Informe Situación Evaluativa 3: EduTech Innovators SPA

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.Experiencia 3 – Desarrollo y Operación de Microservicios

Benjamín Torrejón Soto – Alejandra Reyes Duque

Desarrollo Fullstack I 002D

Duoc UC – Ingeniería Informática

Junio 2025

***1.- Introducción***

En esta tercera entrega, el proyecto EduTech Innovators SPA avanza desde una arquitectura monolítica inicial hacia un ecosistema completamente modular basado en microservicios. En las Experiencias 1 y 2 se desarrollaron los servicios de Usuarios, Cursos y Evaluaciones, los cuales sentaron las bases funcionales del sistema. En esta nueva fase se incorporaron cuatro microservicios adicionales: Inscripciones, Contenido, Pagos y Soporte, ampliando significativamente las capacidades de la plataforma. Además, se optimizaron las APIs existentes mediante la unificación de rutas bajo una misma convención REST (/api/[recurso]), la mejora del manejo de errores y la incorporación de Swagger para la documentación automática. Estos avances permitieron implementar pruebas unitarias e integración de manera rigurosa, fortaleciendo la calidad, mantenibilidad y escalabilidad del sistema.

***2.- Estructura del proyecto y configuración***

La solución EduTech Innovators SPA está organizada como un conjunto de **siete microservicios autónomos**: Usuarios, Cursos, Evaluaciones, Inscripciones, Contenido, Pagos y Soporte. Los tres primeros (Usuarios, Cursos y Evaluaciones) fueron desarrollados durante las Experiencias 1 y 2, mientras que los cuatro restantes se incorporaron en la presente etapa. Cada microservicio cuenta con su propio módulo Maven o carpeta raíz y una estructura de paquetes estandarizada: entities para las clases JPA que mapean las tablas, repositories para las interfaces que extienden JpaRepository, services para la lógica de negocio anotada con @Service, controllers para las APIs REST expuestas bajo la ruta /api/[recurso], y config para clases globales como CorsConfig.java. Cada módulo define en su archivo application.properties un puerto exclusivo (por ejemplo, 8081 para Cursos, 8082 para Evaluaciones, 8083 para Usuarios, hasta 8087 para Contenido), así como su propia configuración de conexión a una base de datos MySQL aislada (jdbc:mysql://localhost:3306/edutech\_[servicio]) y el uso de HikariCP como pool de conexiones. En el pom.xml de cada microservicio se incluyen dependencias comunes como spring-boot-starter-web, spring-boot-starter-data-jpa, mysql-connector-java, springdoc-openapi-starter-webmvc-ui para la documentación OAS/Swagger, y spring-boot-starter-test para las pruebas automatizadas con JUnit 5, Mockito y MockMvc. Esta arquitectura modular asegura coherencia, escalabilidad y cumplimiento con las buenas prácticas definidas en la rúbrica.

***3.- Diagrama de arquitectura de microservicios***

**Puertos asignados y flujos de comunicación**

**Cursos (8081)**: expone /api/cursos. Gestiona alta, baja y listados de cursos; es consumido por Inscripciones y también consultado desde el frontend para mostrar la oferta académica.

**Evaluaciones (8082)**: expone /api/evaluaciones. Registra y consulta notas de los estudiantes, basándose en la información proveniente del servicio de Usuarios.

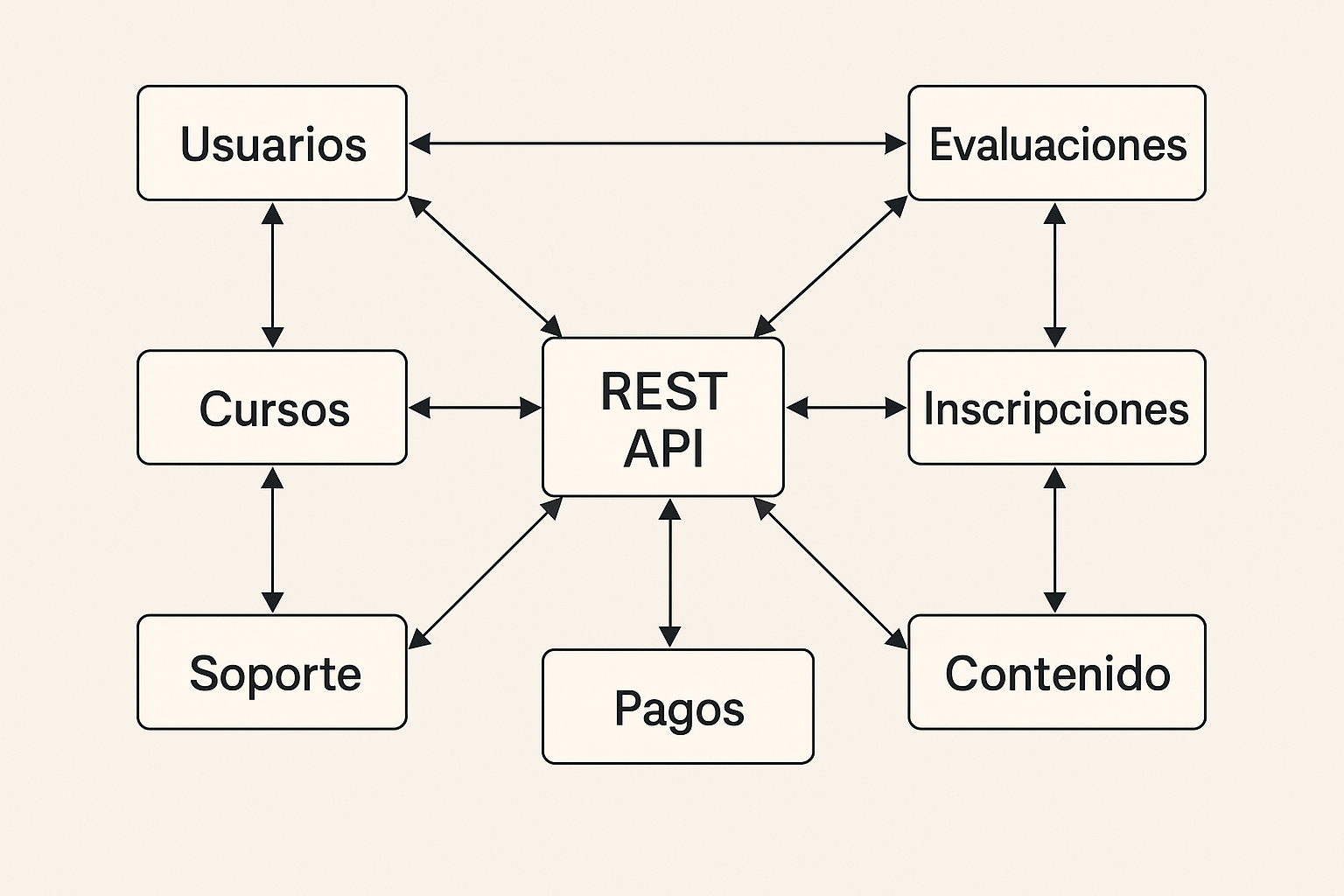
**Usuarios (8083)**: expone /api/usuarios. Administra el registro y consulta de perfiles; su información es utilizada por Inscripciones y Evaluaciones para validar alumnos.

**Inscripciones (8084)**: expone /api/inscripciones. Coordina la relación entre Usuarios y Cursos, validando su existencia antes de registrar la inscripción; además, genera solicitudes hacia el servicio de Pagos.

**Pagos (8085)**: expone /api/pagos. Procesa las transacciones vinculadas a Inscripciones; al recibir confirmación, actualiza el estado de inscripción a pagada.

**Soporte (8086)**: expone /api/soporte. Permite la creación de tickets de atención por parte de los usuarios, gestionando mensajes y el estado de resolución correspondiente.

**Contenido (8087)**: expone /api/contenido. Administra materiales académicos asociados a los cursos (documentos, enlaces, archivos multimedia); puede consultarse desde el frontend y relacionarse con las Inscripciones.



***Nota*:** La arquitectura completa y los flujos de comunicación entre microservicios pueden visualizarse en el diagrama incluido, donde se evidencian las dependencias cruzadas y la interacción con el frontend mediante REST.

***4.- Estructura del proyecto y configuración***

La solución **EduTech Innovators SPA** se estructura en **siete microservicios independientes**, cada uno respetando una convención de paquetes que favorece la claridad y mantenibilidad del código: entities para las clases de dominio con anotaciones JPA, repositories para interfaces que extienden JpaRepository, services para la lógica de negocio anotada con @Service, controllers para exponer endpoints REST bajo la convención /api/[recurso], config para clases como CorsConfig que habilitan la comunicación entre frontend y backend, y dto cuando es necesario estandarizar solicitudes y respuestas JSON. Cada microservicio cuenta con su propio archivo application.properties, donde se define un **puerto exclusivo** (por ejemplo, 8081 para Cursos, 8082 para Evaluaciones, hasta 8087 para Contenido), los **parámetros de conexión a una base de datos MySQL aislada** (spring.datasource.url, username, password) y la configuración de JPA e HikariCP (spring.jpa.hibernate.ddl-auto=update, hibernate.dialect, pool size). En el archivo pom.xml de cada servicio se declaran **dependencias comunes** como: spring-boot-starter-web para la capa web, spring-boot-starter-data-jpa y mysql-connector-java para persistencia, spring-boot-devtools para recarga en caliente, springdoc-openapi-starter-webmvc-ui para documentación Swagger disponible en /swagger-ui/index.html, y spring-boot-starter-test para pruebas unitarias e integración con JUnit 5, Mockito y MockMvc. Esta estructura modular, homogénea y bien documentada facilita la escalabilidad, promueve la separación de responsabilidades y cumple con los criterios exigidos por la rúbrica.

***5.- Desarrollo de controladores***

En esta tercera experiencia se incorporaron **cuatro nuevos controladores REST** que amplían de forma modular la funcionalidad de la plataforma: **InscripcionController**, que expone /api/inscripciones para listar inscripciones mediante GET y registrar nuevas con POST, validando previamente la existencia del usuario y curso antes de persistir la relación; **ContenidoController**, accesible en /api/contenidos, que gestiona el ciclo completo de los recursos académicos de cada curso (obtener con GET, crear con POST, actualizar con PUT y eliminar con DELETE), utilizando DTOs para estandarizar la estructura de los datos; **PagoController**, disponible en /api/pagos, responsable de procesar transacciones con POST (recibiendo monto y método de pago, validando que la inscripción esté pendiente y calculando descuentos o recargos si corresponde), además de consultar el estado de una transacción específica mediante GET /api/pagos/{inscripcionId}; y **SoporteController**, accesible en /api/soporte, permite a los usuarios registrar solicitudes de asistencia con POST, actualizar el estado del ticket con PUT (por ejemplo, de ABIERTO a EN PROGRESO o CERRADO), y eliminar tickets mediante DELETE. Esta funcionalidad está orientada a mejorar la experiencia del usuario ante incidencias o consultas sobre el sistema. Cada controlador delega la lógica a su servicio correspondiente a través de **inyección de dependencias**, lo que asegura una correcta separación de responsabilidades y facilita tanto la escalabilidad como las pruebas unitarias e integración.

***6.- Ejecución de pruebas***

Durante la ejecución de pruebas en los distintos microservicios de la plataforma **EduTech Innovators SPA**, se realizaron inicialmente **pruebas de carga de contexto**, utilizando clases ApplicationTests específicas para cada módulo (por ejemplo, CursosApplicationTests, EvaluacionesApplicationTests y UsuariosApplicationTests). Estas pruebas, configuradas con la anotación @SpringBootTest, permitieron verificar que el entorno de Spring Boot se inicializara correctamente, que todos los *beans* fueran detectados y que no existieran conflictos de configuración. Al ejecutar mvn test, cada uno de estos tests finalizó con un **BUILD SUCCESS**, lo cual confirmó la integridad estructural de los servicios y proporcionó una base sólida para avanzar hacia pruebas unitarias e integración más especializadas, sin riesgo de errores en el arranque ni en la resolución de dependencias.

***7.- Git y Github***

En el repositorio de GitHub (<https://github.com/moonnnluv/Desarrollo-Fullstack-I-002D>) inicializamos el control de versiones con git init, añadimos el origen remoto mediante git remote add origin https://github.com/moonnnluv/Desarrollo-Fullstack-I-002D.git, y para cada entrega seguimos el flujo habitual: git add ., git commit -m "Experiencia 3: pruebas e integración" y git push -u origin main. Cada microservicio cuenta con su propio README en el que documentamos la ruta base y los endpoints disponibles (por ejemplo, /api/usuarios (GET, POST, PUT, DELETE), /api/cursos (GET, POST, DELETE), /api/inscripciones (GET, POST), /api/contenidos (GET, POST, PUT, DELETE), /api/pagos (GET, POST), etc.), de modo que cualquier colaborador pueda probar las APIs directamente desde Postman o curl. Gracias a este esquema de commits claros y READMEs detallados, mantenemos un historial ordenado y facilitamos la incorporación de nuevos desarrolladores al proyecto.

***8.- Buenas practicas***

Implementamos diversas **buenas prácticas** para asegurar la calidad, mantenibilidad y escalabilidad de la plataforma. Cada microservicio respeta el principio de **separación de responsabilidades**, con capas bien definidas para entidades, repositorios, servicios y controladores. Utilizamos **DTOs** para establecer contratos claros de entrada y salida en las APIs, evitando exponer directamente las entidades JPA. Todas las rutas REST siguen una **convención uniforme** (/api/[recurso]) e implementan **idempotencia** cuando corresponde (por ejemplo, el uso de PUT para actualizaciones completas). Centralizamos la configuración de **CORS** mediante clases reutilizables dentro del paquete config, y aplicamos **inyección de dependencias** para desacoplar componentes y facilitar las pruebas. Además, adoptamos **Springdoc/OpenAPI** para generar automáticamente documentación Swagger de cada endpoint, lo que mejora la comunicación con el frontend y con posibles consumidores externos. Finalmente, mantenemos un flujo de trabajo Git limpio, con **commits atómicos** y **READMEs actualizados** por servicio que describen los endpoints y su uso esperado. Todo esto garantiza un código limpio, testeable y listo para futuras expansiones.

***9.- Documentación y descubrimiento de APIs***

Cada microservicio de EduTech Innovators SPA fue documentado utilizando Springdoc OpenAPI, generando de forma automática una interfaz Swagger accesible desde la ruta /swagger-ui/index.html. Esta documentación expone todos los endpoints REST disponibles, con sus métodos HTTP, parámetros requeridos, estructuras de entrada (DTOs) y respuestas esperadas, facilitando el consumo de las APIs tanto por desarrolladores frontend como por herramientas de prueba como Postman. La incorporación de esta capa de documentación no solo mejora la comprensión del sistema, sino que permite validar las rutas directamente desde el navegador y acelera la integración entre servicios o con plataformas externas.

***10.- Conclusión***

En conclusión, esta experiencia nos permitió consolidar la comprensión entre **pruebas unitarias**, enfocadas en validar la lógica interna de los servicios utilizando **JUnit y Mockito**, y **pruebas de integración**, dirigidas a verificar el funcionamiento completo de los endpoints REST mediante **MockMvc** y **Spring Boot Test**. La incorporación de **Swagger/OpenAPI** facilitó la documentación automática y el descubrimiento de cada ruta. Durante el desarrollo, enfrentamos desafíos como conflictos de puertos al ejecutar múltiples microservicios localmente y la sincronización del historial en Git tras fusiones, los cuales resolvimos ajustando con precisión los valores de server.port y aplicando **rebases interactivos** para mantener un historial limpio. Finalmente, reforzamos buenas prácticas clave: arquitectura modular con servicios independientes, uso coherente de convenciones REST (/api/[recurso]), implementación de DTOs, configuración centralizada de CORS y un enfoque de pruebas automatizadas que asegura calidad y facilita la evolución futura de la plataforma.