- `solicitudes`: que enlaza con la tabla `SolicitudCurso`, es decir, los cursos que ese usuario ha propuesto.
- `cursos_impartidos`: que conecta con los cursos donde el usuario actúa como profesor.

A continuación, definimos la tabla `Curso`, que guarda todos los datos de cada curso ofrecido en la academia: nombre, descripción, duración, si está disponible o no, y las plazas disponibles. Además, cada curso puede estar asociado a un profesor mediante `id_profesor`, que es una clave foránea que apunta a la tabla usuarios.

Luego viene la tabla intermedia `Inscripcion`, que representa la relación entre usuarios y cursos: un usuario puede estar en varios cursos, y un curso puede tener muchos alumnos. Aquí también añadimos la fecha de inscripción, que se guarda automáticamente con `datetime.utcnow`.

Por último, tenemos la tabla `SolicitudCurso`, que guarda las propuestas de cursos nuevos enviadas por los usuarios. Cada solicitud tiene un nombre, una descripción, duración, y está asociada a un `id_usuario`, que es quien la ha propuesto.

Todas las clases heredan de `Base`, que definimos antes en `database.py`, lo que permite a SQLAlchemy mapearlas directamente como tablas en la base de datos.



## - SCHEMAS.PY

Una vez definidas las tablas en `models.py`, lo siguiente que hicimos fue crear los esquemas de validación en `schemas.py`. Aquí usamos la librería `Pydantic`, que es la que utiliza `FastAPI` para comprobar que los datos que recibimos o devolvemos cumplen el formato esperado.

Empezamos creando algunos enums para los roles y tipos de usuario. También definimos `TipoUsuario`, aunque ese lo usamos más como etiqueta informativa (estudiante, profesor, otro).

Después, creamos los esquemas relacionados con los usuarios. Por un lado, está `UsuarioCreate`, que usamos cuando alguien se registra; este modelo exige nombre, apellidos, email, contraseña y un rol. Por otro lado, tenemos `UsuarioOut`, que es el formato que usamos cuando devolvemos los datos de un usuario. En este modelo no incluimos la contraseña por seguridad, y activamos `from_attributes = True`.

En cuanto a la parte de autenticación, definimos varios esquemas:

- `LoginRequest`, que contiene el email y contraseña del usuario al hacer login.
- `Token`, que representa el token JWT que devolvemos si el login es exitoso.
- `TokenData`, que usamos internamente para extraer información del token, como el sub (identificador del usuario) y su rol.
- `TokenResponse`, una variante mejorada que incluye también el nombre del rol, para que el frontend sepa qué tipo de usuario ha iniciado sesión.

Luego vienen los esquemas relacionados con los cursos. Tenemos `CursoCreate` para crear nuevos cursos, `CursoUpdate` para modificarlos, y `CursoOut` para devolver la información al cliente.

Al final añadimos algunos esquemas más concretos como `UsuarioUpdate`, para modificar datos del perfil, y `ProfesorOut`, que usamos cuando devolvemos una lista de profesores disponibles.

## - DEPENDENCIES.PY

Aquí definimos todas las funciones comunes relacionadas con la autenticación y la autorización, que luego reutilizamos en distintos routers.

La parte más importante de este archivo es la función `get_current_user()`. Esta función se encarga de leer el token JWT que llega en la cabecera de cada petición (`Authorization: Bearer <token>`), decodificarlo y verificar que sea válido. Si el token es correcto, busca el usuario correspondiente en la base de datos y lo devuelve.

Gracias a esto, podemos proteger cualquier endpoint simplemente añadiendo `Depends(get_current_user)` como parámetro. Esto nos garantiza que solo podrán acceder usuarios que hayan iniciado sesión correctamente.

Además, para gestionar los permisos según el rol del usuario, definimos dos funciones más, `verificar_rol_super`, que comprueba que el usuario tenga rol super (`id_rol = 1`) y `verificar_rol_admin`, que acepta tanto admin como super (`id_rol = 1` o `2`).

Estas funciones se usan como dependencias adicionales en las rutas que requieren permisos especiales, y lanzan un error 403 si el usuario no tiene el rol adecuado. También configuramos el esquema `OAuth2PasswordBearer`, que es el que FastAPI usa por defecto para interpretar los tokens JWT. Además, añadimos un esquema alternativo `HTTPBearer`, que es útil si queremos recibir el token manualmente sin formulario.

Por último, añadimos un ejemplo de ruta protegida (`/profile`), que demuestra cómo usar todo esto: primero se verifica el token con `get_current_user`, y luego se responde con datos del usuario. Esta ruta la usamos sobre todo como prueba para ver si la autenticación está funcionando correctamente.

## - AUTH.PY:

Una vez que ya teníamos lista la conexión con la base de datos, los modelos y las funciones de seguridad, empezamos a construir los **routers**, que son los archivos donde definimos los distintos **endpoints** de la API. El primero que desarrollamos fue `auth.py`, que se encarga de la **autenticación de usuarios**.

Este archivo define el endpoint `/auth/login`, que es la ruta a la que los usuarios deben enviar su email y contraseña para iniciar sesión. La petición se hace mediante un POST, y el cuerpo debe seguir el formato que definimos en el esquema `LoginRequest`.

Lo primero que hace esta función es comprobar si existe un usuario con ese email en la base de datos. Si no existe, devuelve un error 401. Luego verifica que el usuario esté habilitado, y finalmente compara la contraseña introducida con la almacenada, usando la librería `passlib` para trabajar con hashes seguros (`bcrypt`).

Si todo está correcto, el backend genera un token JWT que contiene el ID del usuario (`sub`), su rol (`rol`), y una fecha de expiración. El token se codifica con una clave secreta (`SECRET_KEY`) y el algoritmo `HS256`, y se devuelve al frontend junto con el tipo "bearer" y el nombre del rol. Este token será necesario para acceder al resto de rutas protegidas.

Gracias a este sistema, una vez que el usuario inicia sesión, ya no tiene que enviar su email y contraseña en cada petición. Solo necesita enviar el token en la cabecera Authorization.



## - **USUARIOS.PY:**

El router usuarios.py lo integramos también dentro del módulo auth, aunque aquí ya no tratamos solo autenticación, sino la gestión completa de los usuarios. Esta parte del backend es especialmente importante para los roles admin y super, ya que les permite consultar usuarios, habilitarlos, deshabilitarlos, ascenderlos a administradores, y más.

La primera función destacable aquí, además del login que ya explicamos antes, es el registro de usuarios (/auth/usuarios, método POST). En este punto establecimos una norma de seguridad importante: solo se pueden registrar usuarios normales (rol 3). Si alguien intenta registrarse con un rol de mayor nivel desde el frontend, la API lanza un error 403. Esto evita que se creen cuentas de administrador de forma externa.

Luego, incluimos un endpoint para que los admin y super puedan listar todos los usuarios registrados (GET /auth/usuarios), siempre que tengan el permiso necesario. Usamos la dependencia verificar\_rol\_admin, que controla si el usuario tiene rol 1 o 2.

Los usuarios también pueden ser habilitados o deshabilitados desde el panel de administración. Para eso creamos dos rutas tipo PUT, que modifican el campo habilitado del usuario. Estas acciones están limitadas al rol super, y son útiles, por ejemplo, si se detecta mal uso de la plataforma.

También añadimos funciones para asignar o quitar el rol de administrador a otros usuarios. Con /hacer-admin o /quitar-admin, un super puede escalar o degradar usuarios, actualizando tanto su id\_rol como su tipo.

Por otro lado, creamos la ruta /mis-cursos, que devuelve todos los cursos en los que está inscrito el usuario autenticado. Esta funcionalidad está disponible para cualquier usuario logueado (no requiere ser admin), y es la que usamos en la APP-WEB en la sección de “Mis cursos”.

Añadimos también `/lista-profesores`, que devuelve una lista filtrada con todos los usuarios cuyo tipo sea "profesor". Esto nos resulta útil, por ejemplo, al momento de asignar un profesor a un curso desde la interfaz de administrador.

Para que cada usuario pueda modificar su propio perfil, implementamos la ruta `/usuarios/editar-perfil`, que actualiza nombre, apellidos y correo electrónico. Esta ruta está protegida por `get_current_user`, de modo que solo afecta al usuario autenticado en ese momento.

Por último, incluimos una ruta para consultar todos los roles disponibles (`/roles`), que puede usar la app web para mostrar los nombres de roles en formularios o desplegables.

Todos estos endpoints están visibles en Swagger UI, agrupados bajo la etiqueta "auth" y ordenados con sus métodos (GET, POST, PUT). En la siguiente imagen puede verse parte de estas rutas:

auth		^
POST	/auth/login	Login
POST	/auth/usuarios	Registrar Usuario
GET	/auth/usuarios	Listar Usuarios
PUT	/auth/usuarios/{usuario_id}/deshabilitar	Deshabilitar Usuario
PUT	/auth/usuarios/{usuario_id}/habilitar	Habilitar Usuario
PUT	/auth/usuarios/{usuario_id}/hacer-admin	Hacer Admin
GET	/auth/mis-cursos	Obtener Mis Cursos
GET	/auth/roles	Obtener Roles
PUT	/auth/usuarios/{usuario_id}/quitar-admin	Quitar Admin
GET	/auth/lista-profesores	Obtener Profesores
PUT	/auth/usuarios/editar-perfil	Editar Perfil

## - COURSES.PY:

Uno de los routers más amplios del proyecto es `courses.py`, que agrupa toda la lógica relacionada con la gestión de cursos, incluyendo su creación, visualización, inscripción, asignación de profesores, y también algunas funciones relacionadas con solicitudes.

Lo primero que implementamos fue una función auxiliar llamada `get_current_user_optional`. Esta función es útil porque permite identificar al usuario si ha iniciado sesión, pero sin obligarlo a estar autenticado. Gracias a eso, pudimos hacer que los cursos públicos estén disponibles tanto para usuarios anónimos como para usuarios normales.

Luego definimos la ruta principal GET /cursos, que devuelve todos los cursos disponibles. Si quien consulta es un usuario admin o super, se le devuelven todos los cursos (estén habilitados o no). En cambio, si la consulta la hace un estudiante o alguien no logado, solo se devuelven aquellos cursos que están marcados como disponibles. En el resultado incluimos datos del profesor (si hay uno asignado) y calculamos plazas restantes restando las inscripciones actuales. Esta información se muestra en el frontend desde la página principal.

El endpoint POST / permite crear nuevos cursos, pero solo si el usuario es admin o super. Validamos esto con la dependencia verificar\_rol\_admin. Al crear un curso, es posible dejarlo sin profesor asignado y con un número determinado de plazas.

También incluimos funcionalidades para modificar (PUT) o eliminar (DELETE) un curso, y otras para habilitar o deshabilitarlo sin borrarlo físicamente, lo que permite pausar cursos temporalmente. Estas rutas son útiles desde el panel de administración.

La funcionalidad más relevante para los usuarios normales es sin duda POST /{curso\_id}/inscribirse, que permite inscribirse a un curso si hay plazas disponibles. Controlamos varias cosas aquí:

- Que el curso esté disponible.
- Que el usuario no esté ya inscrito.
- Que el tipo de usuario sea "estudiante" (evitamos que los admin se inscriban).
- Que haya plazas disponibles (y se descuenta una al inscribirse).

Además, creamos GET /mis-cursos para que cualquier usuario pueda consultar sus cursos actuales, y GET /{curso\_id}/inscritos, que devuelve los alumnos inscritos a un curso (solo accesible para admin o super).

Para mantener el control desde la administración, añadimos también la ruta DELETE /{curso\_id}/inscribir/{usuario\_id} que permite eliminar la inscripción de un alumno y devolver su plaza disponible, en caso de error o abandono.

Una funcionalidad muy interesante es POST /solicitar-curso, que permite a cualquier usuario enviar una propuesta de curso. Estas solicitudes se guardan en una tabla aparte (solicitudes), y en el futuro podrían convertirse en cursos reales si el administrador las aprueba.

También desarrollamos POST /{curso\_id}/asignar-profesor, que permite a un usuario con tipo "profesor" asignarse voluntariamente como docente del curso, siempre y cuando todavía no haya otro profesor asignado. Esta acción queda

restringida a los profesores, y se controla mediante su campo tipo y el `current_user`.

CURSOS		^
GET	/cursos/ Obtener Cursos	▼
POST	/cursos/ Crear Curso	▼
GET	/cursos/cursos/{curso_id} Obtener Curso Por Id	▼
PUT	/cursos/cursos/{curso_id} Actualizar Curso	▼
DELETE	/cursos/cursos/{curso_id} Eliminar Curso	▼
POST	/cursos/{curso_id}/inscribirse Inscribirse	▼
GET	/cursos/mis-cursos Obtener Mis Cursos	▼
GET	/cursos/{curso_id}/inscritos Ver Usuarios Inscritos	▼
PUT	/cursos/{curso_id}/deshabilitar Deshabilitar Curso	▼
PUT	/cursos/{curso_id}/habilitar Habilitar Curso	▼
GET	/cursos/disponibles Cursos Disponibles	▼
DELETE	/cursos/{curso_id}/inscribir/{usuario_id} Eliminar Inscripcion	▼
POST	/cursos/solicitar-curso Solicitar Curso	▼
POST	/cursos/cursos/{curso_id}/inscribir/{usuario_id} Inscribirse	▼
DELETE	/cursos/cursos/{curso_id}/inscribir/{usuario_id} Eliminar Inscripcion	▼
POST	/cursos/{curso_id}/asignar-profesor Asignar Profesor	▼
POST	/cursos/cursos/{curso_id}/asignar-profesor Asignar Profesor	▼
GET	/cursos/cursos/ Obtener Cursos	▼
POST	/cursos/cursos/ Crear Curso	▼
GET	/cursos/cursos/cursos/{curso_id} Obtener Curso Por Id	▼
PUT	/cursos/cursos/cursos/{curso_id} Actualizar Curso	▼
DELETE	/cursos/cursos/cursos/{curso_id} Eliminar Curso	▼
POST	/cursos/cursos/{curso_id}/inscribirse Inscribirse	▼
GET	/cursos/cursos/mis-cursos Obtener Mis Cursos	▼
GET	/cursos/cursos/{curso_id}/inscritos Ver Usuarios Inscritos	▼
PUT	/cursos/cursos/{curso_id}/deshabilitar Deshabilitar Curso	▼
PUT	/cursos/cursos/{curso_id}/habilitar Habilitar Curso	▼
GET	/cursos/cursos/disponibles Cursos Disponibles	▼
POST	/cursos/cursos/solicitar-curso Solicitar Curso	▼
POST	/cursos/cursos/cursos/{curso_id}/inscribir/{usuario_id} Inscribirse	▼
POST	/cursos/cursos/cursos/{curso_id}/asignar-profesor Asignar Profesor A Curso	▼

## - SOLICITUDES.PY:

Esta funcionalidad permite a los usuarios sugerir cursos que les gustaría ver en la plataforma, y queda registrada en la base de datos para que los administradores puedan revisarla más adelante.

En primer lugar, definimos el esquema `SolicitudCreate`, que especifica los campos necesarios para crear una solicitud: nombre del curso, descripción y duración. Este esquema se usa para validar que los datos enviados desde el frontend son correctos.

La ruta principal de este módulo es `POST /solicitudes`, que permite crear una nueva solicitud. Está protegida con `get_current_user`, así que solo pueden usarla usuarios autenticados. Una vez se recibe la información, se guarda en la tabla `solicitudes` junto con el `id_usuario` de quien la envió. Esta información se puede usar luego para mostrar quién propuso cada idea.

También añadimos una ruta `GET /solicitudes`, que devuelve todas las solicitudes registradas. Esta ruta está restringida a usuarios con rol `admin` o `super`, gracias a la dependencia `verificar_rol_admin`, ya que solo ellos pueden gestionar las propuestas y decidir si las aprueban o no.

solicitudes		^	
GET	/solicitudes/	Obtener Solicitudes	▼
POST	/solicitudes/	Crear Solicitud	▼

## - MAIN.PY:

Para finalizar la parte del backend, en `main.py` reunimos todos los elementos necesarios para lanzar la aplicación FastAPI y ponerla en marcha. Este archivo es el punto de entrada del backend, y en él conectamos los routers, configuramos la seguridad, y activamos herramientas esenciales como CORS o la inicialización de roles.

Lo primero que hacemos es crear la instancia de la aplicación con `app = FastAPI()`. Luego configuramos el **middleware CORS**, que nos permite aceptar peticiones desde otros orígenes —en este caso, desde el puerto 5000, donde corre la APP-WEB en Flask.

Después, ejecutamos `models.Base.metadata.create_all(bind=engine)`, que es la función que crea todas las tablas en la base de datos si no existen aún. Justo a continuación llamamos a `crear_roles_base()`, una función que insertamos para asegurarnos de que los tres roles principales (`super`, `admin`, `user`) existen siempre que se arranca la app. Esta función revisa si los roles ya están definidos y, si no, los añade.

Luego incluimos todos los routers de la aplicación: `auth`, `usuarios`, `courses` y `solicitudes`. Esto hace que las rutas definidas en cada uno queden activas y disponibles en Swagger UI y desde el frontend. También usamos etiquetas y prefijos para organizarlos correctamente, como por ejemplo `prefix="/cursos"`.

A continuación, definimos dos rutas generales:

- GET /: la ruta raíz, que simplemente devuelve un mensaje de bienvenida para comprobar que el backend está funcionando correctamente.
- GET /usuarios/me: una ruta protegida que devuelve los datos del usuario actual, siempre que se haya enviado un token JWT válido.

default		^
GET	/ Read Root	
GET	/usuarios/me Leer Usuario Actual	



## 5. FRONTEND: APP-WEB (FLASK + PLANTILLAS HTML)

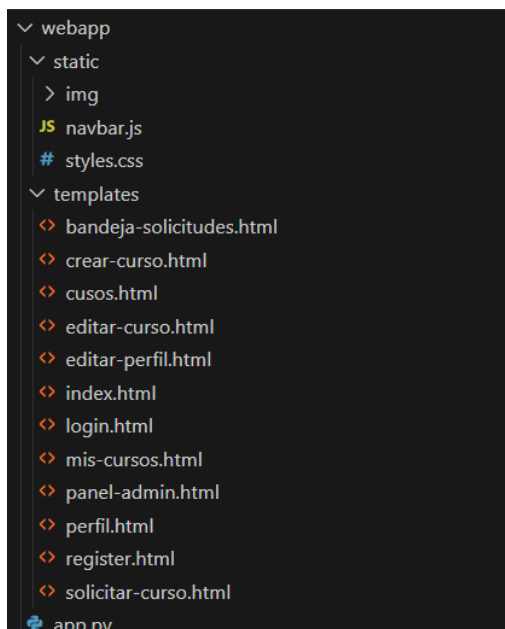
Para el desarrollo del frontend de nuestra aplicación web hemos utilizado Flask en combinación con plantillas HTML, JavaScript y CSS. La estructura general del proyecto se divide principalmente en dos carpetas: static y templates, cada una con un propósito claro y definido.

- **La carpeta templates/** Contiene todas las plantillas HTML que se renderizan desde Flask. Aquí es donde está el corazón visual de la aplicación. Cada archivo representa una vista concreta:
  - index.html: página de inicio que lista todos los cursos.
  - login.html y register.html: vistas públicas para iniciar sesión o registrarse.
  - perfil.html, editar-perfil.html: secciones del usuario para gestionar sus datos.
  - crear-curso.html, editar-curso.html: vistas disponibles solo para administradores y superadmins.
  - solicitar-curso.html, bandeja-solicitudes.html: donde los usuarios pueden proponer nuevos cursos y los admins gestionarlos.
  - panel-admin.html: vista exclusiva para superadmins, donde gestionan todos los usuarios del sistema.
  - mis-cursos.html: para que los estudiantes vean en qué cursos están inscritos.
  - cursos.html: parece un duplicado del index, pero lo confirmaremos más adelante.
- **La carpeta static/** incluye los recursos estáticos como estilos CSS (styles.css), scripts JavaScript (navbar.js) y una carpeta img/ para imágenes (como logotipos o ilustraciones usadas en las cards de cursos). Esta

carpeta es accesible desde cualquier plantilla HTML y permite mantener separada la lógica de diseño respecto al contenido dinámico.

El archivo principal `app.py` se encarga de gestionar las rutas de la aplicación, renderizando las plantillas adecuadas en función del endpoint accedido, y pasando los datos necesarios (como el tipo de usuario, los cursos, o el estado de sesión) para que el contenido pueda adaptarse dinámicamente a cada caso.

Esta estructura modular facilita el mantenimiento del código, permite la reutilización de componentes, y hace posible que cada parte del frontend responda directamente a la lógica de negocio definida en el backend.



Toda la lógica de visualización parte de Flask, que utiliza Jinja2 para renderizar las plantillas. Sin embargo, hemos optado por un enfoque mixto: algunas vistas están renderizadas dinámicamente en el navegador mediante JavaScript (principalmente llamadas a la API REST que devuelven JSON), y otras se renderizan desde el servidor.

La navegación y el acceso a cada página depende del rol del usuario, gracias a `navbar.js`. Este archivo detecta el token JWT almacenado en el `localStorage`, llama al endpoint `/usuarios/me` de la API y genera el menú de navegación con los enlaces correspondientes a su tipo:

- **User:** puede ver cursos, solicitarlos, y editar su perfil.

- **Admin:** además de lo anterior, puede crear cursos y ver solicitudes.
- **Superadmin:** accede a todo lo anterior más el panel de administración y la gestión de usuarios.

Una vez obtenida una visión general de nuestra estructura

## - INDEX.HTML

La vista principal de nuestra aplicación es index.html, que actúa como punto de entrada para todos los usuarios, estén o no autenticados. Desde esta interfaz se muestra el catálogo de cursos disponibles en la plataforma, y se adapta en función del rol del usuario que accede (visitante, estudiante, profesor, admin o superadmin).

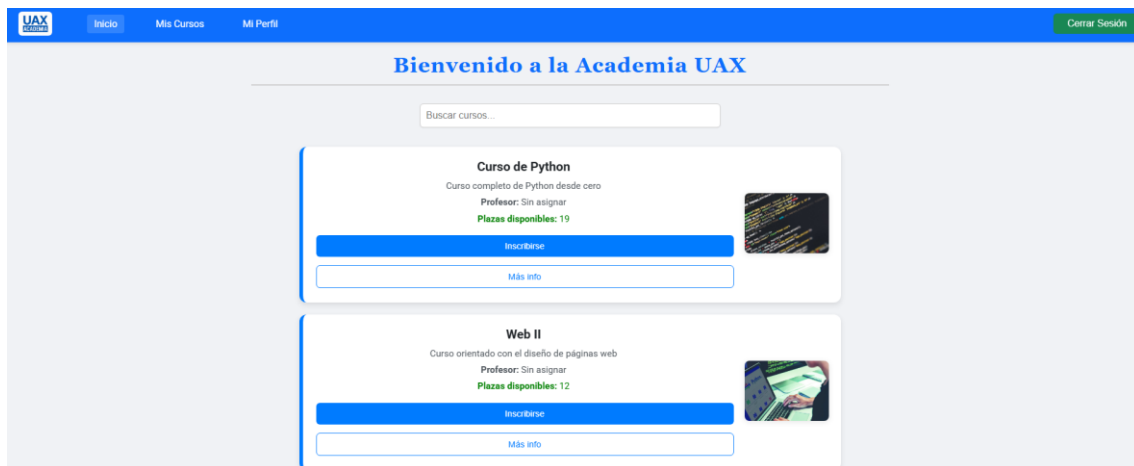
Nada más cargar esta vista, se ejecuta el script navbar.js que se encarga de pintar dinámicamente la barra de navegación según el rol del usuario. Esto nos permite que la misma plantilla sirva para todos los casos, personalizando la navegación en tiempo real.

Dentro del contenido principal (<main>), presentamos un título de bienvenida y un campo de búsqueda que permite filtrar los cursos en tiempo real. Esta funcionalidad es puramente cliente y se ejecuta con JavaScript, sin necesidad de llamadas adicionales al servidor. Posteriormente, una vez cargado el contenido, se realiza una petición a la API (/cursos) para obtener el listado de cursos en formato JSON.

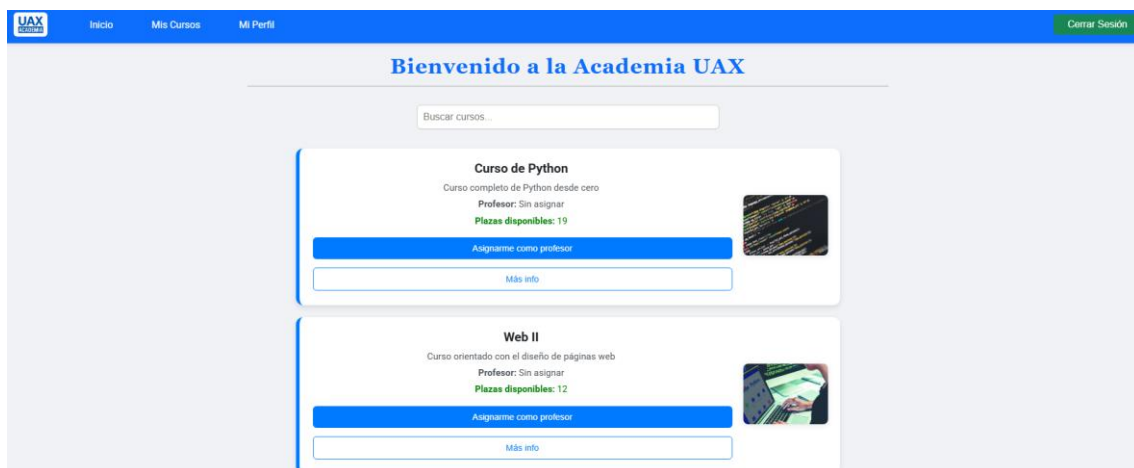
La respuesta se recorre dinámicamente para generar tarjetas (cards) con la información básica del curso: nombre, descripción, profesor asignado (si lo hay) e imagen ilustrativa. Para mejorar la experiencia de usuario, el diseño incorpora sombras, bordes redondeados y diferenciación visual de los cursos con pocas plazas.

Además, el sistema adapta los botones disponibles según el rol:

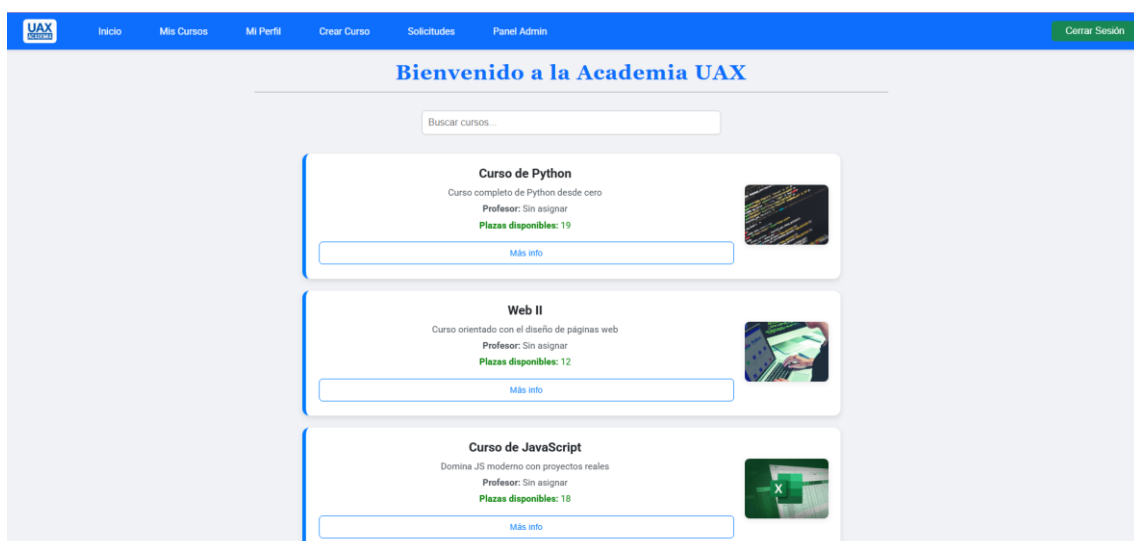
- Estudiantes pueden inscribirse directamente en el curso.



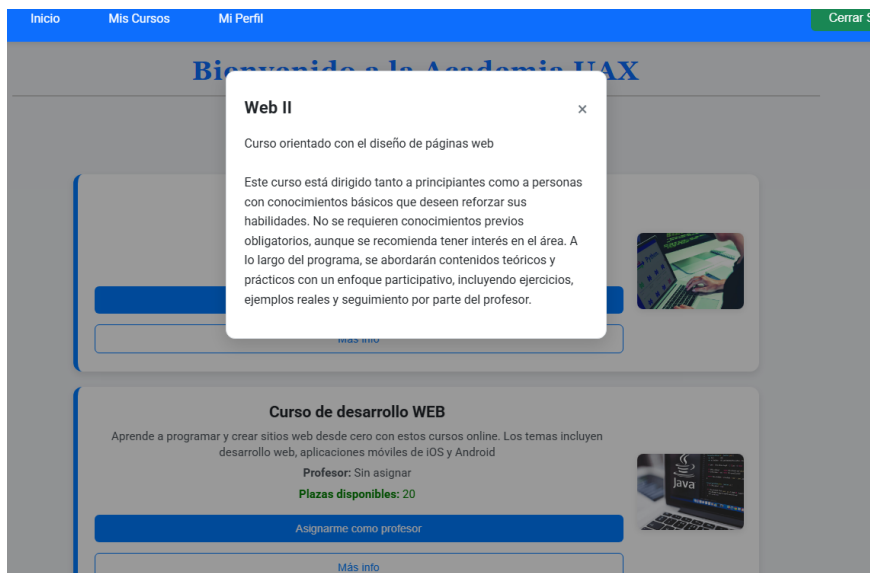
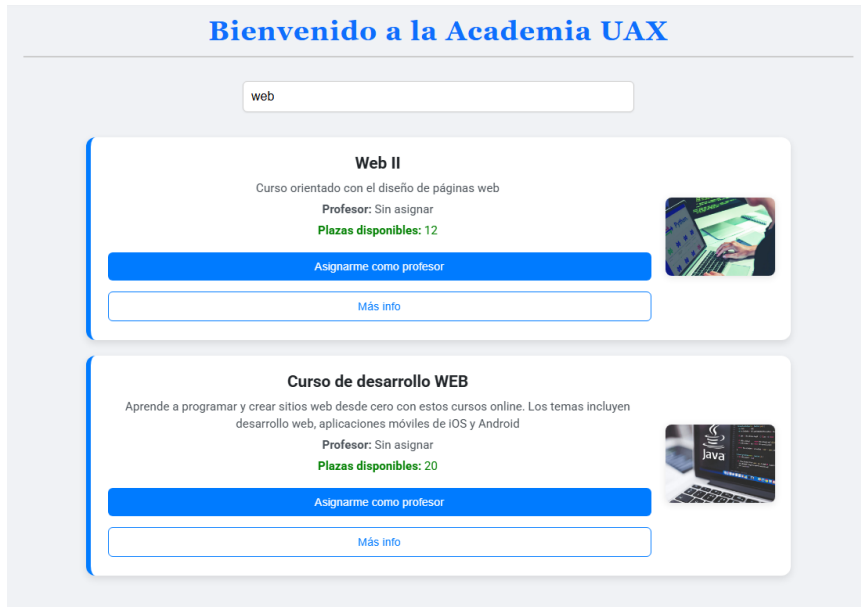
- Profesores pueden asignarse como responsables si el curso no tiene docente.



- Admins y superadmins tienen vistas específicas para gestionar o editar cursos, accesibles desde otras rutas.

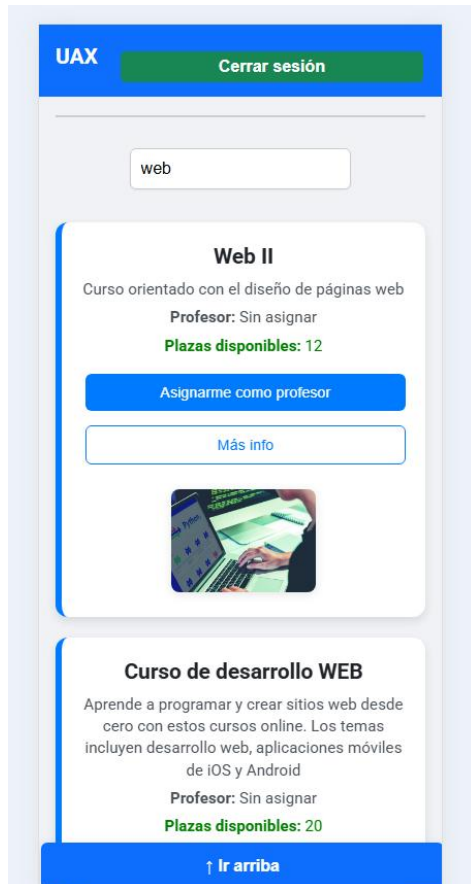


Cada tarjeta incluye un botón “Más info” que abre un modal emergente con una descripción extendida del curso. Esta descripción está pensada para simular información complementaria útil como requisitos, objetivos y nivel de dificultad. Hemos añadido una cadena extra ficticia para que el texto sea más largo y visualmente más atractivo, mejorando la experiencia en dispositivos grandes.



Para los dispositivos móviles, se ha implementado una barra fija que solo aparece al hacer scroll. Esta barra simplificada muestra el logo y el botón de cierre de sesión. A su vez, también se incluye un botón flotante en la parte inferior derecha para permitir al usuario volver rápidamente al principio de la página.

Toda la lógica de esta vista reside en el propio HTML, complementada con JavaScript puro que interactúa directamente con la API mediante fetch, sin necesidad de recargar la página, asegurando así una experiencia fluida y moderna.



## - Formularios de autenticación (login.html y register.html)

Para facilitar el acceso personalizado a la plataforma, desarrollamos dos páginas esenciales dentro de nuestro frontend: una dedicada al **inicio de sesión** y otra al **registro de nuevos usuarios**. Ambas siguen una estética coherente con el resto del sitio y están adaptadas para dispositivos móviles y escritorio.

### - Login (login.html)

La vista de inicio de sesión es sencilla, limpia y centrada visualmente. Incluye dos campos: email y contraseña, así como un botón para enviar la solicitud. Está diseñada para funcionar de forma dinámica mediante JavaScript, evitando recargas innecesarias.

Al hacer clic en el botón "Entrar", se lanza una petición POST al endpoint /auth/login de nuestra API. Si las credenciales son válidas, almacenamos el token JWT en el localStorage del navegador y redirigimos

automáticamente al usuario al índice. En caso de error, se muestra un mensaje contextualizado directamente bajo el formulario.

The screenshot shows the login page of the UAX platform. At the top, there is a blue header with the UAX logo on the left and three links: 'Inicio', 'Iniciar sesión', and 'Registrarse'. The main content area is light gray and contains a white box titled 'Iniciar Sesión'. Inside this box, there are two input fields: 'Email' and 'Contraseña'. Below these fields is a green button labeled 'Entrar'.

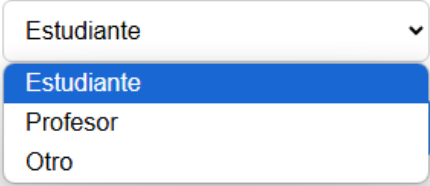
### - Registro (register.html)

La página de registro presenta un formulario algo más extenso, que solicita nombre, apellidos, email, contraseña y tipo de usuario (estudiante, profesor u otro). Hemos definido un id\_rol por defecto que se asigna automáticamente como 3 (rol estándar "user"), permitiendo una futura gestión por parte de los administradores.

Una vez completado el formulario, se envía una petición POST a /auth/usuarios. Si la operación es exitosa, se lanza de forma automática un login con los mismos datos, simulando una experiencia fluida de incorporación a la plataforma.

Por último, ambos documentos incluyen también la mini-navbar flotante para móviles y el botón de scroll para volver arriba, que permanecen consistentes en toda la aplicación y mejoran la accesibilidad.

The screenshot shows the registration page of the UAX platform. At the top, there is a blue header with the UAX logo on the left and three links: 'Inicio', 'Iniciar sesión', and 'Registrarse'. The main content area is light gray and contains a white box titled 'Registro de Usuario'. Inside this box, there are five input fields: 'Nombre', 'Apellidos', 'Email', and 'Contraseña'. Below these fields is a dropdown menu labeled 'Tipo de usuario' with 'Estudiante' selected. At the bottom of the box is a blue button labeled 'Registrarse'.



**Tipo de usuario**

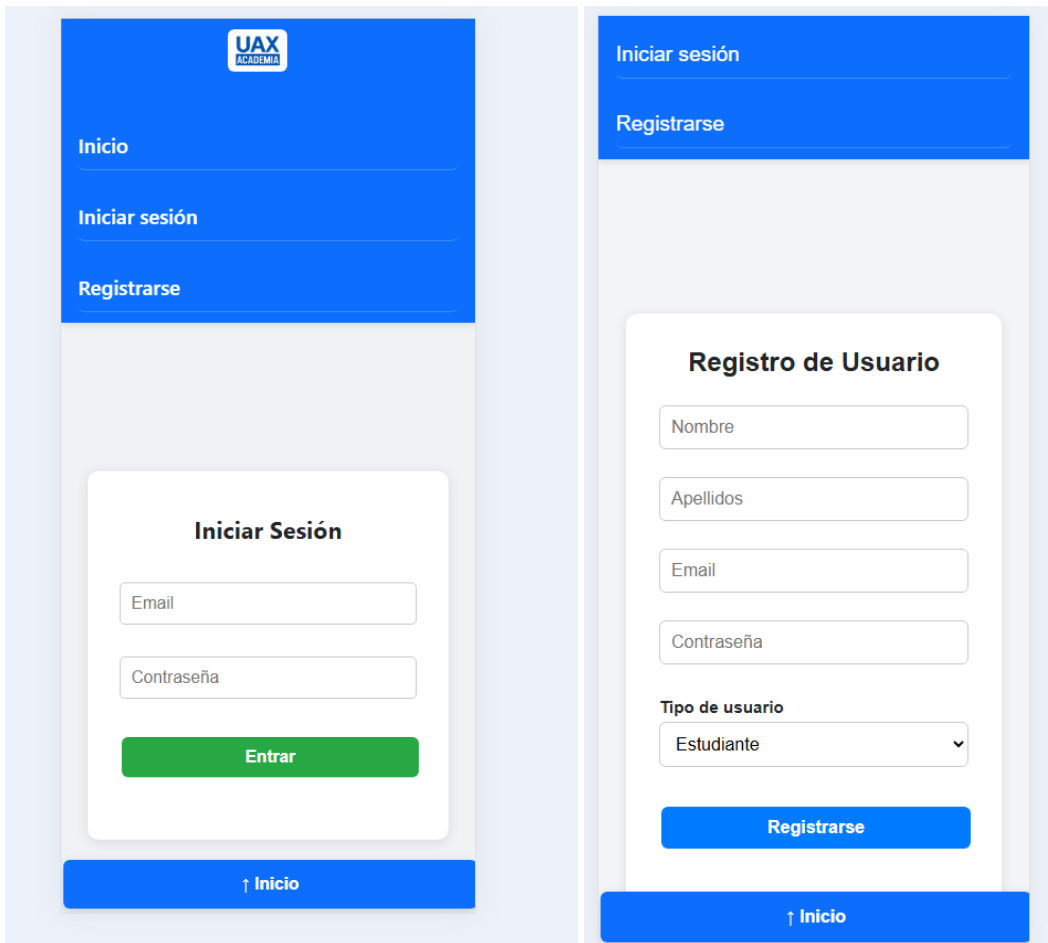
Estudiante ▼

Estudiante

Profesor

Otro

Como se ve desde el responsive:



The image displays two mobile app screens side-by-side. The left screen is the login page, featuring a blue header with the UAX ACADEMIA logo and navigation links: Inicio, Iniciar sesión, and Registrarse. Below the header is a white card titled 'Iniciar Sesión' with input fields for Email and Contraseña, a green Entrar button, and a bottom blue bar with an '↑ Inicio' link. The right screen is the registration page, with a similar blue header and navigation links. It features a white card titled 'Registro de Usuario' with input fields for Nombre, Apellidos, Email, and Contraseña, a 'Tipo de usuario' dropdown menu (set to Estudiante), a blue Registrarse button, and a bottom blue bar with an '↑ Inicio' link.

## - Mis-cursos.html

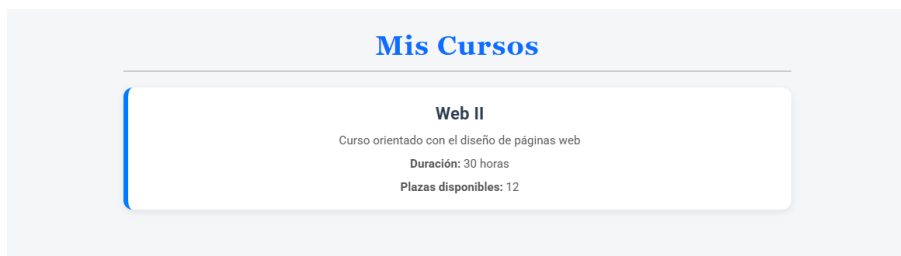
Una vez autenticado, el usuario puede acceder a su espacio personal a través de la opción “Mis Cursos”, visible en el menú de navegación. Esta vista es clave para centralizar la experiencia del usuario en nuestra plataforma, ya que desde aquí puede ver en qué cursos está inscrito y, dependiendo de su rol, también gestionarlos.



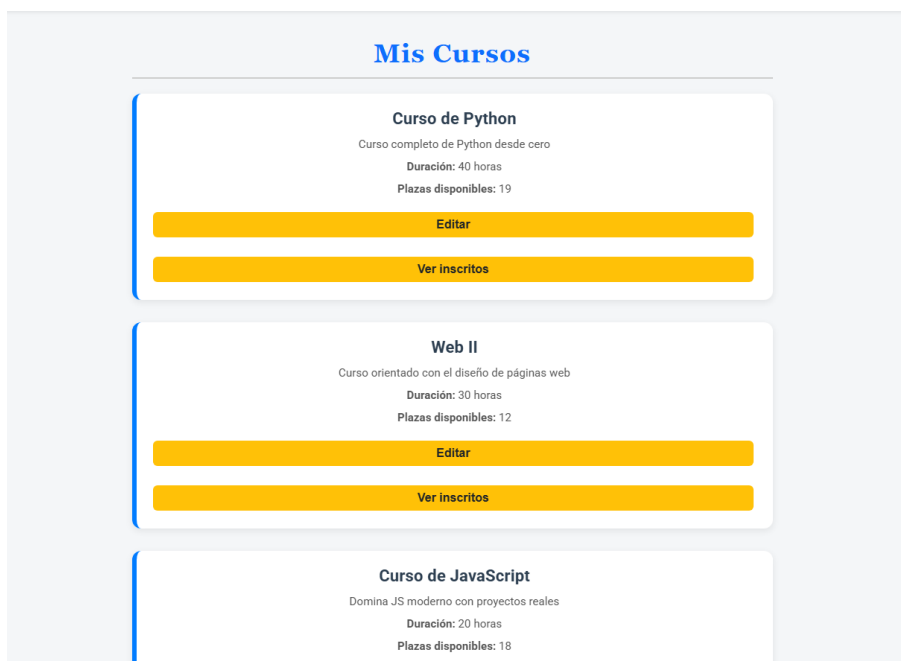
## Diferenciación por rol

Una de las principales funcionalidades que aplicamos aquí es la personalización del contenido según el **tipo de usuario**:

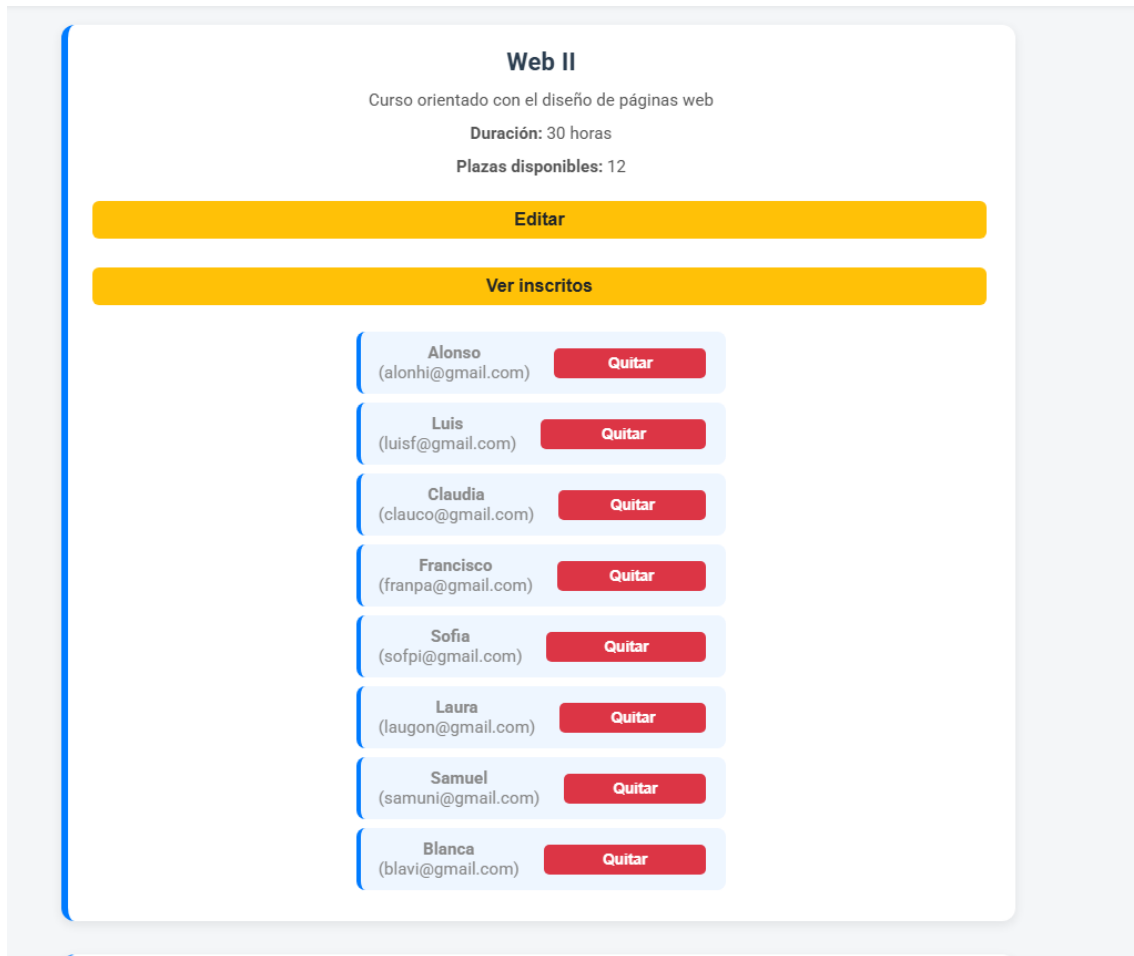
- Si el usuario tiene rol de **estudiante**, simplemente se le muestran sus cursos con información relevante como el nombre, descripción, duración y plazas disponibles.



- Si el usuario es **admin** o **superadmin**, se le presentan **todos los cursos** disponibles en el sistema (no solo aquellos en los que está implicado), y además se le habilitan botones para **editar cursos** y **ver inscritos**.



Desde el botón “Ver inscritos”, los administradores pueden desplegar una lista con los alumnos inscritos en cada curso, y se les permite eliminar inscripciones si es necesario.



Además, los bloques se cargan dinámicamente usando JavaScript, evitando recargas innecesarias.

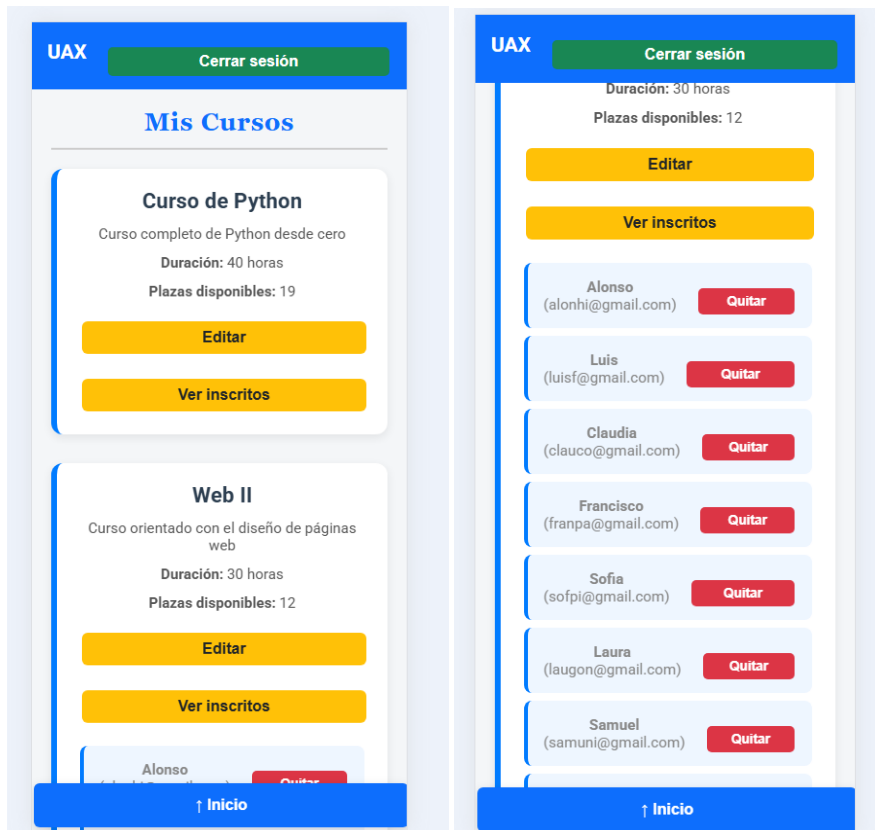
Cuando el usuario entra en esta página, se realiza una doble petición:

1. **GET a /usuarios/me** para conocer el rol del usuario.
2. **GET a /cursos/mis-cursos** o a **/cursos** dependiendo del rol, para cargar la lista correspondiente.

El contenido se inserta en el DOM mediante innerHTML, generando dinámicamente tarjetas (.curso-card) para cada curso. Los botones “Editar” y “Ver inscritos” sólo aparecen para administradores, reforzando el control de acceso desde el frontend.

En caso de seleccionar “Ver inscritos”, se ejecuta una nueva llamada a **/cursos/:id/inscritos**, y se genera dinámicamente un bloque .alumno-card por cada usuario inscrito. Para mejorar la experiencia, el sistema alterna la visibilidad del listado en cada clic y muestra mensajes contextualizados si no hay alumnos registrados.

Así es como se ve en modo responsive:



- editar-curso.html

Desde la vista de “Mis Cursos”, los administradores tienen la posibilidad de acceder a una funcionalidad específica para editar o eliminar un curso ya existente. Esta acción se activa al pulsar el botón “Editar” correspondiente a un curso, el cual redirige a esta página utilizando un parámetro id en la URL (editar-curso?id=...).

La pantalla de edición permite modificar los siguientes campos de un curso:

- **Nombre del curso**
- **Descripción**
- **Duración en horas**
- **Estado de disponibilidad** (disponible / no disponible)

Además, se ofrecen dos botones importantes:

- **Guardar Cambios:** actualiza los datos del curso mediante una petición PUT al endpoint correspondiente (/cursos/cursos/{id}).
- **Eliminar Curso:** permite eliminar permanentemente un curso mediante una petición DELETE.

También se incluye un botón para **volver atrás** (← Volver), facilitando la navegación dentro del panel del administrador.

### Flujo de datos

Nada más cargarse la vista, se lanza una petición GET para traer los datos actuales del curso y rellenar los campos del formulario. Todo esto depende del parámetro id que se recibe desde la URL. Si falta el token o no se proporciona un ID, se muestra un mensaje de error preventivo.

Una vez que el usuario ha realizado cambios, se recopilan todos los datos del formulario y se envían al servidor. Si el proceso se completa correctamente, se muestra un mensaje de confirmación; en caso de error, se informa adecuadamente en pantalla sin necesidad de recargar la página.

Este formulario es exclusivo para usuarios con permisos de tipo **admin** o **superadmin**, ya que está enlazado directamente desde el módulo de gestión de cursos. El servidor también realiza su propia validación para evitar que usuarios no autorizados accedan a esta ruta mediante manipulación directa de la URL.

The screenshot shows the 'Editar Curso' (Edit Course) form. The form is titled 'Editar Curso' in blue. It contains four input fields: 'Curso de Python', 'Curso completo de Python desde cero', '40', and 'Disponible'. Below the input fields are three buttons: a blue 'Guardar Cambios' button, a red 'Eliminar Curso' button, and a white button with a blue border and text '← Volver'. The top navigation bar is blue with links: Inicio, Mis Cursos, Mi Perfil, Crear Curso, Solicitudes, and a green 'Cerrar Sesión' button on the right.

### - perfil.html y editar-perfil.html

Dentro de nuestra aplicación web, una vez que el usuario ha iniciado sesión correctamente, tiene la posibilidad de consultar sus datos personales accediendo al apartado **Mi Perfil**, disponible desde la barra de navegación superior. Esta funcionalidad se encuentra implementada en la plantilla perfil.html.

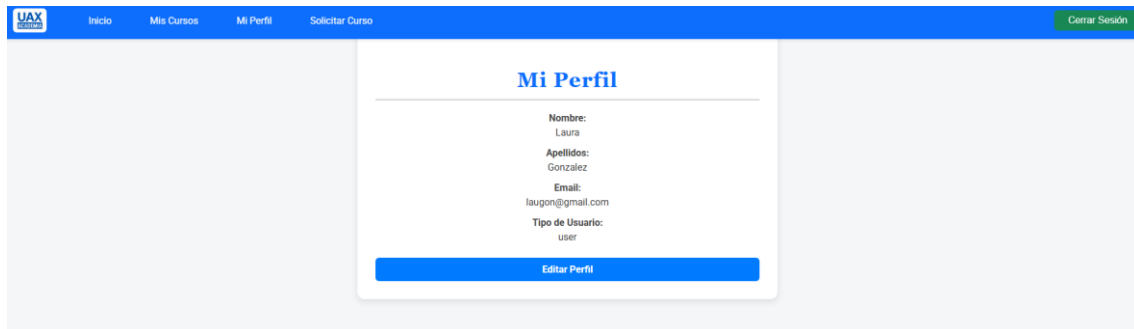
En esta vista mostramos de forma clara y ordenada los datos básicos del usuario:

- Nombre
- Apellidos
- Dirección de correo electrónico

- Rol de usuario (estudiante, profesor, admin, etc.)

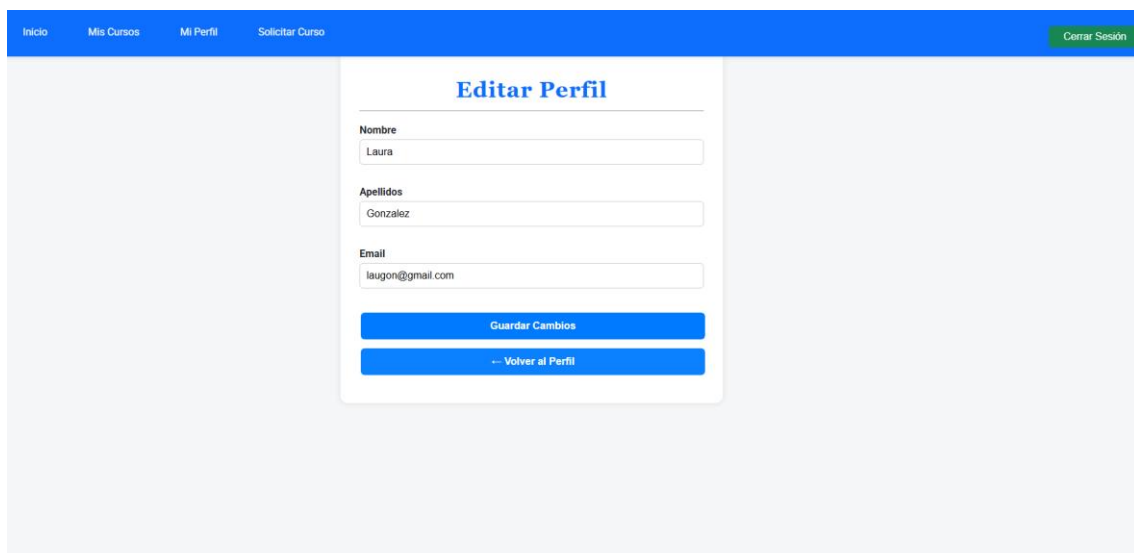
Para ello, realizamos una petición GET al endpoint /usuarios/me, enviando el token JWT almacenado en localStorage. La respuesta se muestra dentro de una tarjeta centrada, manteniendo la coherencia visual con el resto del diseño de la plataforma. Esta pantalla es accesible únicamente para usuarios autenticados.

Debajo de los datos aparece un botón “Editar Perfil” que permite redirigir a la siguiente plantilla.



Cuando el usuario accede a la vista editar-perfil.html, se cargan automáticamente sus datos actuales en un formulario editable mediante una petición al mismo endpoint de perfil. Aquí podrá modificar los campos nombre, apellidos y email. Una vez editados, se realiza una petición PUT al backend en /auth/usuarios/editar-perfil, enviando los datos en formato JSON.

Tras la actualización, se muestra un mensaje de confirmación y el usuario es redirigido nuevamente a su perfil tras unos segundos. Si se produce algún error, se informa al usuario de forma clara sin necesidad de recargar la página.



## - Solicitar-cursos.html

Una de las funcionalidades exclusivas del rol **estudiante** dentro de nuestra aplicación es la posibilidad de sugerir o solicitar nuevos cursos a través de un formulario accesible desde el menú principal bajo el enlace "Solicitar Curso". Este apartado está pensado como un canal de comunicación entre los estudiantes y el equipo académico o de administración.

La plantilla solicitar-curso.html permite a los usuarios enviar propuestas de cursos que consideren necesarios o interesantes. Desde el frontend, se habilita un formulario donde se introducen los siguientes datos:

- **Nombre del curso:** campo obligatorio de texto.
- **Descripción:** área de texto para detallar el contenido, enfoque o necesidad detectada.
- **Duración estimada en horas:** número entero.

Toda esta información se recopila y, al hacer clic en el botón "Enviar Solicitud", se envía mediante una petición POST al endpoint /solicitudes/. Esta acción se realiza de forma asíncrona a través de fetch, incluyendo el token JWT en la cabecera de autorización para validar la identidad del usuario.

Antes de realizar la solicitud, el sistema valida que el usuario haya iniciado sesión. En caso contrario, se muestra un mensaje de advertencia. Además, si hay errores en la conexión o en los datos enviados, el usuario recibe retroalimentación inmediata en pantalla, manteniendo la usabilidad del sistema.

Este mecanismo permite que el backend almacene las solicitudes para que los administradores puedan revisarlas más tarde desde su panel correspondiente (bandeja de solicitudes), fomentando así un entorno dinámico donde se tienen en cuenta las necesidades reales de los usuarios.

The screenshot shows a web application interface for 'UAX Universidad Alfonso X el Sabio'. The top navigation bar is blue with links: 'Inicio', 'Mis Cursos', 'Mi Perfil', and 'Solicitar Curso'. A 'Cerrar Sesión' button is on the right. The main content area is light blue and features a white card titled 'Solicitar un Nuevo Curso'. Inside the card, there are three input fields: 'Nombre del curso', 'Descripción', and 'Duración (horas)'. Below these fields is a blue button labeled 'Enviar Solicitud'.

## - **Bandeja-solicitudes.html**

Una vez que un estudiante envía una solicitud desde la sección "*Solicitar Curso*", esta información no queda almacenada de manera pasiva, sino que se envía directamente a un sistema de revisión gestionado por los roles **admin** o **superadmin**. Este sistema es la bandeja de solicitudes, implementada a través de la plantilla `bandeja-solicitudes.html`.

Esta sección está protegida por permisos y solo es accesible para usuarios con rol administrativo. En cuanto se accede, la vista realiza una solicitud GET al backend mediante la ruta `/solicitudes`, utilizando el token de autenticación almacenado en `localStorage`. Si el usuario no tiene privilegios suficientes o el token no es válido, no podrá acceder a los datos.

El contenido se carga de forma dinámica y se renderiza en una tabla. Esta tabla muestra cada solicitud con los siguientes datos:

- **Nombre del curso propuesto**
- **Descripción del contenido**
- **Duración estimada**
- **Acciones disponibles** (crear o eliminar)

### **Acciones disponibles**

- **Crear Curso:** Redirige al formulario de creación (`crear-curso.html`) donde un administrador puede generar oficialmente ese curso y ponerlo a disposición de los estudiantes. Si bien el botón realiza una redirección básica, en un futuro se podría vincular automáticamente con los datos de la solicitud.
- **Eliminar Solicitud:** Envía una petición DELETE al backend. Tras confirmar la acción, se elimina de la bandeja y se actualiza la vista para mantener los datos sincronizados sin necesidad de recargar la página completa.

Esta bandeja funciona como una herramienta de moderación y control. La existencia de este mecanismo refleja un flujo de trabajo más completo: la propuesta por parte de estudiantes no genera cambios directos en la base de datos de cursos activos, sino que pasa primero por una validación administrativa, fomentando así una arquitectura robusta y segura.

Solicitudes de Nuevos Cursos			
Nombre	Descripción	Duración	Acciones
Curso C ++	Holame gustaria hacer un curso acerca de C++ para obtener los conocimientos necesarios y poder entender el lenguaje	35 horas	<a href="#">Crear Curso</a> <a href="#">Eliminar</a>
Curso de Diseño App	Queria un curso para poder saber los estandares básicos de los diseños de una web	15 horas	<a href="#">Crear Curso</a> <a href="#">Eliminar</a>

## - Crear-curso.html

La vista de creación de cursos forma parte esencial del panel de gestión al que solo tienen acceso los roles **admin** y **superadmin**. Esta sección está diseñada para permitir la alta manual de nuevos cursos, bien de forma directa o como respuesta a solicitudes enviadas previamente por los estudiantes.

### Comprobación de permisos

Nada más cargar la página, se lanza una verificación del rol mediante la función `obtenerPerfil()`. Si el usuario que accede no tiene permisos de tipo *admin* o *super*, se deniega el acceso mostrando un mensaje de error. Esta es una medida de seguridad indispensable para garantizar que solo personal autorizado pueda gestionar la oferta educativa.

El formulario de creación contiene los siguientes campos:

- **Nombre del curso**
- **Descripción**
- **Duración** en horas
- **Disponibilidad** (checkbox)
- **Número de plazas disponibles**
- **Profesor responsable** (opcional, seleccionado de una lista dinámica)

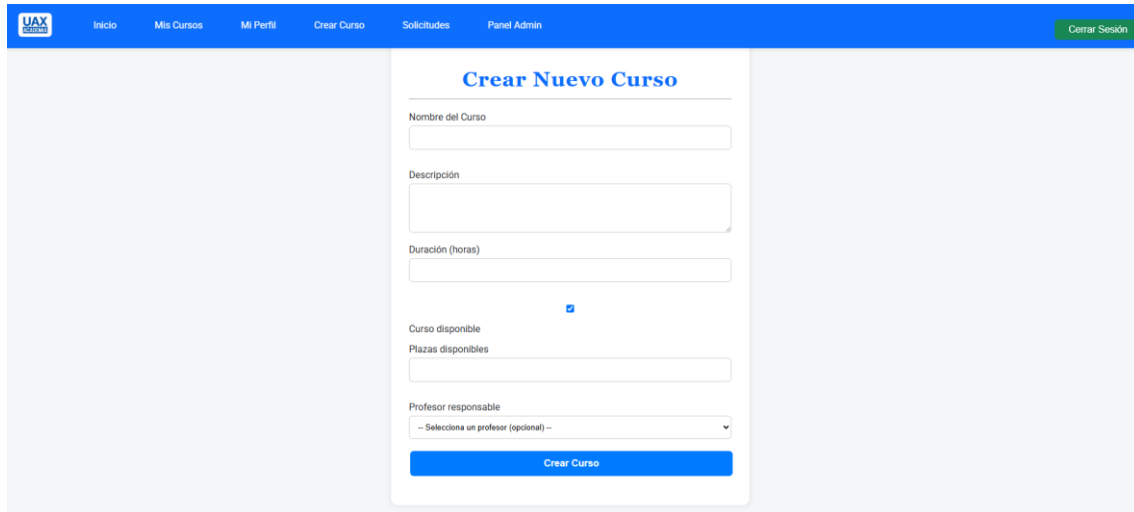
Cada uno de estos campos es validado antes de su envío para asegurar integridad en los datos almacenados. En caso de error o valores no válidos, se proporciona feedback inmediato al usuario.

### Selección de profesor

El campo de selección de profesor se rellena dinámicamente al cargar la página con un fetch a `/auth/lista-profesores`. Esto permite ofrecer en tiempo real todos los docentes disponibles para ser asignados al nuevo curso, facilitando así la administración de manera eficiente.



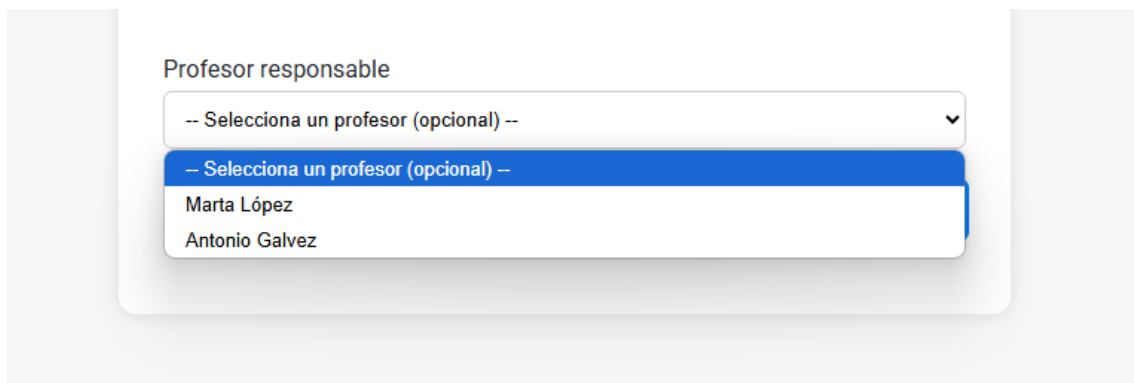
Cuando el usuario confirma la creación, se lanza una petición POST al backend (/cursos/cursos). Los datos se envían en formato JSON, incluyendo opcionalmente el ID del profesor. Si la operación es exitosa, se muestra un mensaje de confirmación y el formulario se resetea para facilitar la creación de más cursos sin recargar la página.



The screenshot shows the 'Crear Nuevo Curso' form in the UAX admin panel. The form is titled 'Crear Nuevo Curso' and contains the following fields:

- Nombre del Curso
- Descripción
- Duración (horas)
- Curso disponible (checkbox)
- Plazas disponibles
- Profesor responsable (dropdown menu with the text '-- Selecciona un profesor (opcional) --')

A blue button labeled 'Crear Curso' is located at the bottom of the form.



The screenshot shows the 'Profesor responsable' dropdown menu. The menu is open, displaying the following options:

- Selecciona un profesor (opcional) --
- Marta López
- Antonio Galvez

#### - Panel-admin.html

Dentro del desarrollo de la plataforma, una de las funcionalidades más críticas ha sido la gestión de usuarios. Para ello implementamos un **panel de administración** exclusivo para los usuarios con rol de **superadmin** (rol más elevado). Este panel permite realizar tareas de mantenimiento y control sobre todos los usuarios registrados en la plataforma.

Al cargar la vista, lo primero que se ejecuta es una comprobación de privilegios. Si el usuario no posee el rol "super", se muestra un mensaje de acceso denegado, impidiendo por completo el acceso a la tabla. Este control de acceso es indispensable para preservar la integridad del sistema y evitar manipulaciones no autorizadas.

El panel muestra una tabla dinámica donde se cargan todos los usuarios registrados a través de una petición GET a /auth/usuarios. Los campos visualizados son:

- ID
- Nombre y apellidos
- Email
- Rol actual
- Estado (habilitado o deshabilitado)
- Acciones disponibles
- Opciones de cambio de rol

La tabla cuenta además con un buscador en tiempo real que permite filtrar por nombre, correo, ID o tipo de rol, facilitando la administración especialmente en sistemas con grandes volúmenes de usuarios.

Cada fila de usuario incluye una serie de botones que permiten:

- **Hacer admin / Quitar admin:** asigna o retira permisos administrativos.
- **Habilitar / Deshabilitar:** controla el acceso al sistema de forma inmediata.
- **Cambiar rol:** permite redefinir el tipo de usuario (estudiante, profesor u otro) mediante un selector desplegable.

Estas acciones se ejecutan enviando peticiones PUT a las rutas correspondientes del backend, con validación previa del token de sesión.

Después de cada acción (cambio de rol, deshabilitar, etc.), se vuelve a lanzar la carga de usuarios con `cargarUsuarios()` para actualizar en tiempo real el estado de la tabla. Esto permite mantener al día los cambios aplicados sin necesidad de recargar la página.

### Panel de Administración

Buscar por ID, nombre, email o rol...

ID	Nombre	Apellido	Email	Rol	Estado	Acciones	Cambiar Rol
1	Marta	López	marta@ejemplo.com	profesor	Habilitado	<div>Hacer Admin</div> <div>Deshabilitar</div>	—
2	Marta	García	marta@example.com	user	Habilitado	<div>Hacer Admin</div> <div>Deshabilitar</div>	<div>Estudiante</div> <div>Actualizar</div>
3	Admin	Academia	admin@ejemplo.com	user	Habilitado	<div>Hacer Admin</div> <div>Deshabilitar</div>	<div>Estudiante</div> <div>Actualizar</div>
4	Marta	Merino	mpamimer@gmail.com	admin	Habilitado	<div>Quitar Admin</div> <div>Deshabilitar</div>	—
5	Pepe	Sanches	peper@gmail.com	estudiante	Habilitado	<div>Hacer Admin</div> <div>Deshabilitar</div>	—
6	Luis	Fernandez	luisf@gmail.com	estudiante	Habilitado	<div>Hacer Admin</div> <div>Deshabilitar</div>	—

### Panel de Administración

Buscar por ID, nombre, email o rol...

ID	Nombre	Apellido	Email	Rol	Estado	Acciones	Cambiar Rol
1	Marta	López	marta@ejemplo.com	admin	Habilitado	<div>Quitar Admin</div> <div>Deshabilitar</div>	—
2	Marta	García	marta@example.com	user	Deshabilitado	<div>Hacer Admin</div> <div>Habilitar</div>	<div>Estudiante</div> <div>Actualizar</div>
3	Admin	Academia	admin@ejemplo.com	user	Habilitado	<div>Hacer Admin</div> <div>Deshabilitar</div>	<div>Estudiante</div> <div>Actualizar</div>
4	Marta	Merino	mpamimer@gmail.com	admin	Habilitado	<div>Quitar Admin</div> <div>Deshabilitar</div>	—
5	Pepe	Sanches	peper@gmail.com	estudiante	Habilitado	<div>Hacer Admin</div> <div>Deshabilitar</div>	—
6	Luis	Fernandez	luisf@gmail.com	estudiante	Habilitado	<div>Hacer Admin</div> <div>Deshabilitar</div>	—

Estudiante

Estudiante

Profesor

Otro

y podemos ver como en nuestra base de datos se cambia al momento:

id	nombre	apellidos	email	password	tipo	fecha_creacion	fecha_modificacion	habilitado	id_rol
1	Marta	López	marta@ejemplo.com	\$2b\$12\$e2lin.ILSYDfBqfVNeYTOIRAvc.ODuA...	admin	2025-05-02 11:11:10	2025-05-07 21:35:56	1	2
2	Marta	García	marta@example.com	\$2b\$12\$96nAFF8jeF9dE/1mee7qFOJkkLolvoT...	user	2025-05-02 11:31:51	2025-05-07 21:31:47	0	3
3	Admin	Academia	admin@ejemplo.com	\$2b\$12\$F52mVr6w1CfSvWbK37KynpOt68SPd/kh...	user	2025-05-02 11:49:28	2025-05-07 17:13:11	1	3
4	Marta	Merino	mpamimer@gmail.com	\$2b\$12\$AjTKMcGmmU109/aIS9dTte2janJygzA...	admin	2025-05-04 13:36:56	2025-05-07 17:14:58	1	2
5	Pepe	Sanchez	peper@gmail.com	\$2b\$12\$DgIp1Aftz9.Ep6LBjTubVeEbac4LYww27...	estudiante	2025-05-04 14:08:40	2025-05-07 20:28:46	1	3
6	Luis	Fernandez	luisf@gmail.com	\$2b\$12\$3ykadjTtH4as8x1Bg3bbemf1J08JHXz...	estudiante	2025-05-04 14:43:48	2025-05-04 14:43:48	1	3
7	Claudia	Corona	clauco@gmail.com	\$2b\$12\$IZLLeqfUEDEHzEYfYxZOI7Pr8XhMOJL...	estudiante	2025-05-04 15:28:35	2025-05-04 15:28:35	1	3
8	Marta	Pamies	marta.pamiesm@gmail.com	\$2b\$12\$yoOEgHeXQr3W4ff5qssousHATKvWlp...	user	2025-05-05 13:58:18	2025-05-05 13:58:18	1	3
9	marta	pamies	marta@gmail.com	\$2b\$12\$y0Qa10qR2yvrflCuDWglatUXSRsOt4x...	user	2025-05-05 14:01:54	2025-05-06 08:20:13	1	3
10	Pedro	Pinto	pedropi@gmail.com	\$2b\$12\$0mMSKGItdckQHn4Wdmhb.C2y0ZAMV...	super	2025-05-05 14:57:49	2025-05-05 14:57:49	1	1
11	Marcos	Perez	marquitos@gmail.com	\$2b\$12\$e4ggOpbVdkavms1Kn3/wguS28IEeLP2...	admin	2025-05-05 22:42:48	2025-05-06 08:20:28	1	2
12	Francisco	Palomar	franpa@gmail.com	\$2b\$12\$eedK.Ogf1mr4yLoc8AndheTyyJ5BYsJB...	estudiante	2025-05-06 13:56:13	2025-05-06 13:56:13	1	3
13	Alonso	Hilario	alonhi@gmail.com	\$2b\$12\$EBVJ1Fn1W4au0Q.ocfx/8.IB1gdH4K1...	estudiante	2025-05-06 15:42:03	2025-05-06 15:42:03	1	3
14	Antonio	Galvez	antogal@gmail.com	\$2b\$12\$yia7nY2gQP3hSbQB4LzONodJDbw2BD...	profesor	2025-05-06 16:33:30	2025-05-06 16:33:30	1	3

Y en modo responsive se vería tal que así:

UAX

Cerrar sesión

ID

1

Nombre

Marta

Apellido

López

Email

marta@ejemplo.com

Rol

admin

Estado

Habilitado

Acciones

Quitar Admin

Deshabilitar

Cambiar Rol

—

ID

2

Nombre

Marta

Apellido

García

Email

marta@example.com

Rol

user

Estado

Deshabilitado

Acciones

Hacer Admin

↑ Inicio

UAX

Cerrar sesión

Panel de Administración

1

ID

1

Nombre

Marta

Apellido

López

Email

marta@ejemplo.com

Rol

admin

Estado

Habilitado

Acciones

Quitar Admin

Deshabilitar

Cambiar Rol

—

ID

10

Nombre

Pedro

Apellido

Pinto

Email

pedroni@gmail.com

↑ Inicio

## - Navbar.js

Uno de los elementos centrales para mejorar la experiencia de navegación en nuestra aplicación fue la implementación de una barra de navegación dinámica. Para ello, desarrollamos un script JavaScript que se encarga de construir e insertar automáticamente el contenido del `<nav>` en todas las páginas de la plataforma. Gracias a esto, evitamos repetir manualmente el mismo HTML en cada archivo, facilitando tanto la escalabilidad como el mantenimiento del código.

### Construcción dinámica del menú

La función principal, `construirNavbar(rol)`, recibe como parámetro el tipo de usuario autenticado (`rol`) y en función de este genera diferentes enlaces visibles en la interfaz. El contenido de la barra se construye mediante `innerHTML`, con las siguientes condiciones:

- Si el usuario **no ha iniciado sesión**, se muestran los enlaces de "Inicio", "Iniciar sesión" y "Registrarse".
- Si el usuario **está autenticado**:
  - Se añaden enlaces comunes como "Inicio", "Mis Cursos" y "Mi Perfil".
  - Si su rol es "user", se incluye la opción "Solicitar Curso".
  - Si el rol es "admin" o "super", se incluyen "Crear Curso" y "Solicitudes".
  - Si es "super", también aparece el enlace al "Panel Admin".

Esto permite que cada tipo de usuario vea solo las opciones relevantes, mejorando la claridad y seguridad de la interfaz.

La barra se inserta dinámicamente en el DOM al comienzo del `<body>` utilizando `document.body.prepend(nav)`. De esta forma, se garantiza su presencia en todas las páginas sin necesidad de copiar el mismo bloque HTML repetidamente.

Además, si el usuario está autenticado, se activa también el botón de "Cerrar sesión", que limpia el token del almacenamiento local (`localStorage`) y redirige al usuario al inicio.

La función `cargarNavbar()` se ejecuta automáticamente al cargar cualquier página. Comprueba si existe un token almacenado, y si es así, realiza una petición al endpoint `/usuarios/me` del backend para identificar al usuario. En base a la respuesta, se obtiene su rol, que se utiliza para construir el navbar personalizado llamando a `construirNavbar(rol)`.

En caso de error o ausencia de sesión, se carga el navbar para invitados.

- **App.py:**

El archivo app.py es el punto de entrada de nuestra interfaz web construida con Flask. Su función principal es definir todas las rutas que componen la parte visual de la plataforma. Cada ruta representa una página del sitio, como el inicio, el login, el panel de administración o la ficha de perfil.

Las rutas cubren tanto funcionalidades generales (inicio, login, registro, dashboard), como funcionalidades específicas para usuarios autenticados (mis cursos, perfil, editar perfil), y vistas restringidas a administradores o profesores (panel de administración, creación de cursos, gestión de solicitudes).

El servidor Flask se ejecuta en el puerto 5000, y se conecta con el backend en el puerto 8000 para obtener datos en tiempo real. Esta separación entre frontend y backend nos permitió mantener una arquitectura modular y clara.

## 6. PROBLEMAS ENCONTRADOS

A lo largo del desarrollo del proyecto, surgieron diversas dificultades técnicas y organizativas que condicionaron tanto la estructura del sistema como los tiempos de entrega. A continuación, se enumeran los más relevantes:

### 1. Elección equivocada del enfoque inicial: React en lugar de Flask

El error más crítico fue comenzar desarrollando la interfaz web con **React**, cuando necesitábamos usar librerías flask, esto provoco:

- Desvió el enfoque técnico del proyecto.
- Provocó una desconexión entre frontend y backend.
- Hizo inviable la integración directa con los endpoints REST en local.
- Requirió más tiempo y esfuerzo para deshacer el trabajo previo y rehacer toda la interfaz desde cero en HTML/JS.

Este error fue determinante y retrasó el trabajo, ya que obligó a replantear toda la estructura web.

### 2. Desincronización entre frontend y backend

Hubo momentos en los que los cambios realizados en el backend no se veían reflejados en el frontend, o viceversa. Algunas causas fueron:

- Mal uso de rutas (por ejemplo, errores 404 en endpoints).
- CORS mal configurado.
- Mal formateo de las respuestas JSON o estructuras no coincidentes entre el backend y lo esperado en JS.

### 3. Fallos en la gestión de sesión y visibilidad condicional

Durante gran parte del proyecto, hubo problemas con:

- El botón Cerrar sesión, que no funcionaba por conflictos de ámbito en el script.
- La barra de navegación, que no cambiaba dinámicamente según el estado del usuario (logueado/no logueado).

- La falta de visibilidad dinámica de los botones de “Iniciar sesión” o “Cerrar sesión” dependiendo del token.

#### **4. Ausencia inicial de diseño responsive**

El diseño de la interfaz no era responsive en sus primeras versiones. Fue necesario:

- Reescribir completamente los estilos CSS.
- Implementar un sistema de navbar responsive con menú hamburguesa.
- Asegurar que la experiencia fuera fluida en móviles.

#### **5. Desajuste en la gestión de roles de usuario**

Aunque los roles estaban correctamente definidos en la base de datos (por ejemplo, admin, super, user), estos no se reflejaban adecuadamente en la interfaz web.

Existía una desconexión entre la forma en que el backend proporcionaba la información del rol y cómo el frontend la interpretaba o visualizaba.

#### **6. Administración de usuarios no actualizaba dinámicamente**

Al cambiar el estado de un usuario (habilitar, deshabilitar, hacer admin), los cambios no se reflejaban visualmente en la tabla de administración.

El backend actualizaba correctamente los datos, pero el frontend no refrescaba automáticamente la información mostrada.



## 7. CONCLUSIONES

Este proyecto nos ha permitido desarrollar de forma completa una aplicación web modular, conectando un frontend funcional y dinámico con un backend robusto y seguro. A través de la simulación de una academia online de programación, hemos logrado implementar una arquitectura realista compuesta por una API REST con FastAPI y una interfaz de usuario con Flask, comunicadas mediante peticiones HTTP y protegidas con autenticación basada en tokens JWT.

A pesar de los desafíos técnicos que surgieron hemos sabido reconducir el proyecto, aplicar soluciones viables y entregar un sistema funcional, completo y modular.

Este trabajo nos ha servido no solo para consolidar conocimientos de desarrollo web, sino también para adquirir una visión integral de cómo se estructura, conecta y despliega una aplicación profesional. El resultado es una plataforma realista y escalable, en la que cada decisión técnica ha sido justificada por la experiencia práctica, el trabajo en equipo y la búsqueda de soluciones efectivas a problemas reales.