

**DESARROLLO FULLSTACK I**

DSY1103 - 011D

Duoc UC: sede Maipu

**“EduTech Innovators SPA”**

Jonathan Larraguibel

Daniel Morales

Danae Miranda

EDUARDO ANTONIO BAEZA

26/05/2025

**ÍNDICE DE CONTENIDO**

[**Introducción 3**](#_7j8xp3euar8b)

[**1. Diagrama de arquitectura de microservicios. 3**](#_ctp81ey68jfe)

[**2. Estructura del proyecto. 4**](#_g0bypk6twcry)

[C. Capa de Servicios (Lógica de Negocio) 6](#_p64up0ii9ls)

[D. Capa de Repositorios (Acceso a Datos) 6](#_kcfqvbwk8jd3)

[E. Clase Principal de Arranque 6](#_uudyl9xq1ctq)

[**3. Base de datos. 8**](#_4nz9lkqojdbd)

[**4. Implementación de los Servicios. 10**](#_5spk65k0y8t3)

[**5. Implementación de Vistas. 19**](#_f3cy3xzfspxx)

[**6. Uso de Git – GitHub. 23**](#_wl2w4zpxh7e0)

[**Conclusión 26**](#_bis7mk55t1o3)

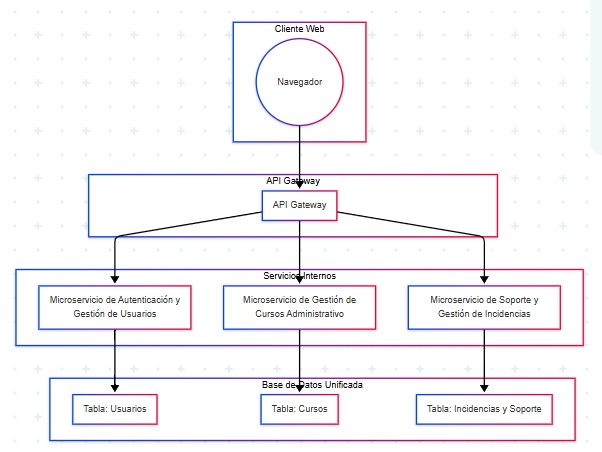
# Introducción

Este informe expone el diseño e implementación de una solución basada en microservicios para la gestión integral de usuarios, cursos y soporte en un entorno exclusivamente web. La arquitectura propuesta se compone de tres servicios independientes, cada uno responsable de un dominio específico, que se comunican a través de un API Gateway y comparten una única base de datos, en la que cada servicio opera sobre tablas especializadas. La aplicación se organiza en capas de presentación, servicios y acceso a datos, lo que garantiza una estructura modular, escalable y de fácil mantenimiento.

El documento detalla la implementación de cada microservicio, abarcando desde la definición de entidades y la exposición de endpoints REST hasta la validación de las operaciones mediante pruebas en Postman.

Con este enfoque, se busca demostrar cómo una arquitectura modular y orientada a servicios permite desarrollar aplicaciones robustas y adaptables, optimizando tanto el desarrollo como el mantenimiento continuo del sistema.

# 1. Diagrama de arquitectura de microservicios.



# 2. Estructura del proyecto.

A continuación se presentará una descripción detallada de la estructura del proyecto, abarcando las dependencias configuradas, los componentes implementados y una explicación de cada parte importante. Cabe mencionar que el proyecto se ha dividido en tres microservicios independientes (Usuarios, Cursos y Soporte), y fue desarrollado como un proyecto Spring Boot, que comparte una única base de datos (donde cada servicio accede a su tabla correspondiente).

**2.1. Dependencias y Configuración en el POM**

Cada microservicio utiliza Maven como herramienta de construcción y comparte dependencias esenciales para el desarrollo de aplicaciones web y de persistencia. Entre las principales dependencias se encuentran:

* **spring-boot-starter-data-jpa**: Facilita la integración con JPA para realizar el mapeo objeto-relacional (ORM) y simplifica el acceso a la base de datos mediante interfaces como ***"CrudRepository”***.
* **spring-boot-starter-web**: Proporciona soporte para construir aplicaciones web y servicios REST. Permite definir controladores anotados con ***“@RestController”*** y manejar solicitudes HTTP (GET, POST, PUT, DELETE).
* **spring-boot-starter-thymeleaf**: Permite la integración con el motor de plantillas Thymeleaf. Es utilizado en los controladores de vista para renderizar páginas HTML y gestionar formularios web.
* **spring-boot-devtools**: Aporta funcionalidades de recarga automática (hot reload) y mejora el proceso de desarrollo, permitiendo ver los cambios sin tener que reiniciar la aplicación manualmente.
* **mysql-connector-java**: Es el driver JDBC que posibilita la conexión a una base de datos MySQL en tiempo de ejecución.
* **spring-boot-starter-test**: Incluye herramientas y librerías (como JUnit) para la realización de pruebas unitarias e integradas.

El archivo POM también declara a “**spring-boot-starter-parent**” como padre para aprovechar la configuración predefinida y gestionar la versión de Java (en este caso, Java 23).

**2.2 Componentes Implementados y su Función**

Cada microservicio se estructura en tres capas: presentación, servicios y acceso a datos. Esta organización promueve una arquitectura clara, modular y fácil de mantener. A continuación se describen brevemente cada una de ellas:

**A. Capa de Entidades (Modelo de Datos)**

Estas clases representan los objetos del dominio y se mapean a tablas en la base de datos mediante anotaciones JPA.

* En el microservicio de Cursos:

***‘Curso’*** Contiene campos como ***`id`***, ***`nombreCurso`***, ***`descripcion`***, ***`instructor`***, ***`cursoPublicado`*** y ***`reporteInscripcion`***. Explicación: Define la estructura de la información relacionada con cursos y se usa para persistir y recuperar datos de la tabla ***`curso`***.

* En el microservicio de Soporte:

***`Ticket`*** Contiene campos como ***`id`***, ***`titulo`***, ***`descripcion`***, ***`estado`***, ***`prioridad`*** y ***`tipo`***. Explicación: Representa los tickets de soporte o incidencias, permitiendo hacer búsquedas (por ejemplo, por título) y su gestión completa.

* En el microservicio de Usuarios:

***`Usuario`*** Contiene campos como ***`id`***, ***`nombre`***, ***`apellido`***, ***`correo`***, ***`contrasenna`***, ***`rol`*** y ***`activado`***. Explicación: Permite la gestión de los datos del usuario, abarcando tanto información personal como datos para autenticación y autorización.

**B. Capa de Controladores**

Se ha implementado dos tipos de controladores en cada microservicio:

1. **Controladores REST:** Manejan las peticiones HTTP y devuelven respuestas en formato JSON.

* Ejemplo en el microservicio de Cursos: ***`CursoController`*** Implementa métodos con ***`@GetMapping`***, ***`@PostMapping`***, ***`@PutMapping`*** y ***`@DeleteMapping`*** para exponer la API REST que gestiona las operaciones CRUD sobre cursos.
* Ejemplo en el microservicio de Soporte: ***`TicketController`*** Posee métodos para listar tickets, ver detalles, crear, actualizar y eliminar tickets (usando respuestas HTTP con códigos de estado adecuados).
* Ejemplo en el microservicio de Usuarios: ***`UsuarioRestController`*** Expone endpoints REST para la gestión de usuarios, permitiendo obtener usuarios por id, listar usuarios activos, crear, actualizar y eliminar usuarios.

1. **Controladores de Vista (MVC):** Estos controladores trabajan con Thymeleaf para renderizar páginas HTML y manejar formularios.

* En Cursos: ***`CursoViewController`*** gestiona las vistas para listar cursos, muestra un formulario de creación y edición, y manejar la eliminación mediante redirecciones.
* En Soporte: ***‘SoporteViewController’*** administra la visualización de formularios para crear y editar tickets, así como la búsqueda y listado de tickets.
* En Usuarios: ***‘UsuarioViewController’*** se encarga de listar usuarios, mostrar el formulario para crear o editar información y gestionar la eliminación a través de vistas web.

### **C. Capa de Servicios (Lógica de Negocio)**

La capa de servicios actúa como intermediaria entre los controladores y la capa de acceso a datos. Se define como una interfaz y su implementación correspondiente, encargada de manejar la lógica transaccional y posibles validaciones:

* Para Cursos:  
   **CursoService** y **CursoServiceImpl**  
   Métodos como findByAll(), findById(Long), save(Curso) y delete(Curso) permiten abstraer la lógica de negocio relacionada con cursos, garantizando el uso de transacciones donde es necesario.
* Para Soporte:  
   TicketService y TicketServiceImpl  
   Además de los métodos CRUD habituales, se implementa un método adicional para búsquedas por título (findByTituloContainingIgnoreCase), lo que permite filtrar tickets de soporte de manera flexible.
* Para Usuarios:  
   **UsuarioService** y **UsuarioServiceImpl**  
   Además de los métodos básicos de CRUD, se incluyen métodos para listar solo usuarios activos y para desactivar un usuario, lo cual permite manejar la lógica de activación/desactivación sin eliminar físicamente el registro.

### **D. Capa de Repositorios (Acceso a Datos)**

Los repositorios utilizan Spring Data JPA para acceder a la base de datos sin necesidad de escribir consultas SQL manuales. Se extienden de ***‘CrudRepository’***, lo que proporciona métodos predefinidos para operaciones CRUD.

* En Cursos: ***‘CursoRepository’*** permite operaciones básicas de persistencia sobre la entidad Curso.
* En Soporte: ***‘TicketRepository’*** incluye métodos personalizados, por ejemplo, ***‘findByTituloContainingIgnoreCase’***, para búsquedas específicas en la tabla ticket.
* En Usuarios: ***‘UsuarioRepository’*** no solo proporciona métodos CRUD, sino también métodos personalizados para obtener usuarios por rol o recuperar únicamente aquellos usuarios que estén activados.

### **E. Clase Principal de Arranque**

El proyecto cuenta con **una única clase principal de arranque**, anotada con **@SpringBootApplication,** que actúa como punto de entrada para inicializar el contexto de Spring Boot y ejecutar toda la aplicación.

* **Clase principal:  
   EdutechApplication**  
   Ubicada en el paquete raíz **com.edutech.Aplicaciones**, esta clase inicia todos los componentes definidos en los distintos paquetes del proyecto (controladores, servicios, repositorios, etc.). Su ejecución permite que todos los módulos —usuarios, cursos e incidencias— funcionen como parte de una misma aplicación monolítica estructurada.

Esta configuración facilita el desarrollo modular dentro de un único servicio unificado, manteniendo claridad y separación lógica entre funcionalidades sin necesidad de dividir en microservicios reales.

**2.3. Explicación General y Consideraciones Adicionales**

**Modularidad y Separación de Responsabilidades:**

Cada microservicio está diseñado para cumplir una función específica. Esto permite que se desarrollen, desplieguen y escalan de forma independiente. Además, la división entre controladores REST y controladores de vistas favorece la separación entre la API pública y la interfaz de usuario interna.

**Gestión de Transacciones**:

En la capa de servicios se han aplicado anotaciones ***`@Transactional`*** (con `readOnly = true` en los métodos de consulta) para garantizar la consistencia y el rendimiento al interactuar con la base de datos.

**Acceso a Datos con Spring Data JPA**:

La utilización de interfaces que extienden ***`CrudRepository`*** permite que casi todas las operaciones CRUD se implementen de manera automática. Esto agiliza el desarrollo y facilita el mantenimiento del código.

**UI con Thymeleaf**:

El uso de Thymeleaf en los controladores de vistas permite generar páginas web dinámicas que integran los datos provenientes del backend, haciendo que la interacción del usuario con la aplicación sea intuitiva y responsiva.

**API Gateway (Arquitectura Global)**:

A pesar de que cada microservicio es desarrollado de forma independiente, en un entorno integrado se contempla la utilización de un API Gateway o un proxy inverso para centralizar el enrutamiento de peticiones, aplicar políticas de seguridad y facilitar los procesos de monitoreo y escalabilidad.

Esta estructura y el conjunto de dependencias permiten desarrollar una solución modular, robusta y escalable, donde cada componente juega un rol bien definido en la gestión de cursos, soporte y usuarios.

# 3. Base de datos.

A continuación se detalla la forma en que se ha implementado la solución en cuanto a la base de datos:

**Motor de Base de Datos Utilizado**

MySQL

El proyecto utiliza MySQL como motor de base de datos, aprovechando el conector mysql-connector-j incluido en las dependencias. MySQL es altamente popular en aplicaciones Java/Spring Boot debido a sus siguientes características:

* Escalabilidad y Rendimiento:

MySQL es capaz de manejar grandes volúmenes de datos y múltiples conexiones concurrentes, ideal para aplicaciones web modernas.

* Estructura de la Base de Datos

En la arquitectura planteada se utiliza una única base de datos para centralizar la información de los tres microservicios que conforman la solución. Sin embargo, para mantener la separación lógica entre dominios y facilitar el mantenimiento, se han definido tablas específicas para cada uno de ellos:

1. Tabla: Usuarios

- Propósito: Almacenar datos de autenticación y perfil de los usuarios.

- Campos:

- `id`: Identificador único (clave primaria).

- `nombre`: Nombre del usuario.

- `apellido`: Apellido.

- `correo`: Correo electrónico (único).

- `contrasenna`: Contraseña encriptada.

- `rol`: Rol asignado (por ejemplo, ADMIN, USER).

- `activado`: Indicador booleano para saber si el usuario está activo o no.

2. Tabla: Curso

- Propósito: Gestionar la información de los cursos ofrecidos.

- Campos:

- `id`: Identificador único (clave primaria).

- `nombreCurso`: Nombre del curso.

- `descripcion`: Descripción detallada.

- `instructor`: Nombre del instructor responsable.

- `cursoPublicado`: Indicador booleano que define si el curso está publicado o no.

- `reporteInscripcion`: Información sobre inscripciones o estadísticas asociadas.

3. Tabla: Ticket (o Incidencias y Soporte)

- Propósito: Registrar y hacer seguimiento a los tickets de soporte o incidencias.

- Campos:

- `id`: Identificador único (clave primaria).

- `titulo`: Título breve del ticket.

- `descripcion`: Descripción detallada del problema.

- `estado`: Estado actual del ticket (por ejemplo, Pendiente, En Progreso, Resuelto).

- `prioridad`: Indica la urgencia o el nivel de importancia (Baja, Media, Alta).

- `tipo`: Clasificación del ticket (Soporte, Incidente, Consulta).



# 

# 

# 4. Implementación de los Servicios.

A continuación se ofrece una explicación detallada de la implementación de los servicios definidos en los microservicios (Cursos, Soporte y Usuarios) mediante API Full Rest. Se incluyen ejemplos representativos—simulados en Postman—para cada tipo de petición, evidenciando cómo interactúan con la base de datos.

1. Descripción de los Servicios Implementados

Cada microservicio se ha desarrollado utilizando Spring Boot y se ha estructurado en las siguientes capas:

Controladores REST:

Reciben las peticiones HTTP desde el API Gateway o directamente desde Postman (para pruebas) y transforman las solicitudes a llamadas al servicio. Aquí se definen los endpoints que realizan operaciones CRUD y otras lógicas específicas.

Servicios:

Son responsables de procesar la lógica de negocio, gestionar las transacciones y coordinar el acceso a los datos a través de los repositorios. Se utiliza la anotación `@Transactional` para garantizar la correcta ejecución de las operaciones.

Repositorios (Acceso a Datos):

Con Spring Data JPA se simplifica el acceso a la base de datos. Las interfaces extienden `CrudRepository` o se complementan con métodos personalizados (por ejemplo, búsquedas por nombre o título).

Entidades:

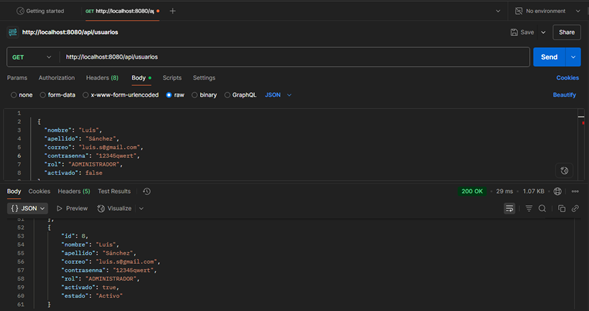
Cada entidad corresponde a una tabla de la base de datos (Usuarios, Curso y Ticket). Se definen con las anotaciones JPA apropiadas (como `@Entity`, `@Table`) para mapearlas a sus respectivas tablas.

Aunque en ambiente productivo se utiliza un API Gateway para el enrutamiento adecuado, para fines de demostración se emplea Postman para interactuar directamente con cada API.

**API-Usuarios**

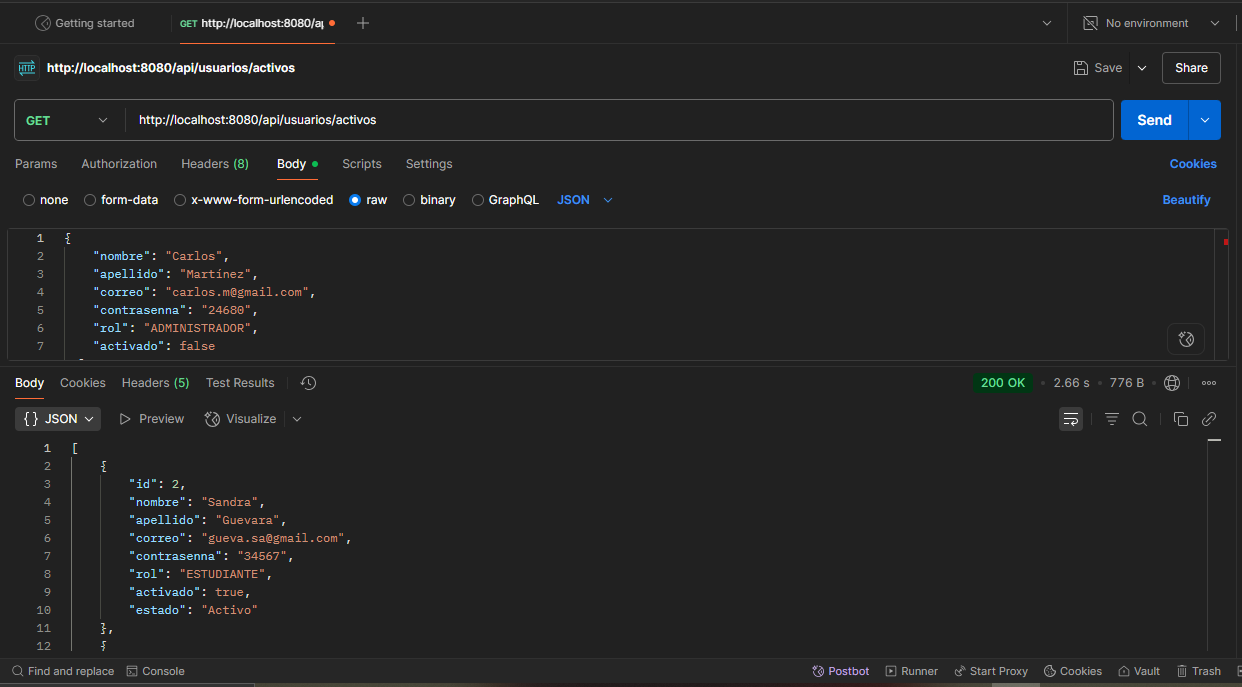
**GET** /api/usuarios**:** Devuelve la lista de todos los usuarios registrados.

*Ejemplo de respuesta (JSON):*

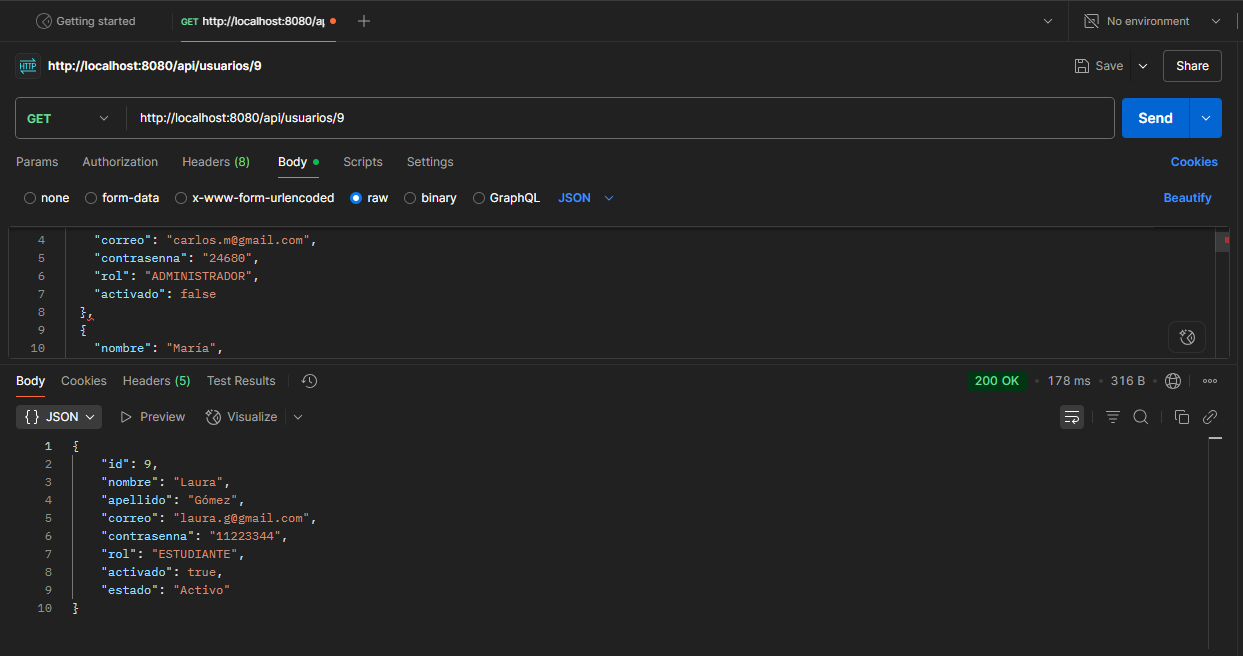
****

**GET** /api/usuarios/activos**:** Filtra y devuelve únicamente los usuarios activados.

*Ejemplo de respuesta (JSON):*

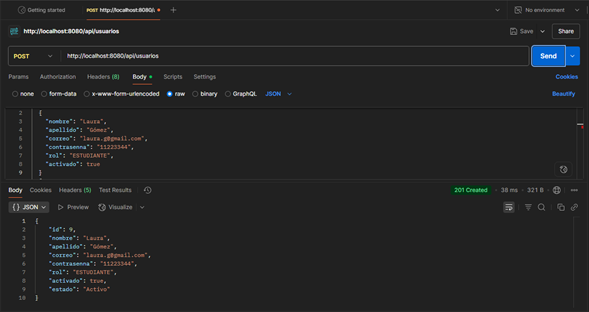
**

**GET** /api/usuarios/{id}**:** Retorna la información del usuario especificado. *Ejemplo de respuesta (JSON):*

**

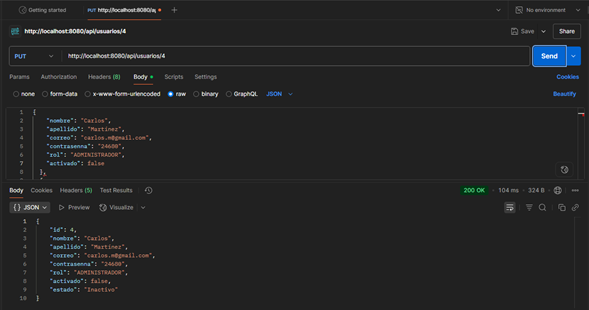
**POST** /api/usuarios**:** Se utiliza para la creación de un nuevo usuario.

*Ejemplo de respuesta (JSON):*



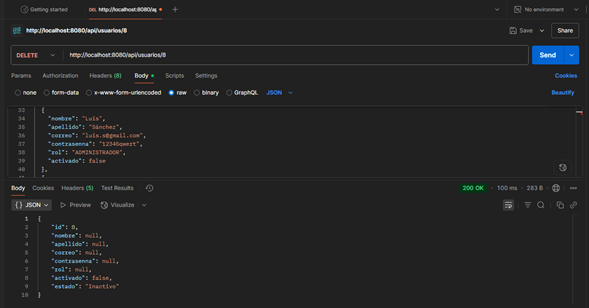
**PUT** /api/usuarios/{id}: Actualiza la información de un usuario existente.

*Ejemplo de respuesta (JSON):*

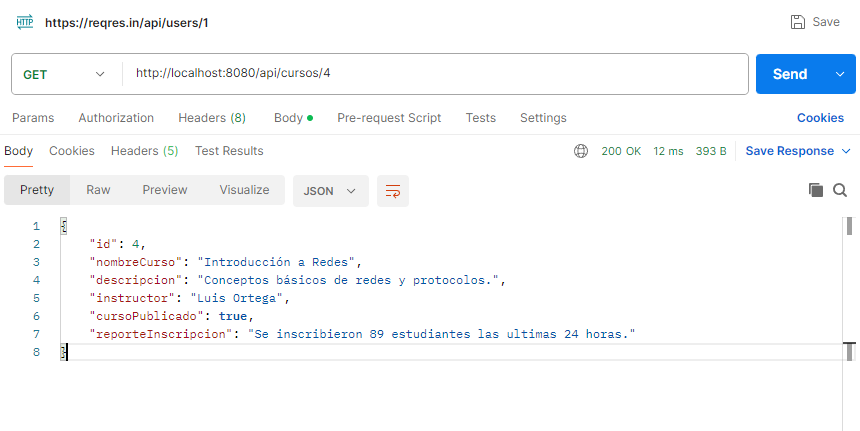
****

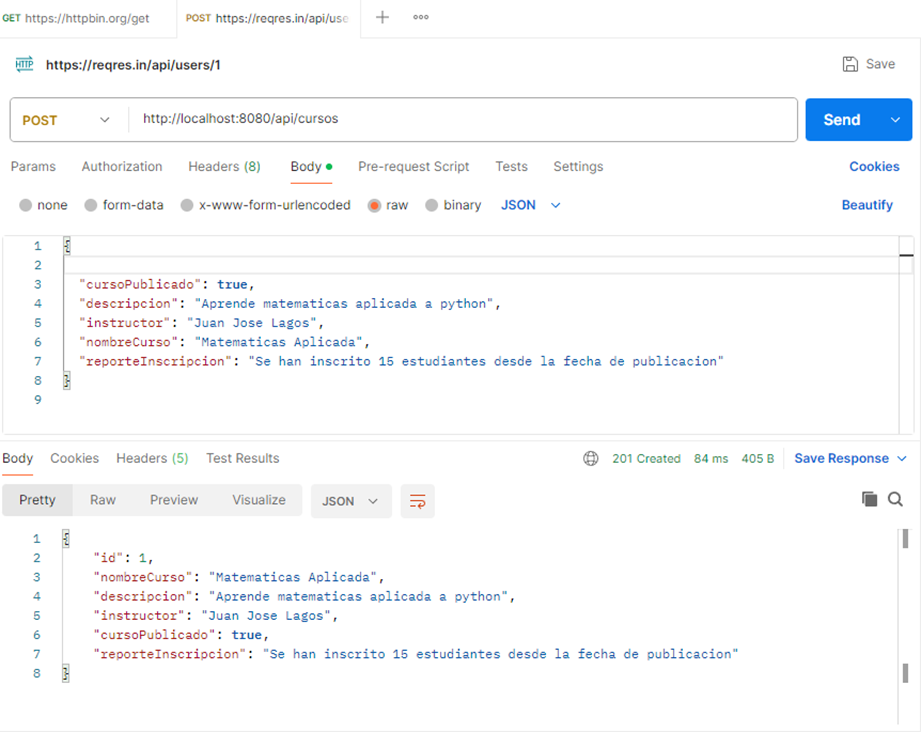
**DELETE** /api/usuarios/{id}: Elimina al usuario identificado.

*Ejemplo de respuesta (JSON):*

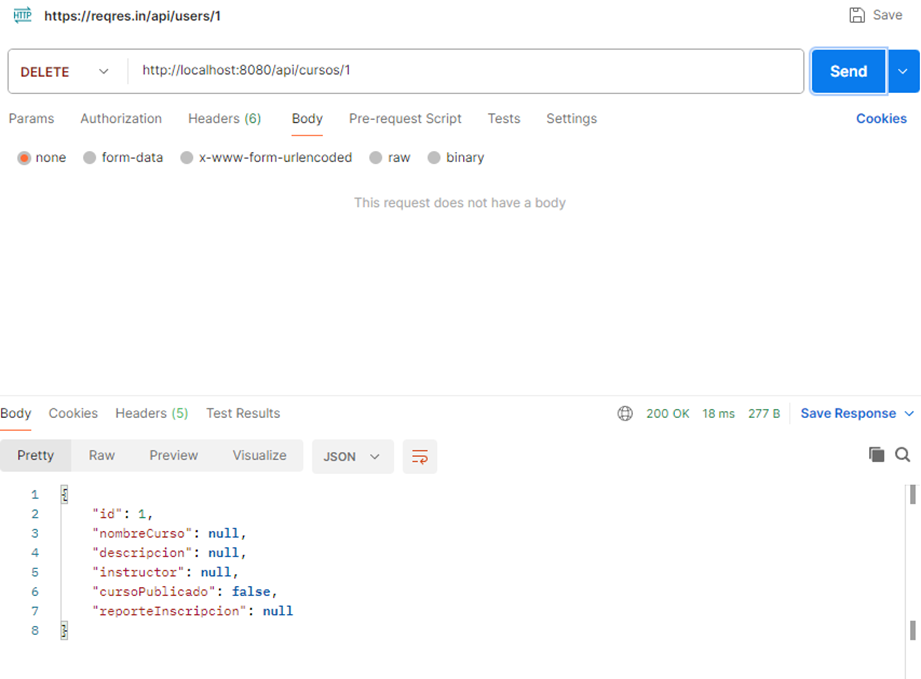
**

**API-Curso**

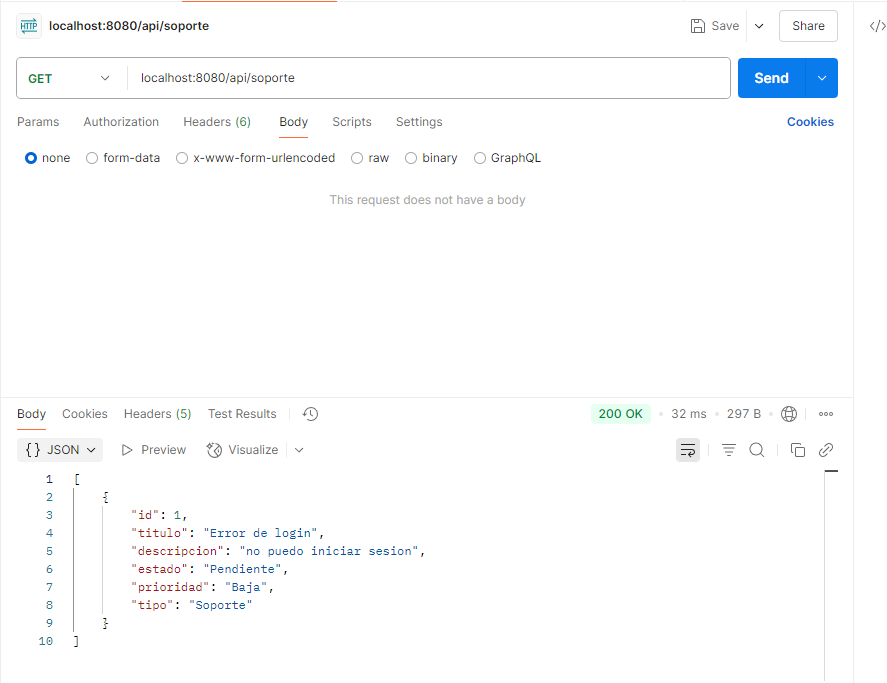
****

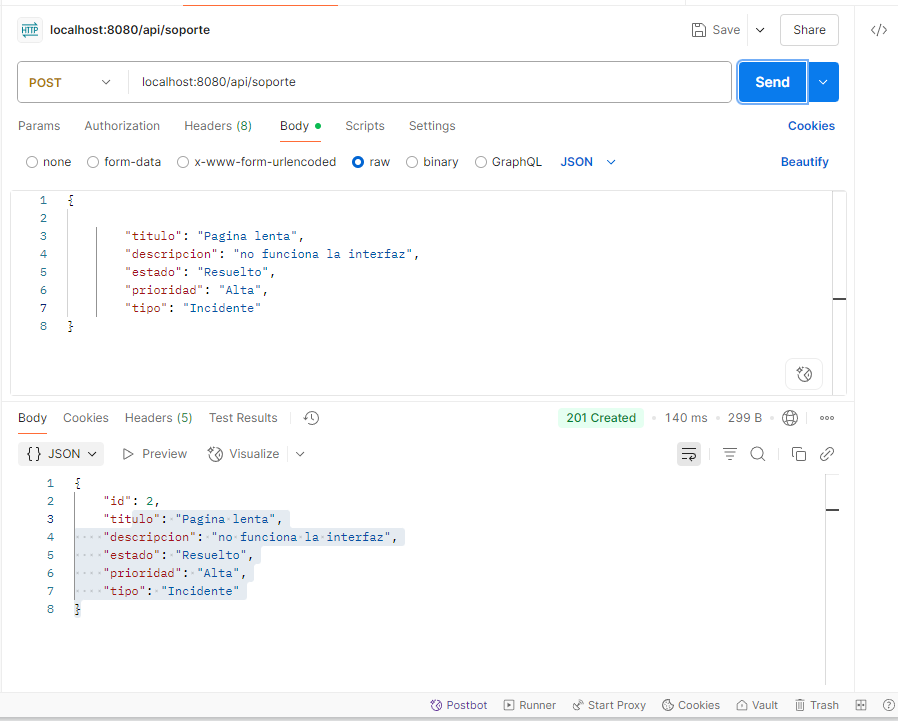
****

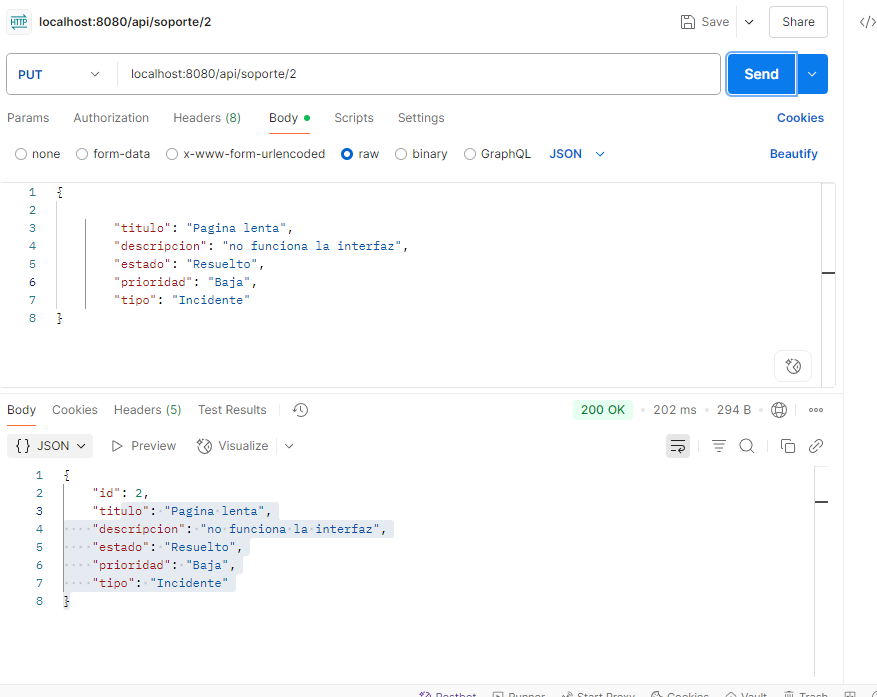
****

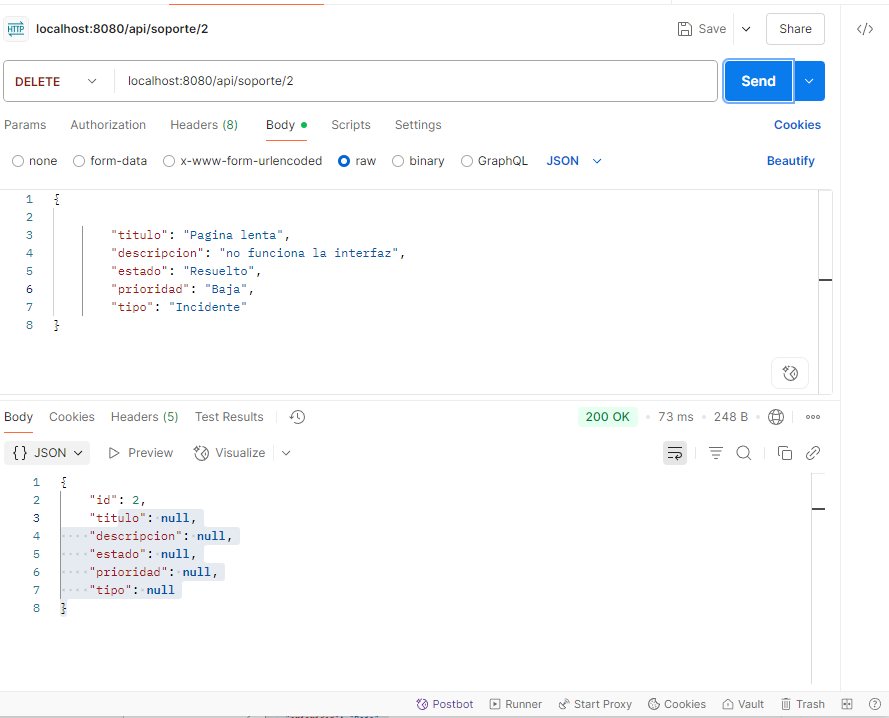
****

**API-Soporte**

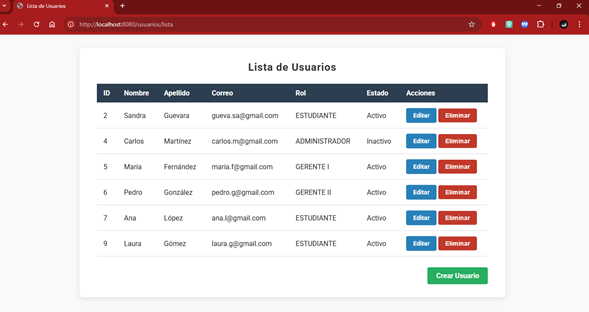




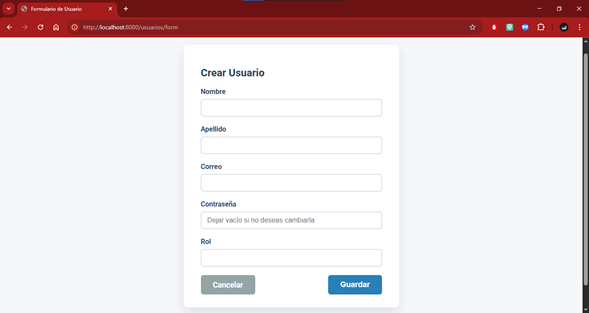


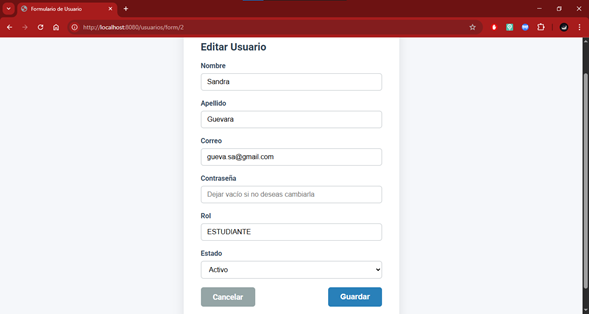


# 5. Implementación de Vistas.

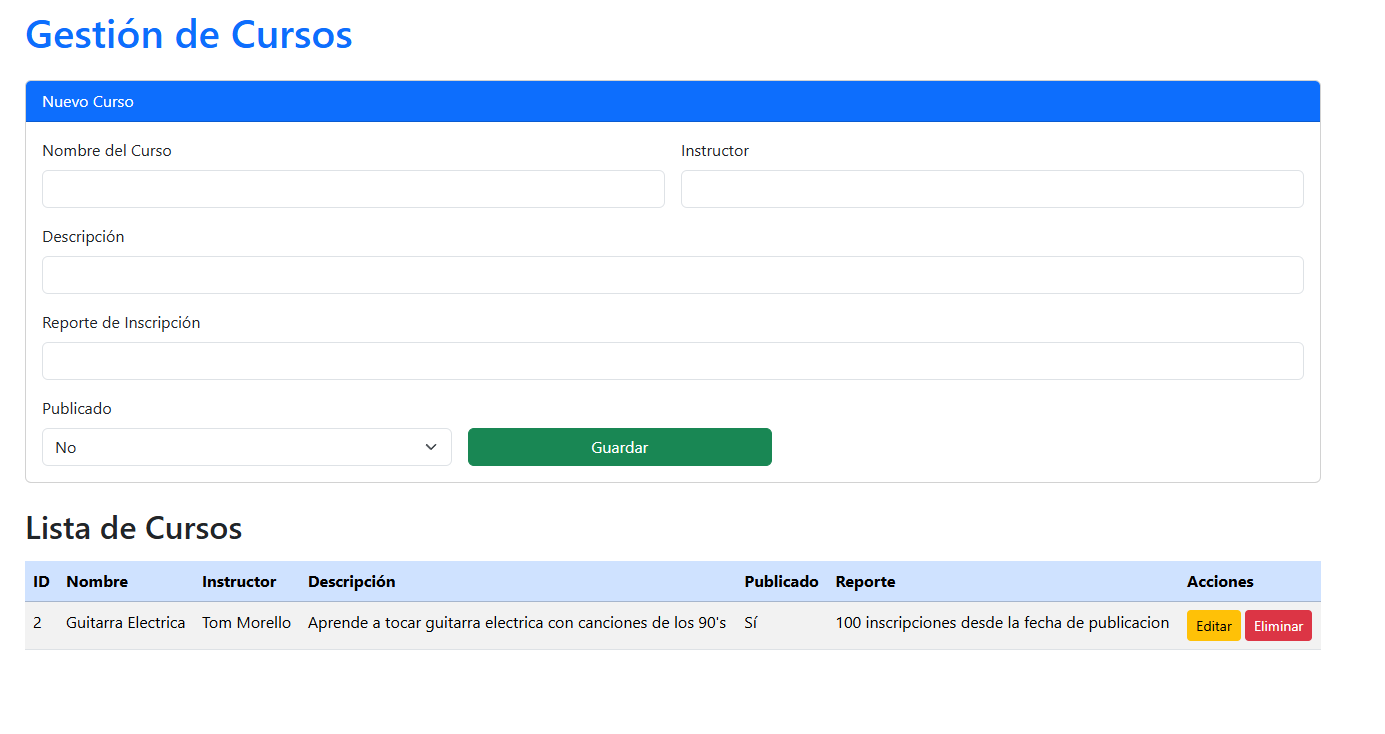
**API-USUARIOS** VISTA DE LISTA:  


**API-USUARIOS** VISTA DE FORM:

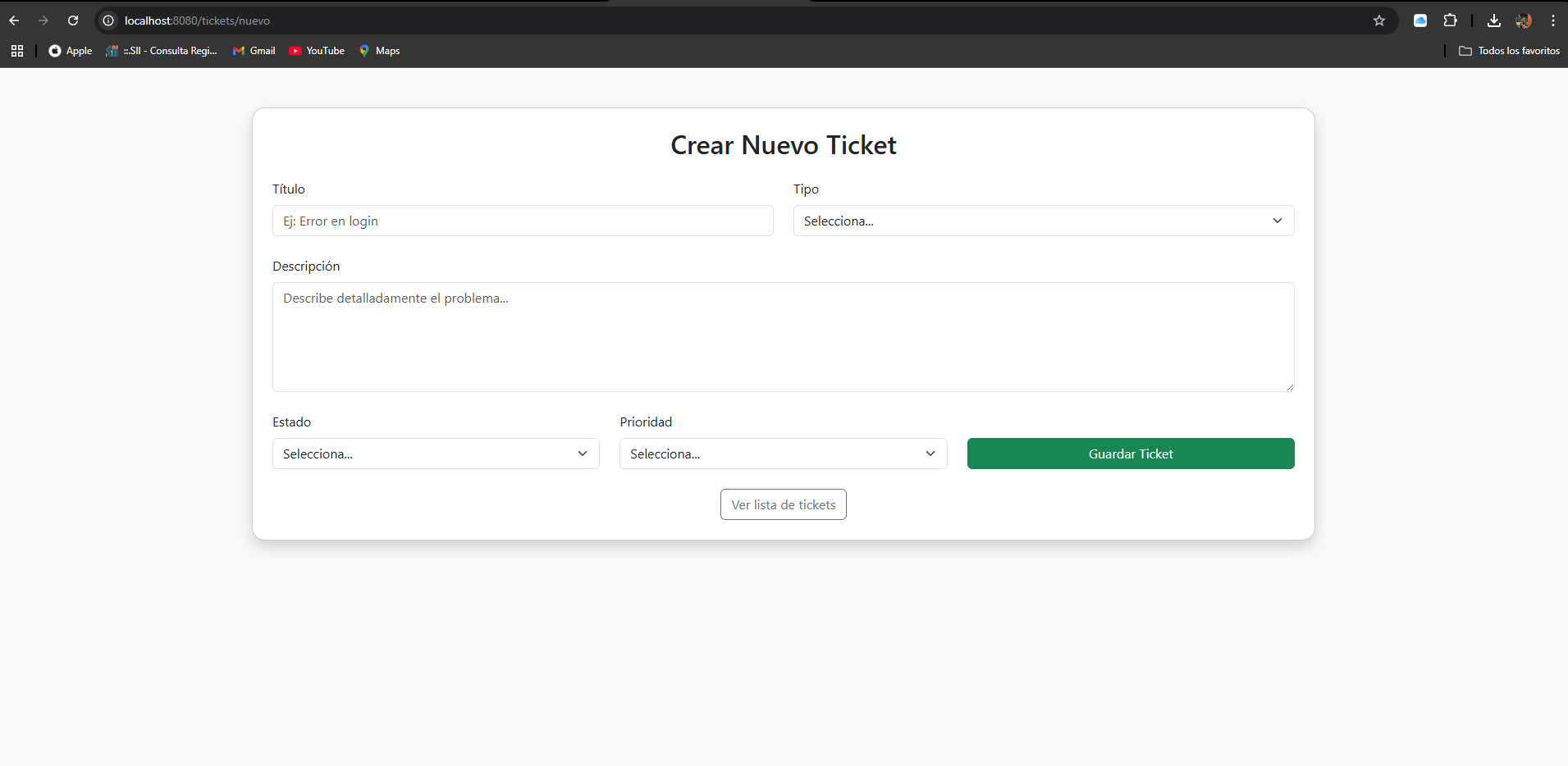


**API-USUARIOS** VISTA FORM POR ID (EL QUE QUIERA PONER, BUSCA A UN USUARIO ESPECÍFICO):

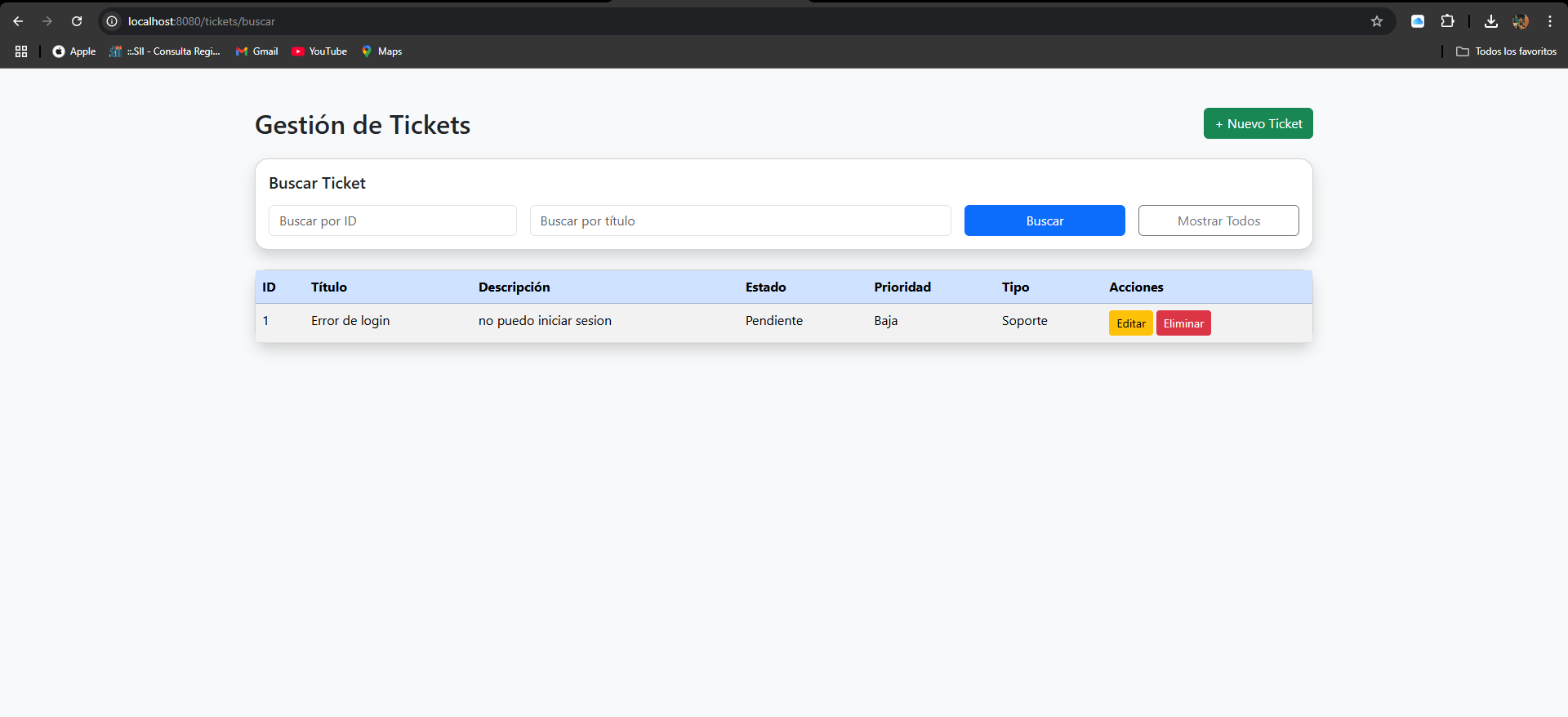
**API-Cursos**



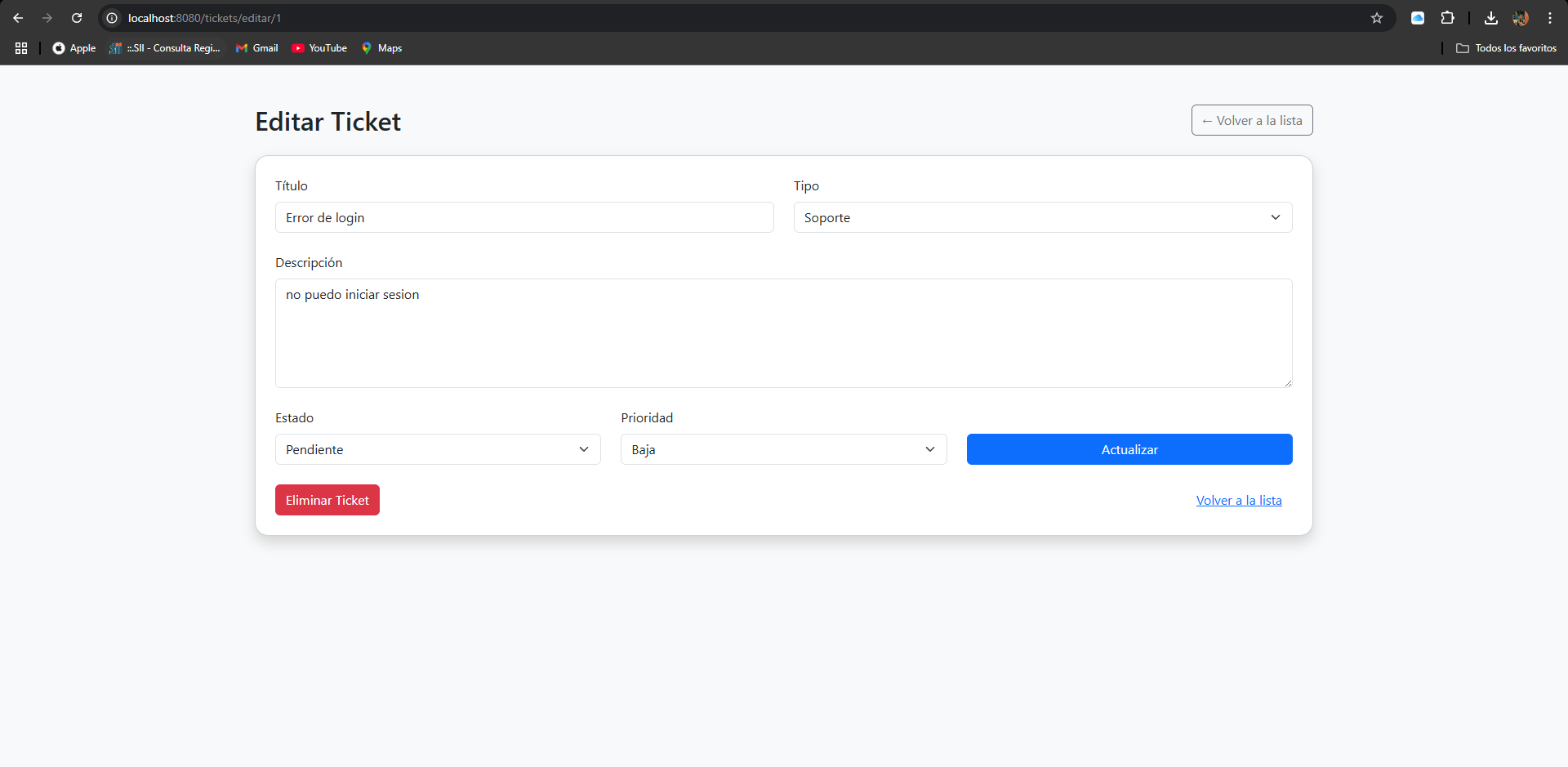
**API-Soporte: Vista “Nuevo-Ticket”**



**API-Soporte: Vista “buscar-lista-tickets”**



**API-Soporte: Vista “editar-ticket”**



# 

# 6. Uso de Git – GitHub.

A continuación se muestra un ejemplo detallado de cómo se ha gestionado el versionado del proyecto con Git y GitHub, explicando los comandos utilizados para subir archivos y cómo se añadió en el archivo README los paths de cada servicio REST con sus peticiones para su ejecución.

Primero, se inicializó el repositorio en la carpeta local del proyecto mediante:

git init

Luego, se configuró el usuario responsable de los commits:

git config user.name "jhonnylarry"

git config user.email "jonathanlaraguibel@gmail.com"

Para subir el proyecto al repositorio remoto, se generó un token de acceso personal en GitHub, el cual fue utilizado como contraseña al realizar el primer push. A continuación, se agregó la URL del repositorio remoto:

git remote add origin https://github.com/dvnxeee/EduTech-Innovators

Luego, se agregaron los archivos del proyecto con:

git add .

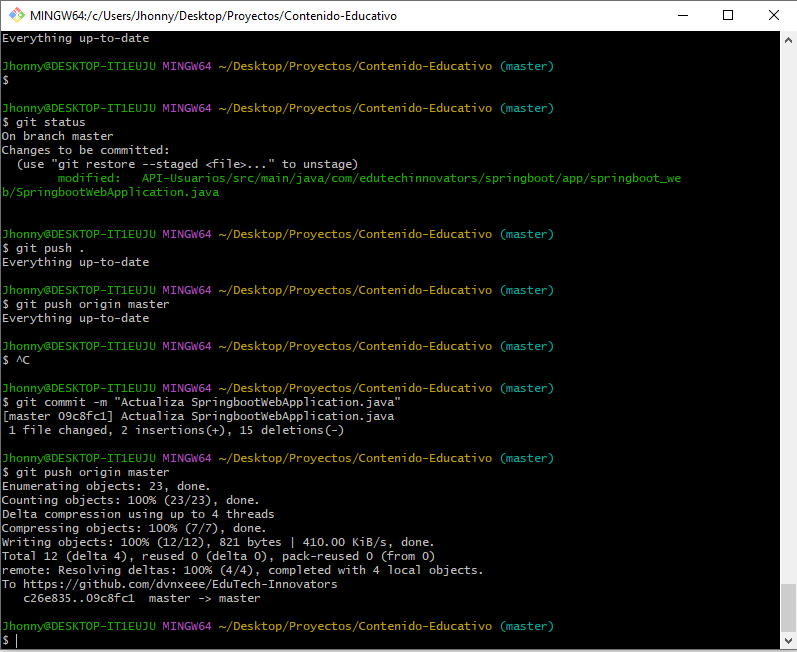
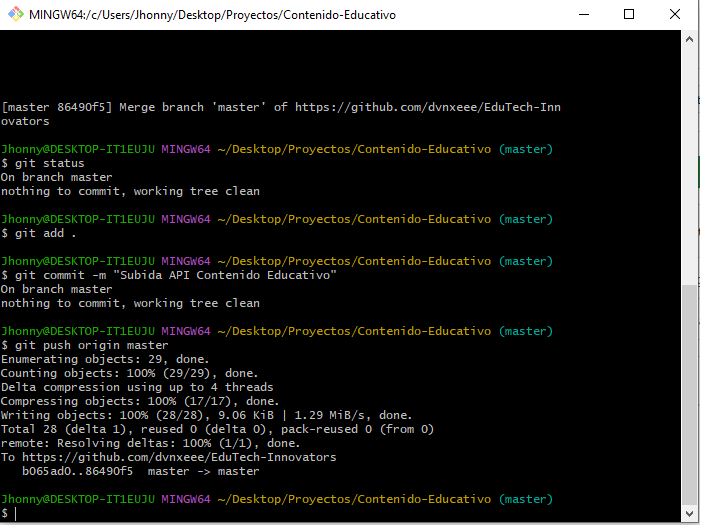
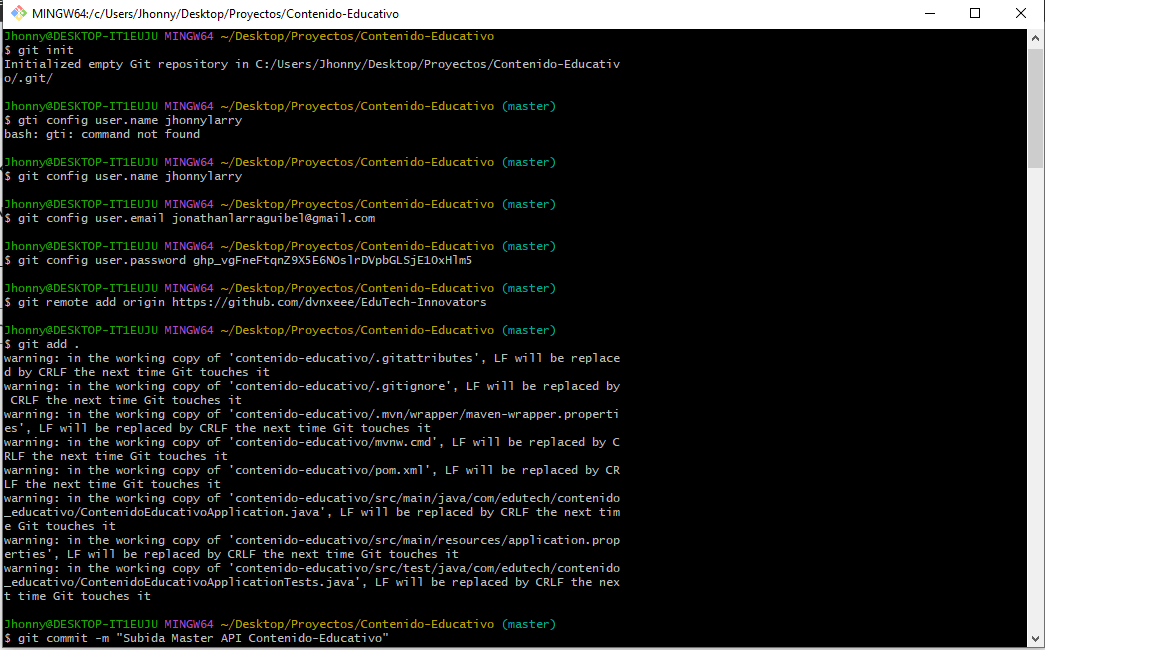
Y se registró un commit representando la primera subida:

git commit -m "Subida Master API Contenido Educativo"

Finalmente, se enviaron los archivos al repositorio remoto:

git push origin master

Este proceso se repitió para nuevas versiones, registrando cambios en clases, archivos de configuración o recursos estáticos, con commits específicos y pushes correspondientes. En las imágenes incluidas en el informe, se visualiza claramente el flujo completo desde la inicialización hasta la subida al repositorio.



# 

# Conclusión

Después de abordar el diseño, desarrollo e implementación del sistema, se puede constatar que la arquitectura basada en microservicios permitió construir una solución web modular, escalable y de fácil mantenimiento. La separación de responsabilidades entre los servicios de Usuarios, Cursos y Soporte facilitó el desarrollo independiente de cada componente, promoviendo un código más limpio y reutilizable.

La integración de tecnologías como Spring Boot, Spring Data JPA y Thymeleaf, junto con una base de datos relacional bien estructurada, permitió cubrir todas las funcionalidades requeridas por el sistema. Además, la exposición de servicios a través de una API REST validada en Postman demostró la correcta comunicación entre las capas del sistema y su capacidad de interactuar con datos reales de manera efectiva.

Por otra parte, la utilización de Git y GitHub permitió gestionar versiones de manera ordenada y profesional, aportando control y trazabilidad en cada etapa del desarrollo. Gracias a todo lo anterior, es posible interpretar que el enfoque adoptado no solo cumple con los objetivos técnicos del proyecto, sino que también sienta una base sólida para su evolución futura y despliegue en entornos reales de producción.

# 