

## Informe Situación Evaluativa 2: EduTech Innovators SPA



### Experiencia 2 – Desarrollo y Operación de Microservicios

Benjamín Torrejón Soto – Alejandra Reyes Duque

Desarrollo Fullstack I 002D

Duoc UC – Ingeniería Informática

Mayo 2025

## Índice

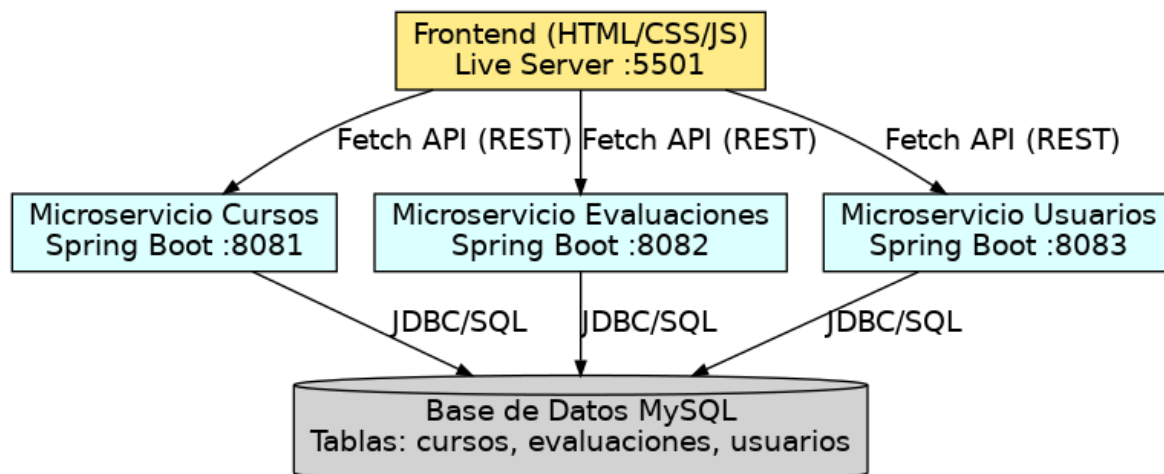
1. Introducción
2. Diagrama de arquitectura de microservicios
3. Estructura del proyecto
  - 3.1. Dependencias
  - 3.2. Componentes implementados
4. Base de datos
  - 4.1. Motor utilizado
  - 4.2. Estructura y tablas
  - 4.3. Ausencia de claves foráneas
  - 4.4. Escalabilidad, rendimiento y seguridad
  - 4.5. Consideraciones para futuras mejoras
5. Implementación de los servicios
  - 5.1. Principios arquitectónicos aplicados
  - 5.2. Flujo lógico en cada microservicio
  - 5.3. Servicios implementados
  - 5.4. Aislamiento e integración
  - 5.5. Validación con Postman
6. Implementación de vistas
  - 6.1. Estructura visual y dinámica JS
  - 6.2. Comunicación con el backend
  - 6.3. Estilos y experiencia visual
  - 6.4. Manejabilidad, modularidad y escalabilidad
  - 6.5. Limitaciones y oportunidades de mejora
  - 6.6. Evaluación general
7. Git y GitHub
8. Conclusión

## 1. *Introducción*

El presente informe documenta el proceso de diseño, desarrollo y validación del proyecto *EduTech Innovators SPA*, el cual representa una plataforma educativa modular desarrollada con tecnologías modernas de backend y frontend. Esta experiencia forma parte de la asignatura Desarrollo Fullstack I, en el contexto de un entorno simulado donde una empresa chilena necesita migrar desde una arquitectura monolítica hacia una estructura basada en microservicios para mejorar su escalabilidad, mantenibilidad y capacidad de respuesta.

La empresa ficticia EduTech, con oficinas en Santiago, Valparaíso y La Serena, enfrenta limitaciones significativas debido a su infraestructura tecnológica obsoleta. Por lo tanto, se plantea una transformación digital orientada al uso de microservicios RESTful y una interfaz ligera que permita la gestión eficiente de usuarios, cursos y evaluaciones.

## 2. *Diagrama de arquitectura de microservicios*



Ejemplo ilustrativo del diagrama de arquitectura (simplificado) para EduTech Innovators SPA. Se aprecia el front-end web (cliente) a la izquierda, que realiza llamadas REST (*Fetch API*) a cada microservicio (centro). Cada microservicio accede únicamente a su propia tabla dentro de la base de datos MySQL (derecha), sin compartir datos directamente con otros servicios.

El proyecto **EduTech Innovators SPA** implementa una arquitectura de microservicios simple y bien delimitada, compuesta por los siguientes componentes:

- **Front-end web estático:** Interfaz de usuario desarrollada en HTML, CSS y JavaScript puro. Se sirve localmente mediante Live Server en <http://localhost:5501>. Este front-end (un navegador web) actúa como cliente que realiza peticiones a los microservicios utilizando *Fetch API* (llamadas REST).
- **Microservicio de Cursos:** Servicio RESTful creado con Spring Boot, desplegado en el puerto 8081. Gestiona la lógica de negocio relacionada con los cursos y está conectado a la tabla de **Cursos** en la base de datos MySQL.
- **Microservicio de Evaluaciones:** Servicio RESTful (Spring Boot) en el puerto 8082, encargado de la lógica de evaluaciones (p. ej. calificaciones). Está vinculado exclusivamente a la tabla **Evaluaciones** en la base de datos MySQL.
- **Microservicio de Usuarios:** Servicio RESTful (Spring Boot) en el puerto 8083, que maneja las operaciones sobre datos de usuarios (crear, consultar, actualizar, eliminar usuarios). Se conecta únicamente a la tabla **Usuarios** de la misma base de datos MySQL.
- **Base de Datos MySQL:** Repositorio de datos relacional que contiene las tablas **Cursos**, **Evaluaciones** y **Usuarios**. Cada microservicio accede solo a *su* tabla correspondiente, sin claves foráneas ni referencias directas entre sí, lo que significa que no hay dependencias a nivel de base de datos entre los servicios.

Esta separación estricta de datos por servicio es intencional y sigue el principio de *database per service* típico en microservicios: cada microservicio maneja su información de forma privada, accesible sólo a través de su propia API REST. Gracias a esto, los servicios están débilmente acoplados y los cambios en el esquema de datos de un servicio no afectan a los demás. Asimismo, **no existe comunicación directa entre microservicios** a nivel de lógica: ninguna de las aplicaciones Spring Boot invoca a otra, sino que **toda interacción ocurre desde el front-end hacia cada microservicio** por separado, vía solicitudes HTTP (*fetch*). En un sistema pequeño, esta estrategia de comunicación directa cliente–microservicio es suficiente y simplifica la arquitectura (evitando la introducción de un gateway u otra complejidad)

### 3. *Estructura del proyecto*

#### 3.1. Dependencias

Durante el desarrollo del proyecto, utilizamos Maven como herramienta principal para gestionar las dependencias y facilitar el ciclo de vida del proyecto. Maven nos permitió declarar las librerías necesarias en el archivo pom.xml, evitando la configuración manual y asegurando la compatibilidad entre los distintos módulos y bibliotecas.

Las dependencias principales incluyeron:

- Spring-boot-starter-web: para crear los endpoints REST y habilitar la comunicación entre los microservicios.
- spring-boot-starter-data-jpa: para la gestión de las bases de datos con JPA (Java Persistence API), simplificando las consultas y operaciones CRUD.
- mysql-connector-java: para la conexión a las bases de datos MySQL de cada microservicio.
- spring-boot-devtools: para mejorar la experiencia de desarrollo, permitiendo recargas automáticas durante la codificación.

La configuración de estas dependencias permitió establecer un entorno estable y consistente para el desarrollo de los microservicios. Además, facilitó la integración continua y las futuras actualizaciones de bibliotecas, asegurando

que el proyecto se mantenga compatible con las últimas versiones de las herramientas.

### 3.2. Componentes implementados

En el desarrollo de los microservicios se implementaron varios componentes que son fundamentales en la arquitectura de Spring Boot y que aseguran el correcto funcionamiento del sistema, su modularidad y su escalabilidad. Estos componentes son:

- **Controllers (Controladores)**
  - Exponen los endpoints REST del sistema.
  - Su función principal es recibir las peticiones HTTP (GET, POST, PUT, DELETE) y delegar la lógica de negocio al Service correspondiente.
  - Contienen anotaciones como `@RestController`, `@RequestMapping` y `@CrossOrigin` para permitir peticiones externas.
- **Services (Servicios)**
  - Representan la capa de lógica de negocio de cada microservicio.
  - Su función es procesar las peticiones que llegan desde los controladores y decidir cómo interactuar con la base de datos (a través de los Repositories).
  - Tienen la anotación `@Service` y aseguran la separación de responsabilidades, manteniendo un código más limpio y fácil de mantener.

- Entities (Entidades)
  - Son clases que modelan las tablas de la base de datos.
  - Utilizan anotaciones como `@Entity`, `@Table`, `@Id` y `@GeneratedValue` para definir la estructura de las tablas y sus relaciones.
  - Ejemplo:
    - `Usuario.java` representa la tabla usuarios.
    - `Curso.java` representa la tabla cursos.
    - `Evaluacion.java` representa la tabla evaluaciones.
- Repositories (Repositorios)
  - Son interfaces que extienden de `JpaRepository` o `CrudRepository`, permitiendo realizar operaciones CRUD sin escribir consultas SQL manualmente.
  - Utilizan la anotación `@Repository` y permiten que Spring gestione automáticamente la persistencia de datos.
  - Ejemplo:
    - `UsuarioRepository`
    - `CursoRepository`
    - `EvaluacionRepository`

Cada uno de los componentes utilizados contribuyen a:

- La modularidad: Separando las responsabilidades y facilitando la escalabilidad.
- La mantenibilidad: Permitiendo modificaciones sin impactar todo el sistema.
- La facilidad de pruebas: Ya que cada componente está bien definido y desacoplado.

## 4. *Base de datos*

La persistencia de datos en el proyecto *EduTech Innovators SPA* se sustenta en una arquitectura de microservicios desacoplada, donde cada servicio gestiona su propia base de datos de forma independiente. Esta decisión técnica permite garantizar la autonomía de cada módulo, evitar cuellos de botella en el acceso a datos, y facilitar la evolución y el mantenimiento de la información en entornos distribuidos.

### 4.1. Motor utilizado

Para el almacenamiento y gestión de los datos se seleccionó el sistema de gestión de bases de datos MySQL 8.x, una tecnología madura, estable y ampliamente adoptada en la industria. Las razones de esta elección incluyen:

- Compatibilidad nativa con Spring Data JPA, lo que facilita su integración con el ecosistema Spring Boot sin necesidad de configuraciones complejas.
- Soporte para el motor InnoDB, que proporciona características avanzadas como transacciones ACID, bloqueo a nivel de fila, claves únicas e integridad referencial cuando se requiere.
- Ecosistema de herramientas complementarias, como MySQL Workbench para la visualización y administración gráfica, y compatibilidad con servicios cloud (AWS RDS, Google Cloud SQL, etc.).
- Rendimiento optimizado para consultas concurrentes y cargas moderadas, ideal para una arquitectura de microservicios como la desarrollada.

Cada microservicio posee su propia instancia de conexión, definida en el archivo `application.properties`, con credenciales y URL independientes, reforzando el principio de separación de contextos de datos.



Configuración típica:

properties

- `spring.datasource.url=jdbc:mysql://localhost:3306/edudb`
- `spring.datasource.username=eduuser`
- `spring.datasource.password=eduP@ssw0rd`
- `spring.jpa.hibernate.ddl-auto=validate`
- `spring.datasource.hikari.maximum-pool-size=10`

El parámetro `ddl-auto=validate` fue elegido específicamente para entornos de validación o producción, ya que impide que Hibernate modifique el esquema automáticamente, obligando a que cualquier alteración se realice de forma controlada mediante scripts de migración (ej. Flyway o Liquibase).

Se utilizó además el pool de conexiones HikariCP, incluido por defecto en Spring Boot, con parámetros configurados para soportar múltiples peticiones simultáneas y evitar conexiones inactivas.

#### 4.2. Estructura y tablas

Cada microservicio opera sobre una tabla principal que representa su dominio funcional:

- **usuarios:** almacena información básica del usuario (id, nombre, correo, contraseña).
- **cursos:** mantiene el detalle de los cursos ofrecidos (id, nombre, descripción, instructor, duración).
- **evaluaciones:** gestiona los registros de evaluaciones por estudiante, incluyendo sus notas y promedio (id, nombre, curso, nota1, nota2, nota3, promedio, estudiante).

Características comunes de las tablas:

- Motor de almacenamiento: InnoDB (ideal para transacciones seguras y concurrencia).
- Codificación de caracteres: utf8mb4, que permite almacenar texto en múltiples idiomas y emojis.
- Colaciones: utf8mb4\_unicode\_ci, para una comparación insensible a mayúsculas y acentos.
- Clave primaria: columna id autoincremental en cada tabla.
- Índices recomendados: en campos como correo (usuarios) o curso (evaluaciones), para optimizar búsquedas y garantizar unicidad lógica.

#### 4.3. Ausencia de claves foráneas

Un aspecto clave del diseño fue evitar el uso de claves foráneas explícitas entre tablas, en favor de una arquitectura desacoplada. Este enfoque responde directamente al paradigma de microservicios, donde cada módulo debe ser lo más autónomo posible.

Por ejemplo, en lugar de tener un `id_curso` como clave foránea en la tabla `evaluaciones`, se almacena el nombre del curso como texto. Esto evita dependencias estructurales entre las bases de datos, simplifica el despliegue independiente de los servicios y facilita escenarios donde cada servicio se ejecuta en su propio contenedor o infraestructura.

Cualquier validación cruzada como verificar si el nombre de un curso existe se realizaría desde el frontend o mediante servicios de orquestación en una versión futura, manteniendo así el principio de bajo acoplamiento.

#### 4.4. Escalabilidad, rendimiento y seguridad

Gracias al diseño modular de las bases de datos, el sistema está preparado para:

- Escalar horizontalmente los servicios de manera independiente.
- Realizar particionamiento lógico o réplicas si se requieren configuraciones de alta disponibilidad.
- Aplicar controles de acceso a nivel de usuario en cada instancia de base de datos, permitiendo que cada microservicio tenga únicamente los permisos necesarios (lectura, escritura, etc.).

Este enfoque minimiza los riesgos de corrupción de datos, mejora la auditabilidad y permite adoptar prácticas de seguridad más estrictas conforme el sistema evolucione.

#### 4.5. Consideraciones para futuras mejoras

Entre las mejoras que podrían implementarse en una versión más avanzada del proyecto, se identifican:

- Auditoría de operaciones: con tablas de bitácora o triggers para registrar acciones de modificación o eliminación.
- Normalización avanzada: si se agregan más entidades relacionadas (por ejemplo, roles de usuario o categorías de cursos).
- Migraciones controladas: incorporando herramientas como Flyway para aplicar versiones del esquema de manera segura y automatizada.

## 5. *Implementación de los servicios*

La implementación de los servicios en el proyecto *EduTech Innovators SPA* se realizó bajo el paradigma de arquitectura de microservicios, permitiendo la separación de responsabilidades por dominio funcional: gestión de usuarios, administración de cursos y evaluaciones académicas. Cada uno de estos dominios fue desarrollado como un servicio RESTful independiente, ejecutándose en su propio entorno, con su base de datos, puerto de comunicación y lógica de negocio.

Este enfoque garantiza un desarrollo escalable y mantenible, donde cada servicio puede evolucionar de forma autónoma sin afectar al resto del sistema. Asimismo, facilita la trazabilidad, depuración y pruebas unitarias, al estar claramente definidos los límites funcionales de cada microservicio.

### 5.1. Principios arquitectónicos aplicados

Desde el inicio del desarrollo se establecieron principios de diseño orientados a mantener un sistema robusto, eficiente y de fácil mantenimiento. Entre ellos:

- Separación de responsabilidades: cada componente del microservicio (controlador, servicio, repositorio) cumple una función específica dentro de la lógica del flujo de datos.
- Encapsulamiento: cada servicio gestiona su propia persistencia, sin depender directamente de otros módulos o tablas externas.
- Validación anticipada: las clases modelo utilizan anotaciones de Bean Validation (`@NotBlank`, `@Email`, etc.), lo que asegura que los datos ingresados cumplan reglas mínimas antes de ser procesados.
- Control centralizado de errores: se implementó una clase de tipo `@ControllerAdvice` que permite capturar y gestionar excepciones de forma global, enviando respuestas significativas y con códigos HTTP apropiados.

- Uso de estándares REST: todos los servicios exponen sus operaciones siguiendo buenas prácticas RESTful, utilizando los métodos HTTP correctos (GET, POST, PUT, DELETE) y recursos con rutas significativas.

## 5.2. Flujo lógico en cada microservicio

Cada microservicio maneja sus operaciones CRUD (crear, leer, actualizar y eliminar) mediante una serie de componentes bien definidos:

- El Controlador expone los endpoints y recibe las solicitudes del cliente.
- El Servicio ejecuta la lógica de negocio, aplicando validaciones y transformaciones cuando corresponda.
- El Repositorio accede a la base de datos utilizando la interfaz JpaRepository, que permite ejecutar operaciones sin necesidad de escribir código SQL directamente.

Este flujo permite separar el qué se hace (la lógica de negocio) del cómo se almacena (la lógica de persistencia), lo cual es crucial para mantener la claridad del sistema y facilitar su prueba.

## 5.3. Servicios implementados

### Microservicio de Usuarios

Permite registrar nuevos usuarios, obtener todos los usuarios existentes, buscar por ID, actualizar datos y eliminar registros. Se priorizó la validación de correo electrónico y la protección básica de campos requeridos. La tabla usuarios mantiene la información asociada a cada persona que interactúa con la plataforma.

## Microservicio de Cursos

Este servicio gestiona la oferta educativa de la plataforma. Se pueden registrar cursos nuevos, definir su descripción, duración e instructor, así como listar, editar o eliminar cada uno. La lógica fue diseñada para permitir futuras extensiones como asignación por niveles o categorías.

## Microservicio de Evaluaciones

En este módulo se implementa el registro y seguimiento de las calificaciones de los estudiantes. Una de las particularidades de este servicio es que al recibir las tres notas (nota1, nota2 y nota3), se calcula automáticamente el promedio y se guarda como atributo adicional. Este cálculo se realiza en la capa de servicio, asegurando coherencia entre los datos ingresados y almacenados.

### 5.4. Aislamiento e integración

Cada servicio se ejecuta en un puerto distinto y conecta a su propia base de datos. Esta independencia garantiza que el fallo de un servicio no afecta directamente a los demás. No se utilizan claves foráneas entre las tablas, lo cual refuerza el principio de bajo acoplamiento. Las relaciones, cuando son necesarias, se manejan a nivel de aplicación (por ejemplo, haciendo referencia al nombre del curso en las evaluaciones en lugar de un ID foráneo).

### 5.5. Extensibilidad futura

Gracias a la estructura modular adoptada, el sistema puede ampliarse fácilmente. Por ejemplo, podría incorporarse un microservicio para manejo de asistencia, reportes académicos, notificaciones o autenticación, sin alterar los servicios existentes. Esta escalabilidad está garantizada por el enfoque desacoplado y estandarizado en el desarrollo de cada servicio.

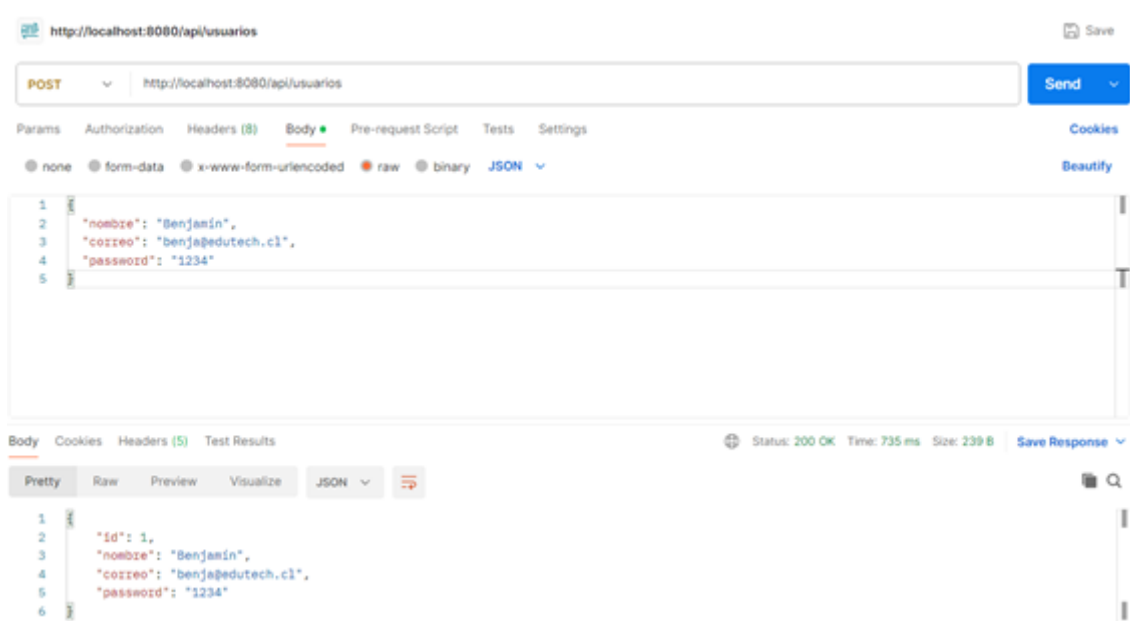
### 5.6. Validación con Postman

Para garantizar que los microservicios funcionan correctamente y exponen sus endpoints según las especificaciones REST, se realizó un conjunto de pruebas utilizando Postman, una herramienta ampliamente empleada para la validación de APIs.

A través de Postman, se ejecutaron las operaciones básicas (GET, POST, PUT y DELETE) de cada microservicio, verificando la correcta respuesta del servidor, los códigos de estado HTTP y la integridad de los datos almacenados o recuperados de la base de datos.

## Evidencias postman (usuarios)

1.-



2.-

http://localhost:8080/api/usuarios

GET http://localhost:8080/api/usuarios

Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Bulk Edit
Key	Value	

Body Cookies Headers (5) Test Results Status: 200 OK Time: 33 ms Size: 241 B Save Response

Pretty Raw Preview Visualize

```
[{"id":1,"nombre":"Benjamia","correo":"benja@edutech.cl","password":"1234"}]
```

3.-

POST http://localhost:8080/api/ GET http://localhost:8080/api/ GET http://localhost:8080/api/ PUT http://localhost:8080/api/ DEL http://localhost:8080/api/ + ...

http://localhost:8080/api/usuarios/1

GET http://localhost:8080/api/usuarios/1

Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Bulk Edit
Key	Value	

Body Cookies Headers (5) Test Results Status: 200 OK Time: 69 ms Size: 239 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {"id": 1,  
2  "nombre": "Benjamín",  
3  "correo": "benja@edutech.cl",  
4  "password": "1234"  
5 }  
6
```



4.-


The screenshot shows a REST client interface with the following components:

- URL Bar:** `http://localhost:8080/api/usuarios/1`
- Method:** `PUT`
- Body Tab:** Selected, showing a JSON payload:

```
1 {  
2   "nombre": "Benjamín Torrejón",  
3   "correo": "benjamin@edutech.cl",  
4   "password": "admin123"  
5 }
```
- Response Tab:** Selected, showing the response body in JSON format:

```
{ "id": 1, "nombre": "Benjamín Torrejón", "correo": "benjamin@edutech.cl", "password": "admin123" }
```

5.-

 **http://localhost:8080/api/usuarios/1**

**DELETE** ▾

http://localhost:8080/api/usuarios/1

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Query Params

	Key	Valu
	Key	Valu

Body

Cookies

Headers (4)

Test Results


Pretty

Raw

Preview

Visualize

Text ▾



1

## Evidencias postman (cursos)

1.-

The screenshot shows a Postman interface for a GET request to `http://localhost:8081/api/cursos`. The request is configured with the following parameters:

Key	Value
Key	Value

The response is displayed in the 'Body' tab, showing a JSON object with the following structure:

```
1 [
2   {
3     "id": 1,
4     "nombre": "Curso de Java Spring",
5     "descripcion": "Aprende a hacer APIs REST con Spring Boot",
6     "instructor": null,
7     "duracion": 60
8   }
9 ]
```

The status of the request is 200, indicated by the 'Status: 2' label in the bottom right corner.

2.-

POST

▼

http://localhost:8081/api/cursos

ParamsAuthorizationHeaders (8)Body●Pre-request ScriptTestsSettings

● none

● form-data

● x-www-form-urlencoded

● raw

● binary

JSON ▼

1

2

3

4

5

200

200

```
"nombre": "Curso de Java Spring",
"descripcion": "Aprende a hacer APIs REST con Spring Boot",
"duracion": 60
```

BodyCookiesHeaders (5)Test Results

PrettyRawPreviewVisualizeJSON ▼

⌵

1

2

3

4

5

6

7

200

200

```
"id": 1,
"nombre": "Curso de Java Spring",
"descripcion": "Aprende a hacer APIs REST con Spring Boot",
"instructor": null,
"duracion": 60
```

3.-


The screenshot shows a REST client interface with the following details:

- URL:** `http://localhost:8081/api/cursos/1`
- Method:** `PUT`
- Body:** A JSON object with the following fields:

```
{  "id": 1,  "nombre": "Curso actualizado",  "descripcion": "Contenido actualizado",  "duracion": 75}
```
- Response:** Status `200 OK`. The response body is a JSON object with the following fields:

```
{  "id": 1,  "nombre": "Curso actualizado",  "descripcion": "Contenido actualizado",  "instructor": null,  "duracion": 75}
```

4.-

 **http://localhost:8081/api/cursos/1**

**DELETE** ▾

http://localhost:8081/api/cursos/1

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Query Params

	Key
	Key

Body

Cookies

Headers (4)

Test Results


Pretty

Raw

Preview

Visualize

Text ▾



1

## Evidencias postman (Evaluaciones)

1.-

The screenshot displays a Postman interface for an API endpoint. At the top, the URL bar shows `http://localhost:8082/api/evaluaciones`. Below it, the request method is set to `GET`. The 'Params' tab is active, showing a table for query parameters:

Key	Value
Key	Value

Below the parameters, the 'Body' tab is selected, showing the response in JSON format. The status bar at the top right indicates `Status: 200 OK`. The JSON response is as follows:

```
1 {
2   {
3     "id": 2,
4     "nombre": "Prueba Unidad 1",
5     "curso": "Programación Orientada a Objetos",
6     "nota": 6.3,
7     "ponderacion": 30,
8     "estudiante": "Sofía Muñoz"
9   },
10  {
11    "id": 3,
12    "nombre": "Prueba Unidad 1",
13    "curso": "Programación Orientada a Objetos",
14    "nota": 6.3,
15    "ponderacion": 30,
16    "estudiante": "Sofía Muñoz"
17  }
18 }
```

2.-

The screenshot shows a REST client interface with a POST request to `http://localhost:8082/api/evaluaciones`. The request body is a JSON object with the following fields: `nombre`, `curso`, `nota`, `ponderacion`, and `estudiante`. The response body is a JSON object that includes an additional `id` field.

**Request:**


```
1  {
2    "nombre": "Prueba Unidad 1",
3    "curso": "Programación",
4    "nota": 6.3,
5    "ponderacion": 30,
6    "estudiante": "Catalina Vera"
7  }
```

**Response:**

```
1  {
2    "id": 3,
3    "nombre": "Prueba Unidad 1",
4    "curso": "Programación",
5    "nota": 6.3,
6    "ponderacion": 30,
7    "estudiante": "Catalina Vera"
8  }
```



3.-

 **http://localhost:8082/api/evaluaciones/3**

PUT

▼

http://localhost:8082/api/evaluaciones/3

Params   Authorization   Headers (8)   **Body** ●   Pre-request Script   Tests   Settings

☐ none

☐ form-data

☐ x-www-form-urlencoded

☒ raw

☐ binary

JSON ▼

```
1  {
2    "nombre": "Prueba Unidad 1 - Actualizada",
3    "curso": "Programación Avanzada",
4    "nota": 5.8,
5    "ponderacion": 40,
6    "estudiante": "Catalina Vera"
7  }
8
```

Body   Cookies   Headers (8)   Test Results


Pretty   Raw   Preview   Visualize

JSON ▼



```
1  {
2    "id": 3,
3    "nombre": "Prueba Unidad 1 - Actualizada",
4    "curso": "Programación Avanzada",
5    "nota": 5.8,
6    "ponderacion": 40,
7    "estudiante": "Catalina Vera"
8  }
```

4.-

 <http://localhost:8082/api/evaluaciones/2>

DELETE

▼

http://localhost:8082/api/evaluaciones/2

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Query Params

	Key	Value
	Key	Value

Body

Cookies

Headers (6)

Test Results


Pretty

Raw

Preview

Visualize

Text ▼



1

## 6. *Implementación de vistas*

La interfaz de usuario de *EduTech Innovators SPA* fue desarrollada utilizando tecnologías web estándar: HTML, CSS y JavaScript puro. La decisión de prescindir de frameworks o bibliotecas externas se tomó con el objetivo de entender y controlar de forma total el flujo de interacción con los microservicios backend, así como optimizar la carga y el rendimiento en entornos locales o de baja latencia.

El frontend se implementó bajo un enfoque modular y dinámico, donde cada microservicio tiene su propio archivo JavaScript responsable de consumir sus respectivas APIs REST y renderizar los datos de forma interactiva en el navegador.

### 6.1. Estructura visual y dinámica JS

La estructura de la vista principal está organizada en un único archivo `index.html`, que actúa como punto de entrada de toda la aplicación. Esta página incluye:

- Un encabezado estático con el título “Plataforma EduTech”.
- Tres botones que representan los módulos disponibles: Usuarios, Cursos y Evaluaciones.
- Un contenedor main donde se inserta dinámicamente el contenido proveniente de cada módulo.
- Un archivo `main.js` que gestiona la navegación interna y decide qué módulo cargar según la interacción del usuario.

Cuando el usuario selecciona un módulo, el archivo `main.js` inyecta el script correspondiente (`usuarios.js`, `cursos.js` o `evaluaciones.js`) dentro de la página. Esta técnica evita recargas innecesarias y mantiene el sitio en una sola página (SPA parcial sin enrutamiento), mejorando la velocidad de navegación.

Cada módulo JavaScript contiene funciones específicas que:

- Realizan llamadas fetch al microservicio correspondiente mediante solicitudes GET.
- Procesan las respuestas en formato JSON.
- Generan una tabla HTML con los datos obtenidos.
- Insertan dicha tabla dentro del contenedor principal de la página.

## 6.2. Comunicación con el backend

El frontend se comunica directamente con los microservicios, los cuales están levantados en distintos puertos locales:

- Usuarios: <http://localhost:8083/api/usuarios>
- Cursos: <http://localhost:8081/api/cursos>
- Evaluaciones: <http://localhost:8082/api/evaluaciones>

Cada una de estas rutas es consumida desde el navegador mediante Fetch API, que permite enviar solicitudes HTTP de manera asincrónica. Los datos recibidos son renderizados dinámicamente, permitiendo al usuario visualizar el estado actual de la base de datos sin necesidad de una recarga completa del sitio.

Los métodos utilizados hasta esta etapa son principalmente operaciones de lectura (GET), aunque la estructura está preparada para permitir en futuras versiones la implementación de formularios con operaciones POST, PUT y DELETE.

### 6.3. Estilos y experiencia visual

El archivo `styles.css` contiene los estilos globales de la aplicación, implementando principios de diseño limpio, responsivo y legible. Algunas de las características visuales incluyen:

- Paleta de colores sobria y profesional.
- Botones con efecto hover para mejorar la interacción.
- Tablas con borde y espaciado uniforme para facilitar la lectura de datos.
- Tipografías sin serif, que garantizan buena visibilidad en pantallas pequeñas.

Asimismo, se aplicó un enfoque *mobile-first*, lo cual permite que la interfaz sea funcional y legible tanto en dispositivos móviles como en pantallas de escritorio.

### 6.4. Manejabilidad, modularidad y escalabilidad

Uno de los logros del frontend fue construirlo de forma modular, permitiendo que cada script sea responsable exclusivamente de un microservicio. Esto facilita el mantenimiento del sistema, ya que errores o modificaciones en un módulo no afectan a los demás.

Además, esta estructura permite:

- Incluir nuevos microservicios con facilidad, simplemente creando un nuevo archivo `.js` y agregando un botón correspondiente.
- Separar la lógica de presentación de la lógica de comunicación, cumpliendo indirectamente principios como el de responsabilidad única.

## 6.5. Limitaciones y oportunidades de mejora

Si bien el frontend cumple su función de consumir y mostrar datos desde los microservicios, existen aspectos que podrían optimizarse en futuras versiones:

- Validaciones del lado del cliente: actualmente no se implementan formularios ni validación de entrada, lo cual será fundamental en una versión interactiva con creación o modificación de registros.
- Manejo avanzado de errores: por ahora, los errores de red o respuestas vacías se muestran por consola. Se podría incluir una visualización amigable en la interfaz (alerts o modales).
- Paginación y filtrado: en caso de que el volumen de datos crezca, será necesario incorporar herramientas de paginación, búsqueda y filtrado para mejorar la usabilidad.
- Frameworks frontend modernos: aunque se optó por JavaScript puro, en el futuro se podría evaluar el uso de frameworks como React o Vue.js para mejorar el control del estado y la reutilización de componentes.

## 6.6. Evaluación general

El desarrollo de esta capa visual permitió no solo probar el funcionamiento de los microservicios, sino también experimentar con buenas prácticas de diseño web, consumo de APIs y creación de interfaces dinámicas. La simplicidad del enfoque fue una decisión consciente que, lejos de ser una limitación, permitió un mayor entendimiento del flujo entre cliente y servidor sin depender de abstracciones externas.

## **7. Git y GitHub**

Durante el desarrollo del proyecto, utilizamos Git para gestionar el control de versiones y facilitar la colaboración entre los integrantes. Además, empleamos GitHub para centralizar el repositorio, revisar el código y manejar las integraciones de cambios mediante commits. Aunque al principio nos costó adaptarnos a estas herramientas, logramos superarlo y resolver los problemas que surgieron en el proceso. Esto permitió trabajar de forma más organizada y ordenada, mejorando la comunicación y reduciendo errores durante todo el ciclo de desarrollo.

## **8. Conclusión**

El proyecto EduTech Innovators SPA permitió aplicar de forma práctica los conceptos aprendidos en la asignatura, llevando a cabo la transformación de un sistema monolítico a una arquitectura basada en microservicios. A través de la separación de responsabilidades, la validación de datos, y el desarrollo de vistas dinámicas y modulares, logramos construir un sistema más escalable y mantenible.

La experiencia de desarrollo con herramientas como Spring Boot, MySQL, y GitHub nos permitió consolidar conocimientos técnicos y de colaboración, enfrentando desafíos relacionados con la integración de servicios y la optimización de la comunicación entre los distintos componentes.

Si bien el sistema actual funciona correctamente en un entorno de pruebas, identificamos áreas de mejora, como la validación de formularios en el frontend, la implementación de seguridad avanzada y la integración de herramientas de migración de base de datos.

Esta experiencia nos permitió comprender de manera integral la importancia de la modularidad, la documentación, y las buenas prácticas de desarrollo en la construcción de sistemas web modernos y orientados a microservicios.