**Guía de Arquitectura de Microservicios para EduTech**

---

**1. Objetivo del Proyecto**

Migrar la aplicación EduTech de una arquitectura monolítica (todo en una sola aplicación y base de datos) a una arquitectura basada en microservicios (varios servicios independientes que se comunican entre sí). Esta migración permite mayor escalabilidad, facilidad de mantenimiento, despliegue independiente de módulos, y una estructura más limpia. Se usará Spring Boot, Eureka, OpenFeign, Swagger/OpenAPI, HATEOAS, JPA, pruebas con Postman y JUnit/Mockito. La base de datos es MySQL.

---

**2. Tecnologías Principales y Para Qué Sirve Cada Una**

| Tecnología | Propósito |
|---|---|
| **Spring Boot** | Framework que simplifica la creación de aplicaciones Java, especialmente servicios REST. Facilita la configuración. |
| **Eureka Server** | Sirve como "guía telefónica" para los microservicios: cada uno se registra y puede ser descubierto por otros. |
| **OpenFeign** | Permite que un microservicio pueda comunicarse con otro de forma sencilla, como si llamaras un método. |
| **Spring Data JPA** | Te evita escribir SQL directamente. Usas Java para guardar y leer datos desde la base de datos. |
| **MySQL** | Base de datos donde se almacenan los datos del sistema. |
| **Swagger / SpringDoc** | Herramientas que generan una página web para probar y ver toda la documentación de tus APIs. |
| **Spring HATEOAS** | Agrega enlaces dentro de las respuestas, para navegar entre recursos (como una API más intuitiva). |
| **Lombok** | Ahorra código en las clases: no necesitas escribir todos los getters/setters/ manualmente. |
| **JUnit / Mockito** | Permiten crear pruebas automáticas para asegurar que tu código funciona bien. |
| **Postman** | Herramienta visual para enviar peticiones y probar tus APIs manualmente. |

---

**3. Organización del Proyecto (Multimódulo Maven)**

Antes: todo el sistema estaba en un solo proyecto (monolítico). Ahora:

```
edutech-parent/                 # Proyecto padre Maven
├── eureka-server/             # Servidor Eureka donde se registran los
microservicios
```

```
├── ms-user/                    # Microservicio de usuarios
├── ms-course/                  # Microservicio de cursos
├── ms-payment/                 # Microservicio de pagos
├── ms-support/                 # Microservicio de soporte técnico
├── ms-quiz/                    # Microservicio de evaluaciones
├── ms-enrollment/             # Microservicio de inscripciones
├── ms-discount/               # Microservicio de cupones de descuento
├── ms-grade/                  # Microservicio de notas
└── common/                     # Librería compartida (DTOs, errores,
utilidades)
```

Cada microservicio tiene su propia estructura:

```
ms-xyz/
├── controller/      # Contiene los endpoints REST (acceso desde el
navegador/Postman)
├── service/         # Lógica del negocio (validaciones, reglas)
├── repository/      # Interfaces para interactuar con la base de datos
├── entity/          # Clases que representan las tablas de la base de datos
├── dto/             # Clases para enviar y recibir datos (evita exponer
entidades directamente)
└── application.yml  # Configuración del microservicio
```

---

**4. Eureka Server (Descubrimiento de servicios)**

En una arquitectura monolítica, todos los componentes están juntos y no se necesita "descubrir" a otros. En microservicios, los servicios deben saber cómo encontrarse entre sí. Ahí entra **Eureka**.

- Permite que cada microservicio se registre y diga: "Hola, estoy vivo en http://localhost:808X"
- Otros microservicios pueden buscarlo por su nombre.

---

**5. Registro de un Microservicio en Eureka**

Cada microservicio necesita:

1. Decirle a Eureka que existe ( `@EnableDiscoveryClient` )
2. Tener su propio `application.yml` con nombre único y puerto distinto
3. Definir la URL del servidor Eureka

Diferencia: antes todo corría en el mismo puerto. Ahora, cada microservicio tiene su propio puerto (por ejemplo, 8081, 8082...).

---

**6. OpenFeign (Llamadas entre microservicios)**

En el sistema monolítico, una clase llamaba directamente a otra. En microservicios, están separados y deben comunicarse por red (HTTP). **Feign** lo hace más fácil.

Ejemplo: el microservicio de cursos quiere consultar un usuario → en lugar de escribir `HttpClient`, usa una interfaz `UserClient` anotada con `@FeignClient`.

---

**7. Swagger / SpringDoc (Documentación de API)**

Antes, no había una forma clara de ver qué endpoints existían. Ahora, con Swagger:

- Se genera una página web con todos los endpoints.
- Puedes probarlos desde esa interfaz.
- Se usa para verificar cómo se ve una respuesta, qué parámetros necesita, etc.

URL típica: `http://localhost:8081/swagger-ui.html`

---

**8. Spring HATEOAS (Navegación en la API)**

Sirve para hacer más intuitiva la navegación entre recursos REST. En vez de solo devolver datos, también puedes devolver enlaces útiles. Ejemplo:

```
{
  "nombre": "Curso Java",
  "_links": {
    "self": { "href": "/api/courses/1" },
    "instructor": { "href": "/api/users/7" }
  }
}
```

---

**9. JUnit y Mockito (Pruebas Unitarias)**

Antes: pruebas manuales con Postman. Ahora: también se prueban partes pequeñas automáticamente.

- **JUnit** para verificar que un método funciona.
- **Mockito** para simular repositorios o dependencias.

---

**10. Postman (Pruebas de API)**

Aunque se use JUnit, Postman sigue siendo útil para:

- Ver cómo responde tu API realmente.
- Hacer pruebas rápidas.
- Probar errores, validaciones, etc.

Ahora, como hay varios microservicios, lo ideal es tener una carpeta de pruebas Postman para cada uno.

---

**11. Flujo de Desarrollo Recomendado (explicado paso a paso)**

1. **Crear microservicio con Spring Initializr**
2. Selecciona dependencias: Spring Web, JPA, MySQL Driver, Lombok, Validation, Eureka Client
3. Genera estructura base del proyecto
4. **Registrar en Eureka**
5. Agrega `@EnableDiscoveryClient` en la clase principal
6. Configura `application.yml` con nombre, puerto y dirección de Eureka
7. **Crear entidad, DTO, repositorio, servicio, controlador**
8. Entidad = tabla de base de datos (por ejemplo, `User`)
9. DTO = clase simplificada para enviar/recibir datos
10. Repositorio = interfaz que extiende `JpaRepository`
11. Servicio = clase que hace la lógica
12. Controlador = donde defines los endpoints (`@RestController`)
13. **Agregar Swagger y pruebas unitarias**
14. Swagger para ver los endpoints
15. JUnit para probar la lógica automáticamente
16. **Probar con Postman**
17. Verifica que los endpoints funcionan correctamente
18. **Repetir para cada microservicio**
19. Aplica lo anterior para cursos, pagos, inscripciones, etc.
20. **Agregar Feign para comunicación entre microservicios**
21. Cuando un servicio necesita info de otro, crea un cliente Feign
22. **Integrar HATEOAS**
23. Agrega enlaces útiles a las respuestas para facilitar navegación

---

**12. Consideraciones Adicionales**

- Usa puertos distintos por microservicio (ej. 8081, 8082, 8083...)
- Opcionalmente, cada microservicio puede tener su propia base de datos
- Usa el módulo `common` para:
- Compartir DTOs entre servicios
- Definir errores comunes
- Reutilizar utilidades

---

**Fin del documento**