



**UNIVERSIDAD DE LAS FUERZAS ARMADAS
ESPE**

Sistema de Matrículas

Aplicaciones Distribuidas NRC 14930

Presentado por: De Veintemilla Luca, Hernandez Dylan, Reyes Juan, Vargas Kevin

Docente: Ing. Dario Morales

Fecha

21 de Julio del 2024

Tabla de Contenidos

Objetivo General.....	2
Objetivos Específicos.....	2
Introducción.....	2
Desarrollo.....	3
Proceso.....	3
Decisiones de diseño.....	3
Problemas encontrados.....	3
Diagramas de arquitectura.....	3
Flujo de datos.....	3
Pruebas.....	3
Pruebas JUnit.....	3
Pruebas Jasmine.....	3
Pruebas E2E.....	3
Conclusiones.....	3
Recomendaciones.....	3
Referencias Bibliográficas.....	4

Tabla de Gráficos

Arquitectura de Microservicios.....	13
Flujo de la solicitud.....	13
Tests de la capa de servicio del curso.....	19
Test de Integración en Curso Controller.....	29
Tests de Usuarios en capa de servicio.....	35
Captura de todas las pruebas completadas.....	45
Configuración de Entorno.....	46
Pruebas E2E.....	48

Objetivo General

Desarrollar un sistema de matrículas eficiente y escalable que permita la gestión de usuarios y cursos mediante una arquitectura de microservicios.

Objetivos Específicos

- Implementar un microservicio para la gestión de usuarios que permita crear, actualizar, eliminar y consultar información de usuarios.
- Desarrollar un microservicio para la gestión de cursos que permita la creación, actualización, eliminación y consulta de cursos, así como la asignación de usuarios a cursos.
- Garantizar la validación de datos en ambos microservicios utilizando anotaciones de Jakarta Validation para asegurar la integridad de la información.
- Establecer una comunicación eficiente entre los microservicios de usuarios y cursos utilizando Feign.
- Realizar pruebas unitarias y de integración para asegurar el correcto funcionamiento del sistema.

Introducción

La arquitectura de microservicios ha emergido como una solución efectiva para desarrollar sistemas distribuidos escalables y mantenibles, superando las limitaciones de las aplicaciones monolíticas tradicionales (Diego Fernando, 2020). Este enfoque divide las aplicaciones en servicios pequeños y autónomos, facilitando la evolución y el despliegue independiente de componentes (Alan González et al., 2021). Los equipos de desarrollo de microservicios tienden a ser pequeños y multifuncionales, adoptando prácticas ágiles y una cultura DevOps (Alan González et al., 2021). Sin embargo, la implementación de microservicios presenta desafíos, especialmente en el manejo de transacciones distribuidas. Para abordar esto, se han propuesto arquitecturas basadas en eventos y computación en la nube, que permiten la

compensación y reversión de transacciones distribuidas sin afectar el rendimiento de los microservicios (B. Rodríguez & D. Cedeño, 2023). Estas soluciones ofrecen alternativas a los arquitectos y desarrolladores para gestionar eficazmente sistemas complejos y altamente transaccionales.

Desarrollo

Proceso

El programa descrito involucra múltiples componentes trabajando juntos para permitir la creación, actualización, eliminación y consulta de usuarios y cursos, así como la asignación de usuarios a cursos. A continuación, se describe el flujo de operaciones paso a paso:

1. Inicio de una solicitud HTTP

Un cliente (puede ser un navegador web, Postman, o cualquier otro cliente HTTP) envía una solicitud a la aplicación. Esta solicitud puede ser para listar, crear, actualizar, eliminar usuarios o cursos, o para asignar usuarios a cursos.

2. Controladores

La solicitud llega a un controlador apropiado en el backend. Hay dos controladores principales:

UsuarioController y CursoController.

UsuarioController maneja solicitudes relacionadas con usuarios (/api/usuarios).

CursoController maneja solicitudes relacionadas con cursos (/api/cursos).

3. Validación de datos

Si la solicitud involucra la creación o actualización de datos (usuarios o cursos), los datos enviados en el cuerpo de la solicitud (RequestBody) son validados usando anotaciones de Jakarta Validation (@Valid).

Si hay errores de validación, se devuelve una respuesta con estado HTTP 400 (Bad Request) junto con detalles de los errores.

4. Servicios

Si los datos son válidos, el controlador llama a un método apropiado en la capa de servicio (UsuarioService o CursoService) para procesar la solicitud.

Los servicios contienen la lógica de negocio y hacen llamadas a los repositorios para interactuar con la base de datos.

5. Repositorios

Los repositorios (UsuarioRepository y CursoRepository) utilizan Spring Data JPA para interactuar con la base de datos. Estos repositorios abstraen las operaciones CRUD (crear, leer, actualizar, eliminar) y consultas personalizadas.

6. Base de datos

Las operaciones solicitadas se ejecutan en la base de datos. La base de datos puede ser cualquier sistema de gestión de bases de datos compatible con JPA (MySQL, PostgreSQL).

7. Respuesta HTTP

Una vez procesada la solicitud, la capa de servicio devuelve el resultado al controlador, que a su vez construye una respuesta HTTP. Esta respuesta puede incluir datos (como el detalle de un usuario o curso) y un código de estado HTTP apropiado (por ejemplo, 200 OK, 201 Created, 204 No Content).

8. Cliente HTTP

El cliente recibe la respuesta HTTP. Dependiendo del resultado, el cliente puede mostrar los datos, un mensaje de éxito, o un mensaje de error.

9. Comunicación entre microservicios

La comunicación entre microservicios en este contexto se realiza mediante Feign, un cliente declarativo de servicios web que facilita la creación de clientes HTTP. En el caso específico de asignar usuarios a cursos, el proceso se desarrolla de la siguiente manera:

- Definición del Cliente Feign (UsuarioClientRest)

Se define una interfaz en el microservicio de cursos que declara los métodos para interactuar con el microservicio de usuarios. Esta interfaz utiliza anotaciones de Spring Cloud OpenFeign para mapear estos métodos a peticiones HTTP.

```
@FeignClient(name = "msvc-usuarios", url = "localhost:8001")
public interface UsuarioClientRest {
    @GetMapping("/api/usuarios/detalles/{id}")
    Usuario detalle(@PathVariable Long id);

    @PostMapping("/api/usuarios/crear")
    Usuario crear(@RequestBody Usuario usuario);
}
```

- Uso del Cliente Feign en CursoServiceImpl

Cuando se necesita asignar un usuario a un curso, CursoServiceImpl invoca el método detalle o crear del UsuarioClientRest. Esto inicia una solicitud HTTP al microservicio de usuarios para verificar la existencia del usuario o crear uno nuevo.

- Respuesta y Manejo de Errores

La solicitud HTTP puede tener varios resultados. Si el usuario existe o se crea correctamente, el microservicio de usuarios devuelve los detalles del usuario, que el microservicio de cursos puede utilizar para completar la asignación. Si hay un error (por ejemplo, el usuario no existe y no se puede crear), se maneja adecuadamente, posiblemente devolviendo un error al cliente que inició la solicitud.

Este proceso permite una integración estrecha entre microservicios, facilitando operaciones complejas que involucran múltiples recursos distribuidos, manteniendo al mismo tiempo la separación de responsabilidades y la independencia de los servicios.

Ejemplo de proceso

1. Inicio de una solicitud HTTP

El cliente envía una solicitud POST a `/api/usuarios/crear` con un cuerpo JSON que contiene los datos del nuevo usuario.

```
{  
  "nombre": "Nuevo Usuario",  
  "email": "nuevo@example.com",  
  "password": "contraseña"  
}
```

2. Controladores

La solicitud es capturada por `UsuarioController` en el método `crear`.

```
@PostMapping("/crear")  
public ResponseEntity<?> crear(@Valid @RequestBody Usuario usuario, BindingResult  
result){  
    if(result.hasErrors()){  
        return validar(result);  
    }  
    Usuario usuarioGuardado = service.guardar(usuario);  
    return ResponseEntity.status(HttpStatus.CREATED).body(usuarioGuardado);  
}
```

3. Validación de datos

La anotación `@Valid` activa la validación de Jakarta Validation en el objeto `Usuario` recibido. Si hay errores, se invoca el método `validar` para construir una respuesta con los detalles de los errores.

```
private static ResponseEntity<Map<String,String>> validar(BindingResult result){
    Map <String,String> errores = new HashMap<>();
    result.getFieldErrors().forEach(err -> {
        errores.put(err.getField(), "El campo " + err.getField() + " " +
err.getDefaultMessage());
    });
    return ResponseEntity.badRequest().body(errores);
}
```

4. Servicios

Si no hay errores de validación, se llama al método guardar del servicio UsuarioService.

```
@Override
@Transactional
public Usuario guardar(Usuario usuario) {
    return repository.save(usuario);
}
```

5. Repositorios

El servicio utiliza UsuarioRepository, que extiende CrudRepository, para interactuar con la base de datos y guardar el nuevo usuario.

```
public interface UsuarioRepository extends CrudRepository<Usuario, Long>{
}
```

6. Base de datos

La operación de guardar se traduce en una inserción en la base de datos de la tabla correspondiente a la entidad Usuario.

7. Respuesta HTTP

Una vez guardado el usuario, el servicio devuelve el usuario guardado al controlador, que construye una respuesta HTTP con el estado 201 Created y el usuario guardado en el cuerpo.

8. Cliente HTTP

El cliente recibe la respuesta HTTP con el estado 201 Created y los datos del usuario recién creado, completando así el flujo de la solicitud.

Decisiones de diseño

El sistema descrito es un conjunto de microservicios para la gestión de usuarios y cursos, utilizando Java con Spring Boot para el backend, y comunicación entre microservicios mediante Feign.

Arquitectura de Microservicios

- Separación de Responsabilidades: Se divide la lógica de negocio en dos microservicios principales: msvc-usuarios para la gestión de usuarios y msvc-cursos para la gestión de cursos. Esto permite una escalabilidad y mantenimiento más eficientes.
- Comunicación entre Microservicios: Utiliza Feign para la comunicación entre msvc-cursos y msvc-usuarios, permitiendo una integración declarativa y simplificada entre servicios.

Base de Datos

- Elección de la Base de Datos: Se asume el uso de una base de datos relacional compatible con JPA (MySQL, PostgreSQL) debido a la estructura de datos y las relaciones entre entidades.
- JPA para la Persistencia: Se utiliza Spring Data JPA para abstraer las operaciones CRUD y simplificar la interacción con la base de datos.

Modelo de Datos

- Entidades: Se definen entidades como Usuario y Curso, cada una mapeada a su tabla correspondiente en la base de datos.

- Validaciones: Se utilizan anotaciones de Jakarta Validation en las entidades para asegurar la integridad de los datos (por ejemplo, `@NotEmpty`, `@Email`).

API REST

- Endpoints: Se definen endpoints específicos para las operaciones CRUD y adicionales (listar, detalles, crear, actualizar, eliminar) en ambos microservicios.
- Validación de Solicitudes: Se valida el cuerpo de las solicitudes entrantes usando `@Valid` y `BindingResult` para manejar errores de validación de forma elegante.
- Respuestas HTTP: Se utilizan códigos de estado HTTP apropiados para indicar el resultado de las operaciones (por ejemplo, 200 OK, 201 Created, 400 Bad Request).
- Seguridad (Opcional)

Pruebas

- Pruebas Unitarias y de Integración: Se utilizan JUnit para las pruebas, asegurando la calidad del software y que los componentes funcionen como se espera tanto de forma aislada como integrada.

Estas decisiones de diseño proporcionan una base sólida para un sistema de microservicios robusto, escalable y mantenible, con prácticas y herramientas modernas para el desarrollo, pruebas, seguridad y despliegue.

Problemas encontrados

1. Validación de Datos en el Backend

La validación de datos no se realizaba correctamente en las entidades, permitiendo la creación de usuarios y cursos con datos inválidos. Se añadieron anotaciones de Jakarta Validation (`@NotEmpty`, `@Email`, etc.) en las clases de entidad para asegurar que los datos recibidos cumplan con los requisitos antes de ser procesados. Esto se complementó con la verificación de `BindingResult` en los controladores para manejar errores de validación.

```

@NotEmpty
private String nombre;

@NotEmpty
@email
@Column(unique = true)
private String email;

```

Y en el controlador del usuario se verifica con BindingResult para los errores.

```

private static ResponseEntity<Map<String,String>>
validar(BindingResult result){
    Map <String,String> errores = new HashMap<>();
    result.getFieldErrors().forEach(err -> {
        errores.put(err.getField(), "El campo " + err.getField() +
" " + err.getDefaultMessage());
    });
    return ResponseEntity.badRequest().body(errores);
}

```

2. Manejo de Excepciones en Llamadas Feign

Las llamadas entre microservicios con Feign no manejaban adecuadamente las excepciones, resultando en fallos silenciosos o respuestas inesperadas cuando el microservicio de usuarios no estaba disponible o no se encontraba el recurso solicitado. Se implementó un manejo de excepciones específico para capturar FeignException y devolver respuestas HTTP adecuadas. Esto incluyó la verificación del estado de la respuesta y la inclusión de mensajes de error claros para el cliente.

```

@PutMapping("/asignar-usuario/{idcurso}")
public ResponseEntity<?> asignarUsuario(@RequestBody Usuario

```

```

usuario, @PathVariable Long idcurso){
    Optional<Usuario> o;
    try {
        o = service.agregarUsuario(usuario, idcurso);
    } catch (FeignException e){
        return ResponseEntity.status(HttpStatus.NOT_FOUND).
            body(Collections.singletonMap("mensaje", "Error:"
+ e.getMessage()));
    }
    if (o.isPresent()){
        return
ResponseEntity.status(HttpStatus.CREATED).body(o.get());
    }
    return ResponseEntity.notFound().build();
}

```

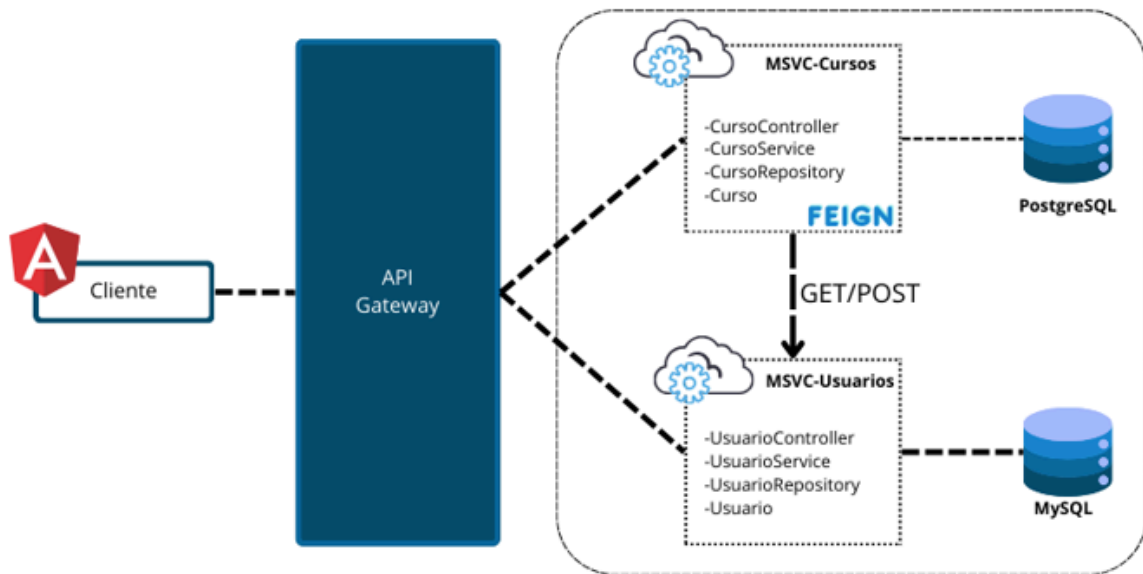
3. Pruebas Unitarias y de Integración

Las pruebas iniciales no cubrían todos los casos de uso y flujos de datos posibles, lo que podría llevar a errores no detectados en la lógica de negocio. Entonces se amplió la cobertura de pruebas, incluyendo pruebas unitarias y de integración para los controladores, servicios y la comunicación entre microservicios. Se utilizaron `@MockBean` para simular dependencias y `MockMvc` para probar los endpoints de la API.

Diagramas de arquitectura

Figura 1

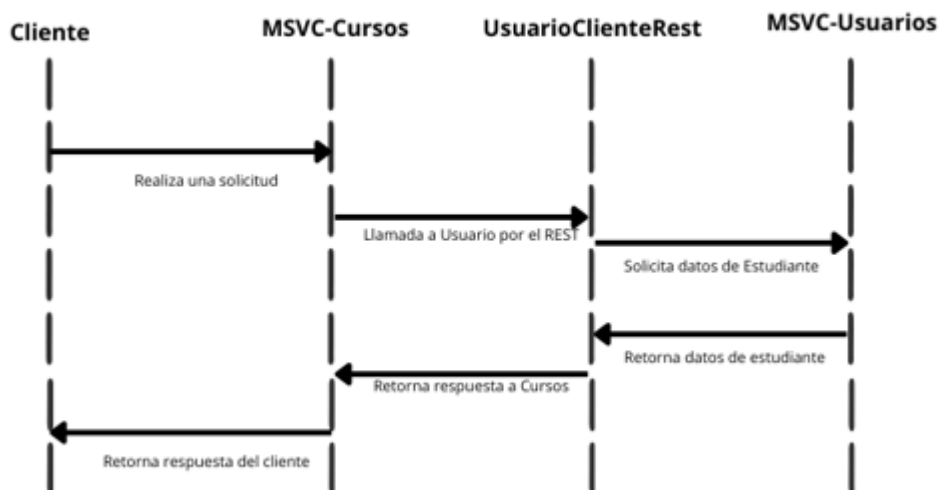
Arquitectura de Microservicios



Flujo de datos

Figura 2

Flujo de la solicitud



Pruebas

Pruebas JUnit

Para la instalación de JUnit deberemos añadir dependencias, deberemos asegurarnos de tener las dependencias necesarias en el archivo pom.xml para JUnit, Spring Boot Test.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
  <scope>test</scope>
</dependency>
```

Estas dos dependencias nos servirán para hacer las pruebas unitarias. En nuestro caso haremos para la capa de servicios y la del controlador.

Pruebas para Capa de Servicio

Se utiliza `@MockBean` para simular las dependencias como el repositorio o clientes Feign. Para mockear las dependencias utilizamos `@MockBean` para crear un mock de `CursoRepository` y `UsuarioClientRest`. Usamos `@Autowired` para inyectar la instancia de `CursoService` en tu clase de prueba.

Finalmente escribimos los métodos de prueba usando `@Test` para definir métodos de prueba. Dentro de estos métodos, se configura expectativas de los mocks y se llama a los métodos del servicio para verificar el comportamiento.

Test de PorID

```

@Test
public void porIdTest() {
    // Configura el mock para devolver un valor específico cuando
    // se llame al método findById
    Curso curso = new Curso();
    curso.setId(1L);

    when(cursoRepository.findById(1L)).thenReturn(Optional.of(curso));

    // Llama al método porId del servicio
    Optional<Curso> resultado = cursoService.porId(1L);

    // Verifica que el resultado es el esperado
    assertTrue(resultado.isPresent(), "El curso no fue
    encontrado");
    assertEquals(1L, resultado.get().getId(), "El ID del curso no
    coincide");

    // Verifica que se llamó al método findById del repositorio
    verify(cursoRepository).findById(1L);
}

```

El test verifica que el resultado de llamar al método `porId` con el valor `1L` sea un `Optional` que está presente. Esto se comprueba con `assertTrue(resultado.isPresent(), "El curso no fue encontrado");`. Si el `Optional` está vacío, el test fallará indicando que el curso no fue encontrado.

Además, el test comprueba que el ID del `Curso` obtenido es igual a `1L`, usando `assertEquals(1L, resultado.get().getId(), "El ID del curso no coincide");`. Si el ID no coincide, el test fallará indicando que el ID del curso no es el esperado.

Finalmente, se verifica que se haya llamado exactamente una vez al método `findById` del `cursoRepository` con el argumento `1L`, mediante `verify(cursoRepository).findById(1L);`. Si el método no se llamó de esta manera, el test fallará indicando que la interacción con el mock no fue la esperada.

Si todas estas condiciones se cumplen, el test pasará. En caso contrario, el test fallará indicando cuál de las condiciones no se cumplió según los mensajes de error proporcionados.

Test Listar

```
@Test
public void listarTest() {
    // Configurar el mock para devolver una lista de cursos
    List<Curso> cursos = new ArrayList<>();
    cursos.add(new Curso());
    when(cursoRepository.findAll()).thenReturn(cursos);

    // Llamar al método listar
    List<Curso> resultado = cursoService.listar();

    // Verificar el resultado
    assertFalse(resultado.isEmpty(), "La lista de cursos está vacía");
    verify(cursoRepository).findAll();
}
```

Se crea una lista de `Curso` con un elemento y se configura el mock para que, cuando se llame al método `findAll()`, retorne esta lista. Esto simula la situación en la que la base de datos contiene al menos un curso.

El método `listar`, bajo prueba, debería invocar internamente el método `findAll()` del repositorio `cursoRepository` para obtener todos los cursos disponibles y retornarlos.

Se asegura que la lista de cursos retornada no esté vacía(`assertFalse(resultado.isEmpty())`). Esto verifica que el servicio está efectivamente recuperando los cursos como se espera, basado en el comportamiento simulado del repositorio.

Con `verify(cursoRepository).findAll()`, se comprueba que el método `findAll()` del repositorio `cursoRepository` fue invocado. Esto asegura que el servicio `cursoService` está interactuando con el repositorio como se espera.

Test Guardar

```
@Test
public void guardarTest() {
    // Configurar el mock para guardar y devolver el curso
    Curso curso = new Curso();
    curso.setNombre("Curso de prueba");

    when(cursoRepository.save(any(Curso.class))).thenReturn(curso);

    // Llamar al método guardar
    Curso resultado = cursoService.guardar(new Curso());

    // Verificar el resultado
    assertNotNull(resultado, "El curso guardado es nulo");
    assertEquals("Curso de prueba", resultado.getNombre(), "El
nombre del curso no coincide");
}
```

Se prepara el mock para que, al llamar al método `save` con cualquier instancia de `Curso`, retorne un objeto `Curso` específico. Este objeto tiene el nombre "Curso de prueba", simulando que el curso se ha guardado correctamente en la base de datos.

Se invoca el método `guardar` del servicio `cursoService`, pasando una nueva instancia de `Curso`. Este es el método que se está probando.

Se verifica que el objeto Curso retornado por el método guardar no es nulo y que el nombre del curso coincide con "Curso de prueba". Esto asegura que el curso se ha guardado correctamente y que los datos retornados son los esperados.

Test Eliminar

```
@Test
public void eliminarTest() {
    // Configurar el mock para simular la eliminación
    doNothing().when(cursoRepository).deleteById(1L);

    // Llamar al método eliminar
    cursoService.eliminar(1L);

    // Verificar que se llamó al método deleteById del
    repositorio
    verify(cursoRepository).deleteById(1L);
}
```

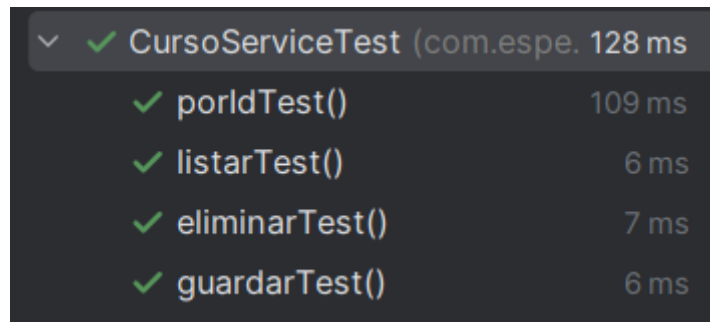
Se prepara el mock para que, al llamar al método deleteById con el ID 1L, no realice ninguna acción. Esto simula la eliminación del curso en la base de datos.

Se invoca el método eliminar del servicio cursoService, pasando el ID 1L del curso a eliminar. Este es el método que se está probando.

Se verifica que el método deleteById del repositorio cursoRepository fue llamado con el ID 1L. Esto asegura que el servicio cursoService intentó eliminar el curso correctamente.

Figura 3

Tests de la capa de servicio del curso

A screenshot of a test runner interface showing the results for 'CursoServiceTest'. The test suite is marked as passed with a green checkmark and took 128 ms. It contains four sub-tests, all of which also passed with green checkmarks: 'porIdTest()' (109 ms), 'listarTest()' (6 ms), 'eliminarTest()' (7 ms), and 'guardarTest()' (6 ms).

✓ CursoServiceTest	(com.espe. 128 ms)
✓ porIdTest()	109 ms
✓ listarTest()	6 ms
✓ eliminarTest()	7 ms
✓ guardarTest()	6 ms

Pruebas de integración Controladores

Las pruebas de integración para controladores verificarán la integración de varios componentes y cómo manejan las solicitudes HTTP.

Usaremos `@SpringBootTest` y `@AutoConfigureMockMvc` para configurar el entorno de prueba para simular el comportamiento de la aplicación en ejecución.

El inyectar `MockMvc` permite enviar solicitudes HTTP y verificar las respuestas sin necesidad de un servidor en ejecución. Cada método de prueba enviará una solicitud HTTP al endpoint correspondiente y verificará la respuesta.

Tests Detalle

```
@Test
public void detalleTest() throws Exception {
    Curso curso = new Curso();
    curso.setId(1L);
    curso.setNombre("Curso de Spring Boot");

    given(cursoService.porId(1L)).willReturn(Optional.of(curso));

    mockMvc.perform(get("/api/cursos/detalles/{id}", 1L)
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk());
}
```

```

        .andExpect(jsonPath("$.nombre").value("Curso de Spring
Boot"));
    }

    @Test
    public void detalleCursoNoExistenteTest() throws Exception {
        Long cursoIdInexistente = 2L;

        given(cursoService.findById(cursoIdInexistente)).willReturn(Optional
            .empty());

        mockMvc.perform(get("/api/cursos/detalles/{id}",
            cursoIdInexistente)
            .contentType(MediaType.APPLICATION_JSON)
            .andExpect(status().isNotFound()));
    }

    @Test
    public void detalleConIdInvalidoTest() throws Exception {
        String idInvalido = "abc";

        mockMvc.perform(get("/api/cursos/detalles/{id}", idInvalido)
            .contentType(MediaType.APPLICATION_JSON)
            .andExpect(status().isBadRequest()));
    }
}

```

detalleTest:

El objetivo de este test es verificar que el endpoint devuelve correctamente los detalles de un curso existente.

1. Se crea un objeto Curso y se le asignan un ID y un nombre.
2. Se configura el mock cursoService.findById para que devuelva un Optional conteniendo el objeto Curso cuando se le pasa el ID correspondiente.
3. Se realiza una petición GET al endpoint con el ID del curso.
4. Se verifica que la respuesta tenga un estado HTTP 200 (OK) y que el cuerpo de la respuesta contenga el nombre del curso.

Verifica que el endpoint devuelve correctamente los detalles de un curso cuando este existe.

detalleCursoNoExistenteTest:

El objetivo es verificar que el endpoint devuelve un estado HTTP 404 (Not Found) cuando el curso solicitado no existe.

1. Se configura el mock `cursoService.findById` para que devuelva un `Optional` vacío cuando se le pasa un ID que no corresponde a ningún curso existente.
2. Se realiza una petición GET al endpoint con un ID inexistente.
3. Se verifica que la respuesta tenga un estado HTTP 404 (Not Found).

Verifica que el endpoint maneja correctamente la situación cuando se solicita un curso que no existe.

detalleConIdInvalidoTest:

Verificar que el endpoint devuelve un estado HTTP 400 (Bad Request) cuando el ID proporcionado no es válido.

1. Se realiza una petición GET al endpoint con un ID que no es un número válido (en este caso, una cadena de texto).
2. Se verifica que la respuesta tenga un estado HTTP 400 (Bad Request).

Verifica que el endpoint valida correctamente el formato del ID del curso y maneja adecuadamente los casos en que el ID no es válido.

Test Crear

```
@Test
public void crearTest() throws Exception {
```

```

        Curso curso = new Curso();
        curso.setId(1L);
        curso.setNombre("Curso de Spring Boot");

given(cursoService.guardar(any(Curso.class))).willAnswer(invocation
on -> invocation.getArgument(0));

        mockMvc.perform(post("/api/cursos/crear")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content("{\"nombre\":\"Curso de Spring
Boot\"}"))
                .andExpect(status().isCreated())
                .andExpect(jsonPath("$.nombre").value("Curso de Spring
Boot"));
    }

    @Test
    public void crearCursoConDatosIncompletosTest() throws Exception
    {
        mockMvc.perform(post("/api/cursos/crear")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content("{}")) // Datos incompletos
                .andExpect(status().isBadRequest()); // Esperamos un
estado HTTP 400 Bad Request
    }

    @Test
    public void crearCursoConDatosInvalidosTest() throws Exception {
        mockMvc.perform(post("/api/cursos/crear")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content("{\"nombre\":\"\"}")) // Nombre
inválido (vacío)
                .andExpect(status().isBadRequest()); // Esperamos un
estado HTTP 400 Bad Request
    }

```

crearTest

Verificar que el endpoint crea un curso correctamente cuando se proporcionan datos válidos.

1. Se crea un objeto Curso con un ID y nombre específicos.
2. Se configura el mock cursoService.guardar para que devuelva el mismo objeto Curso que recibe.
3. Se realiza una petición POST al endpoint con un JSON que representa un curso válido.
4. Se verifica que la respuesta tenga un estado HTTP 201 (Created) y que el cuerpo de la respuesta contenga el nombre del curso, indicando que fue creado exitosamente.

crearCursoConDatosIncompletosTest

Verificar que el endpoint devuelve un error cuando se intenta crear un curso sin proporcionar todos los datos necesarios.

1. Se realiza una petición POST al endpoint con un cuerpo vacío.
2. Se verifica que la respuesta tenga un estado HTTP 400 (Bad Request), indicando que los datos proporcionados son insuficientes o incorrectos para crear un curso.

crearCursoConDatosInvalidosTest

Verificar que el endpoint devuelve un error cuando se intenta crear un curso con datos inválidos (en este caso, un nombre vacío).

1. Se realiza una petición POST al endpoint con un JSON que contiene un nombre de curso vacío.
2. Se verifica que la respuesta tenga un estado HTTP 400 (Bad Request), indicando que los datos proporcionados no son válidos para la creación de un curso

Test actualizar

```

@Test
public void actualizarTest() throws Exception {
    Curso curso = new Curso();
    curso.setId(1L);
    curso.setNombre("Curso de Spring Boot Actualizado");

    given(cursoService.findById(1L)).willReturn(Optional.of(curso));

    given(cursoService.guardar(any(Curso.class))).willReturn(invocation -> invocation.getArgument(0));

    mockMvc.perform(put("/api/cursos/actualizar/{id}", 1L)
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\"nombre\":\"Curso de Spring Boot Actualizado\"}"))
        .andExpect(status().isCreated())
        .andExpect(jsonPath("$.nombre").value("Curso de Spring Boot Actualizado"));
}

@Test
public void actualizarCursoNoExistenteTest() throws Exception {
    Long cursoIdInexistente = 100L; // Un ID que se asume no existe en la base de datos

    given(cursoService.findById(cursoIdInexistente)).willReturn(Optional.empty());

    mockMvc.perform(put("/api/cursos/actualizar/{id}", cursoIdInexistente)
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\"nombre\":\"Curso Inexistente\"}"))
        .andExpect(status().isNotFound());
}

```

actualizarTest

Asegurar que el endpoint actualiza correctamente un curso existente y devuelve el estado HTTP 201 (Created), junto con los datos actualizados del curso.

1. Se crea un objeto Curso con un ID y un nombre específicos ("Curso de Spring Boot Actualizado").
2. Se configura el mock `cursoService.findById` para que devuelva un `Optional` conteniendo el objeto Curso cuando se le pasa el ID 1L, simulando que el curso existe en la base de datos.
3. Se configura el mock `cursoService.guardar` para que devuelva el mismo objeto Curso que recibe, simulando el proceso de actualización.
4. Se realiza una petición PUT al endpoint `/api/cursos/actualizar/{id}` con el ID del curso y el contenido JSON que representa los datos actualizados del curso.
5. Se verifica que la respuesta tenga un estado HTTP 201 (Created) y que el cuerpo de la respuesta contenga el nombre actualizado del curso.

Se verifica que el endpoint de actualización funciona correctamente cuando se actualiza un curso existente.

actualizarCursoNoExistenteTest

Comprobar que el endpoint devuelve un estado HTTP 404 (Not Found) cuando se intenta actualizar un curso que no existe en la base de datos.

1. Se configura el mock `cursoService.findById` para que devuelva un `Optional.empty()`, simulando que el curso con ID 100L no existe.
2. Se realiza una petición PUT al endpoint `/api/cursos/actualizar/{id}` con un ID inexistente (100L) y un cuerpo JSON que intenta actualizar un curso.
3. Se verifica que la respuesta tenga un estado HTTP 404 (Not Found), indicando que el curso no se pudo encontrar y, por lo tanto, no se pudo actualizar.

Se verifica que el endpoint de actualización maneja correctamente los casos en los que se intenta actualizar un curso que no existe, devolviendo un estado adecuado para indicar que el recurso no se encontró.

Test Eliminar

```
@Test
public void eliminarTest() throws Exception {
    Curso curso = new Curso();
    curso.setId(1L);

    given(cursoService.porId(1L)).willReturn(Optional.of(curso));
    doNothing().when(cursoService).eliminar(1L);

    mockMvc.perform(delete("/api/cursos/eliminar/{id}", 1L))
        .andExpect(status().isNoContent());
}
```

Este test asegura que el endpoint de eliminación de cursos maneje correctamente la eliminación de cursos existentes, devolviendo el código de estado adecuado.

Test Asignar Usuario

```
@Test
public void asignarUsuarioTest() throws Exception {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    usuario.setNombre("Usuario Test");

    given(cursoService.agregarUsuario(any(Usuario.class),
    eq(1L))).willReturn(Optional.of(usuario));

    mockMvc.perform(put("/api/cursos/asignar-usuario/{idcurso}",
    1L)
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\"id\":1,\"nombre\":\"Usuario Test\"}"))
        .andExpect(status().isCreated());
}
```

```

        .andExpect(jsonPath("$.nombre").value("Usuario
Test"));
    }

    @Test
    public void asignarUsuarioACursoNoExistenteTest() throws
    Exception {
        Usuario usuario = new Usuario();
        usuario.setId(1L);
        usuario.setNombre("Usuario Test");

        Long cursoIdInexistente = 999L; // ID de curso que se asume no
    existe

        given(cursoService.agregarUsuario(any(Usuario.class),
    eq(cursoIdInexistente))).willReturn(Optional.empty());

        mockMvc.perform(put("/api/cursos/asignar-usuario/{idcurso}",
    cursoIdInexistente)
                .contentType(MediaType.APPLICATION_JSON)
                .content("{\"id\":1,\"nombre\":\"Usuario
Test\"}"))
                .andExpect(status().isNotFound());
    }

    @Test
    public void asignarUsuarioConDatosInvalidosTest() throws
    Exception {
        mockMvc.perform(put("/api/cursos/asignar-usuario/{idcurso}",
    "abc") // ID de curso inválido
                .contentType(MediaType.APPLICATION_JSON)
                .content("{\"id\":\"xyz\",\"nombre\":\"Usuario
Test\"}")) // ID de usuario inválido
                .andExpect(status().isBadRequest());
    }

```

asignarUsuarioTest

Verificar que el endpoint puede asignar correctamente un usuario a un curso existente.

1. Se crea un objeto Usuario con un ID y nombre específicos.
2. Se configura el mock cursoService.agregarUsuario para que devuelva un Optional conteniendo el objeto Usuario cuando se le pasa cualquier objeto Usuario y el ID del curso 1L.
3. Se realiza una petición PUT al endpoint con el ID del curso 1L y un cuerpo JSON que representa al usuario a asignar.
4. Se verifica que la respuesta tenga un estado HTTP 201 (Created) y que el cuerpo de la respuesta contenga el nombre del usuario asignado, indicando que la asignación fue exitosa.

Verificar que el endpoint asigna correctamente un usuario a un curso existente y devuelve los datos del usuario asignado.

asignarUsuarioACursoNoExistenteTest

Comprobar que el endpoint devuelve un estado HTTP 404 (Not Found) cuando se intenta asignar un usuario a un curso que no existe.

1. Se configura el mock cursoService.agregarUsuario para que devuelva un Optional.empty(), simulando que el curso con ID 999L no existe.
2. Se realiza una petición PUT al endpoint con un ID de curso inexistente y un cuerpo JSON que intenta asignar un usuario.
3. Se verifica que la respuesta tenga un estado HTTP 404 (Not Found), indicando que el curso no se pudo encontrar y, por lo tanto, no se pudo asignar el usuario.

Verificar que el endpoint maneja correctamente los casos en los que se intenta asignar un usuario a un curso que no existe.

asignarUsuarioConDatosInvalidosTest

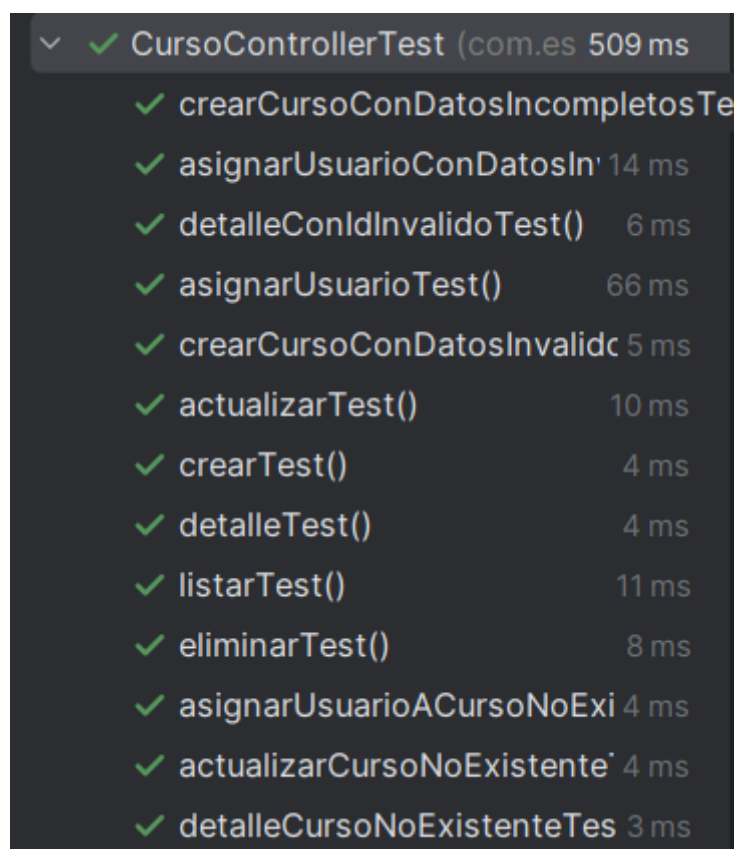
Verificar que el endpoint devuelve un estado HTTP 400 (Bad Request) cuando se proporcionan datos inválidos para la asignación, como un ID de curso o de usuario inválido.

1. Se realiza una petición PUT al endpoint con un ID de curso inválido ("abc") y un cuerpo JSON que contiene un ID de usuario inválido ("xyz").
2. Se verifica que la respuesta tenga un estado HTTP 400 (Bad Request), indicando que los datos proporcionados son inválidos.

Se verifica que el endpoint valida correctamente los datos de entrada y maneja adecuadamente los casos en que estos son inválidos.

Figura 4

Test de Integración en Curso Controller



✓ CursoControllerTest (com.es	509 ms
✓ crearCursoConDatosIncompletosTe	
✓ asignarUsuarioConDatosIn	14 ms
✓ detalleConIdInvalidoTest()	6 ms
✓ asignarUsuarioTest()	66 ms
✓ crearCursoConDatosInvalido	5 ms
✓ actualizarTest()	10 ms
✓ crearTest()	4 ms
✓ detalleTest()	4 ms
✓ listarTest()	11 ms
✓ eliminarTest()	8 ms
✓ asignarUsuarioACursoNoExi	4 ms
✓ actualizarCursoNoExistente	4 ms
✓ detalleCursoNoExistenteTes	3 ms

USUARIO

Pruebas Unitarias de la capa de servicio

Test Listar

```
@Test
public void listarTest() {
    when(usuarioRepository.findAll()).thenReturn(Arrays.asList(new
Usuario()));
    assertNotNull(usuarioService.listar());
    verify(usuarioRepository).findAll();
}

@Test
public void listarTestConUsuariosEspecificos() {
    Usuario usuario1 = new Usuario();
    usuario1.setId(1L);
    usuario1.setNombre("Usuario 1");
    Usuario usuario2 = new Usuario();
    usuario2.setId(2L);
    usuario2.setNombre("Usuario 2");

    when(usuarioRepository.findAll()).thenReturn(Arrays.asList(usuario1, usuario2));
    List<Usuario> usuarios = usuarioService.listar();

    assertNotNull(usuarios);
    assertEquals(2, usuarios.size());
    assertEquals("Usuario 1", usuarios.get(0).getNombre());
    assertEquals("Usuario 2", usuarios.get(1).getNombre());
    verify(usuarioRepository).findAll();
}
```

listarTest

Verificar que el método listar pueda manejar correctamente el caso cuando el repositorio devuelve una lista que contiene un solo usuario (en este caso, un usuario sin datos).

1. Se simula el comportamiento del método `findAll` del repositorio `UsuarioRepository` para que devuelva una lista con un solo `Usuario` (nuevo y sin datos).
2. Se llama al método `listar` del servicio `UsuarioService`.
3. Se verifica que la lista devuelta no sea nula, lo que indica que el método `listar` puede manejar correctamente la respuesta del repositorio.
4. Finalmente, se verifica que el método `findAll` del repositorio fue llamado.

listarTestConUsuariosEspecificos

Verificar que el método `listar` pueda manejar correctamente el caso cuando el repositorio devuelve una lista con usuarios específicos, asegurando que los datos de los usuarios se manejen correctamente.

1. Se simula el comportamiento del método `findAll` del repositorio `UsuarioRepository` para que devuelva una lista conteniendo estos dos usuarios específicos.
2. Se verifica que la lista devuelta no sea nula y que contenga exactamente dos usuarios, lo que indica que el método `listar` puede manejar correctamente la respuesta del repositorio y que los datos de los usuarios se mantienen intactos.
3. Se verifica que los nombres de los usuarios en la lista devuelta coincidan con los nombres de los usuarios específicos creados inicialmente, asegurando que los datos se manejen correctamente.

Test PorID

```
@Test
public void porIdTest() {
    Long id = 1L;

    when(usuarioRepository.findById(id)).thenReturn(Optional.of(new
    Usuario()));
    assertTrue(usuarioService.porId(id).isPresent());
    verify(usuarioRepository).findById(id);
}
```

```

@Test
public void porIdCuandoNoExisteTest() {
    Long idInexistente = 2L;

    when(usuarioRepository.findById(idInexistente)).thenReturn(Optional.empty());
    assertFalse(usuarioService.porId(idInexistente).isPresent());
    verify(usuarioRepository).findById(idInexistente);
}

```

porIdTest

Verificar que el método porId del servicio de usuarios puede encontrar un usuario por su ID.

1. Se configura Mockito para simular que el método findById del repositorio UsuarioRepository devolverá un Optional conteniendo un nuevo usuario cuando se busque por ese ID. Esto simula que el usuario existe en la base de datos.
2. Se verifica que el resultado es un Optional presente, lo que indica que se encontró un usuario.
3. Se verifica que el método findById del repositorio fue llamado exactamente una vez con el ID especificado.

porIdCuandoNoExisteTest

Verificar que el método porId del servicio de usuarios maneja correctamente el caso cuando un usuario no existe.

1. Se define un ID de usuario (2L) que se asume no existe en la base de datos.
2. Se configura Mockito para simular que el método findById del repositorio UsuarioRepository devolverá un Optional vacío cuando se busque por ese ID. Esto simula que el usuario no existe en la base de datos.
3. Se verifica que el resultado es un Optional vacío, lo que indica que no se encontró ningún usuario y se verifica que el método findById del repositorio fue llamado exactamente una vez con el ID inexistente.

Test Guardar

```
@Test
public void guardarTest() {
    Usuario usuario = new Usuario();
    usuario.setNombre("Test User");

    when(usuarioRepository.save(any(Usuario.class))).thenReturn(usuario);

    Usuario savedUsuario = usuarioService.guardar(new Usuario());
    assertEquals("Test User", savedUsuario.getNombre());
    verify(usuarioRepository).save(any(Usuario.class));
}

@Test
public void guardarTestConExcepcion() {
    Usuario usuarioConError = new Usuario();
    usuarioConError.setNombre("Usuario Con Error");

    when(usuarioRepository.save(any(Usuario.class))).thenThrow(new
RuntimeException("Error al guardar"));

    Exception excepcion = assertThrows(RuntimeException.class, ()
-> usuarioService.guardar(usuarioConError));
    assertEquals("Error al guardar", excepcion.getMessage());

    verify(usuarioRepository).save(any(Usuario.class));
}
```

El primer test, `guardarTest`, verifica que al guardar un nuevo usuario con el nombre "Test User", el usuario guardado tenga el mismo nombre. Simula la acción de guardar usando `usuarioRepository.save` y comprueba que el resultado es el esperado.

El segundo test, `guardarTestConExcepcion`, verifica que al intentar guardar un usuario y ocurre un error (simulado por una excepción `RuntimeException`), el servicio maneja

correctamente la excepción. Se asegura que la excepción lanzada tenga el mensaje "Error al guardar".

Test Eliminar

```
@Test
public void eliminarTest() {
    Long id = 1L;
    doNothing().when(usuarioRepository).deleteById(id);
    usuarioService.eliminar(id);
    verify(usuarioRepository).deleteById(id);
}

@Test
public void eliminarUsuarioInexistenteTest() {
    Long idInexistente = 999L;
    doNothing().when(usuarioRepository).deleteById(idInexistente);
    usuarioService.eliminar(idInexistente);
    verify(usuarioRepository).deleteById(idInexistente);
}
```

Los tests `eliminarTest` y `eliminarUsuarioInexistenteTest` verifican el comportamiento del método `eliminar` en `UsuarioService`. El primero asegura que se pueda eliminar un usuario existente, simulando la acción y verificando que el método `deleteById` del repositorio se llama con el ID correcto. El segundo test comprueba que el servicio intenta eliminar un usuario incluso cuando este no existe, también simulando la acción y verificando que `deleteById` se llama con un ID inexistente. Ambos tests utilizan Mockito para simular el comportamiento del repositorio y verificar las interacciones esperadas.

Figura 5

Tests de Usuarios en capa de servicio

✓	✓	UsuarioServiceImplTest	(1 sec 69 ms)
✓		listarTestConUsuarios	1 sec 32 ms
✓		porIdTest()	9 ms
✓		listarTest()	3 ms
✓		eliminarTest()	4 ms
✓		porIdCuandoNoExisteTest()	4 ms
✓		guardarTest()	9 ms
✓		guardarTestConExcepcion()	5 ms
✓		eliminarUsuarioInexistenteTest()	3 ms

Pruebas en el controlador

Test Listar

```
@Test
public void listarTest() throws Exception {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    usuario.setNombre("Juan Perez");
    usuario.setEmail("juan@example.com");
    usuario.setPassword("1234");

    given(usuarioService.listar()).willReturn(Arrays.asList(usuario));
    ;

    mockMvc.perform(get("/api/usuarios/listar")
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.nombre").value("Juan
Perez"));
}
```

```

}

@Test
public void listarTestCuandoListaVacía() throws Exception {
    given(usuarioService.listar()).willReturn(Arrays.asList());

    mockMvc.perform(get("/api/usuarios/listar")
        .contentType(MediaType.APPLICATION_JSON)
        .andExpect(status().isOk())
        .andExpect(content().string("[]")));
}

```

listarTest

Comprobar que el endpoint devuelve correctamente una lista de usuarios cuando hay al menos un usuario disponible.

1. Se crea un usuario de prueba y se configura el servicio para retornar una lista que contiene este usuario cuando se llame al método listar().
2. Se realiza una petición GET al endpoint y se verifica que la respuesta tenga un estado HTTP 200 (OK) y que el contenido de la respuesta incluya el nombre del usuario de prueba.

listarTestCuandoListaVacía

Verificar que el endpoint maneja correctamente el caso en que no hay usuarios disponibles, devolviendo una lista vacía.

1. Se configura el servicio para retornar una lista vacía cuando se llame al método listar().
2. Se realiza una petición GET al endpoint y se verifica que la respuesta tenga un estado HTTP 200 (OK) y que el cuerpo de la respuesta sea una lista vacía ("[]").

Test Detalle

```

@Test
public void detalleTest() throws Exception {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    usuario.setNombre("Juan Perez");

    given(usuarioService.findById(1L)).willReturn(Optional.of(usuario));

    mockMvc.perform(get("/api/usuarios/detalles/{id}", 1L)
        .contentType(MediaType.APPLICATION_JSON)
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.nombre").value("Juan Perez")));
}

@Test
public void detalleUsuarioInexistenteTest() throws Exception {
    Long usuarioIdInexistente = 999L;

    given(usuarioService.findById(usuarioIdInexistente)).willReturn(Optional.empty());

    mockMvc.perform(get("/api/usuarios/detalles/{id}",
        usuarioIdInexistente)
        .contentType(MediaType.APPLICATION_JSON)
        .andExpect(status().isNotFound()));
}

```

detalleTest

Verifica que, al solicitar los detalles de un usuario existente (en este caso, un usuario con ID 1L y nombre "Juan Perez"), el endpoint responde con un estado HTTP 200 (OK) y el nombre correcto del usuario en el cuerpo de la respuesta.

detalleUsuarioInexistenteTest

Comprueba que, al solicitar los detalles de un usuario que no existe (usando un ID inexistente, 999L), el endpoint responde con un estado HTTP 404 (Not Found), indicando que el usuario solicitado no se pudo encontrar.

Test Crear

```
@Test
public void crearTest() throws Exception {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    usuario.setNombre("Juan Perez");

    given(usuarioService.guardar(any(Usuario.class))).willReturn(usuario);

    mockMvc.perform(post("/api/usuarios/crear")
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\"nombre\":\"Juan Perez\",
\"email\":\"juan@example.com\", \"password\":\"1234\"}"))
        .andExpect(status().isCreated())
        .andExpect(jsonPath("$.nombre").value("Juan Perez"));
}
```

crearTest

Verificar que el endpoint crea un usuario correctamente cuando se proporcionan datos válidos. Se espera que el endpoint responda con un estado HTTP 201 (Created) y que el cuerpo de la respuesta contenga el nombre del usuario creado.

Test Editar

```

@Test
public void editarTest() throws Exception {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
    usuario.setNombre("Juan Perez Actualizado");

    given(usuarioService.findById(1L)).willReturn(Optional.of(usuario));

    given(usuarioService.guardar(any(Usuario.class))).willReturn(usuario);

    mockMvc.perform(put("/api/usuarios/actualizar/{id}", 1L)
        .contentType(MediaType.APPLICATION_JSON)
        .content("{\"nombre\":\"Juan Perez Actualizado\", \"email\":\"juan@example.com\", \"password\":\"1234\"}"))
        .andExpect(status().isCreated())
        .andExpect(jsonPath("$.nombre").value("Juan Perez Actualizado"));
}

```

El `editarTest` verifica que el endpoint de actualización de usuarios funcione correctamente. Se prepara un usuario con un ID específico y un nombre actualizado. Se configuran las expectativas para que, al buscar por ID, se devuelva el usuario preparado y, al guardar cualquier usuario, se devuelva el mismo usuario. Luego, se realiza una petición PUT con datos actualizados y se espera que la respuesta tenga un estado HTTP 201 (Created) y que el nombre del usuario en la respuesta coincida con el nombre actualizado.

Test Eliminar

```

@Test
public void eliminarTest() throws Exception {
    Usuario usuario = new Usuario();
    usuario.setId(1L);
}

```

```
given(usuarioService.findById(1L)).willReturn(Optional.of(usuario));
mockMvc.perform(delete("/api/usuarios/eliminar/{id}", 1L))
    .andExpect(status().isNoContent());
}
```

El eliminarTest comprueba que el endpoint de eliminación de usuarios funcione como se espera. Se prepara un usuario con un ID específico y se configura la expectativa de que, al buscar por ese ID, se devuelva un Optional conteniendo el usuario. Luego, se realiza una petición DELETE para eliminar el usuario por su ID y se espera que la respuesta tenga un estado HTTP 204 (No Content), indicando que el usuario fue eliminado exitosamente.

Pruebas Jasmine

Para realizar las pruebas unitarias de Jasmine y Karma se debe seguir los siguientes pasos:

1. Primero debemos instalar las librerías “npm install --save-dev karma karma-jasmine karma-chrome-launcher jasmine-core karma-jasmine-html-reporter karma-coverage”
2. Para que se nos cree el archivo de configuración de Karma para poder indicar “npx karma init”
3. Dentro de nuestro archivo de Karma tendremos las configuraciones iniciales en el que se indicará el puerto, el navegador y lo más importante son los archivos donde haremos las pruebas.

```
module.exports = function(config) {
  config.set({
    basePath: '',
    frameworks: ['jasmine'],
    files: [

'src/app/components/student/student.component.spec.ts',
      'src/app/components/course/course.component.spec.ts'

    ],
```



```

    exclude: [
    ],
    preprocessors: {},
    reporters: ['progress'],
    port: 9876,
    colors: true,
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome'],
    singleRun: false,
    concurrency: Infinity,
  })
}

```

4. Para correr todas el o los archivos de prueba usamos el comando “ng test” y nos correrá todas las pruebas unitarias especificadas en files.

Estudiante

```

it('should load students on init', () => {
  const students = [{ nombre: 'John', email:
'john@example.com', password: '1234' }];
  studentServiceMock.getStudents.and.returnValue(of(students));
  // Ensure it returns an observable

  component.ngOnInit();

  expect(component.students).toEqual(students);
});

```

Esta prueba verifica que cuando el componente se inicializa (**ngOnInit**), se cargue correctamente una lista de estudiantes desde el servicio **studentServiceMock**. La prueba asegura que la lista de estudiantes obtenida del servicio sea asignada a la propiedad **students**

del componente. Esto es esencial para garantizar que los datos de los estudiantes estén disponibles en el componente justo después de su inicialización.

```
it('should add a student', () => {
  const student = { nombre: 'John', email: 'john@example.com',
password: '1234' };
  component.nombreUsuario = 'John';
  component.email = 'john@example.com';
  component.password = '1234';

  component.addStudent();

  expect(studentServiceMock.addStudent).toHaveBeenCalledWith(student);
});
```

Esta prueba verifica que el método `addStudent` del componente funcione correctamente. Al configurar las propiedades `nombreUsuario`, `email`, y `password` del componente, y luego llamar al método `addStudent`, se espera que el servicio `studentServiceMock.addStudent` sea llamado con un objeto estudiante que contiene estos valores. Esta prueba asegura que el componente está construyendo correctamente el objeto estudiante y está llamando al servicio para agregarlo.

```
it('should enroll a student in a course', () => {
  const student = { nombre: 'John', email: 'john@example.com',
password: '1234' };
  const event = { target: { value: '1-Math' } } as unknown as
Event;
  courseServiceMock.enrollStudentInCourse.and.returnValue(of({}));

  component.enrollStudentInCourse(student, event);

  expect(courseServiceMock.enrollStudentInCourse).toHaveBeenCalledWith
```

```
ith(student, 1);  
});
```

Esta prueba verifica que el método `enrollStudentInCourse` del componente funcione correctamente. Cuando se llama a este método con un estudiante y un evento (que contiene información sobre el curso), se espera que el componente extraiga correctamente el ID del curso del evento y llame al servicio `courseServiceMock.enrollStudentInCourse` con el estudiante y el ID del curso. Esta prueba asegura que el componente maneja adecuadamente la inscripción de estudiantes en cursos, enviando los datos correctos al servicio correspondiente.

Curso

```
it('should load courses on init', () => {  
    const courses = [{ nombre: 'Math', cursoUsuarios: [] }];  
    courseServiceMock.getCourses.and.returnValue(of(courses)); //  
    Ensure it returns an observable  
  
    component.ngOnInit();  
  
    expect(component.courses).toEqual(courses);  
});
```

Esta prueba verifica que cuando el componente se inicializa (`ngOnInit`), se cargue correctamente una lista de cursos desde el servicio `courseServiceMock`. La prueba asegura que la lista de cursos obtenida del servicio sea asignada a la propiedad `courses` del componente. Esto es esencial para garantizar que los datos de los cursos estén disponibles en el componente justo después de su inicialización.

```
it('should add a course', () => {  
    const course = { nombre: 'Math' };  
    component.nombre = 'Math';  
  
    component.addCourse();  
});
```

```
expect(courseServiceMock.addCourse).toHaveBeenCalledWith(course);
});
```

Esta prueba verifica que el método `addCourse` del componente funcione correctamente. Al configurar la propiedad `nombre` del componente, y luego llamar al método `addCourse`, se espera que el servicio `courseServiceMock.addCourse` sea llamado con un objeto curso que contiene este nombre. Esta prueba asegura que el componente está construyendo correctamente el objeto curso y está llamando al servicio para agregarlo.

```
it('should return students for a course', () => {
  const course: Course = {
    nombre: 'Math',
    cursoUsuarios: [{
      usuarioId: 1,
      id: 0
    }, {
      usuarioId: 2,
      id: 0
    }]
  };

  const studentIds = component.getStudentsForCourse(course);

  expect(studentIds).toEqual([1, 2]);
});
```

Esta prueba verifica que el método `getStudentsForCourse` del componente funcione correctamente. Al pasar un curso que contiene una lista de `cursoUsuarios` (cada uno con un `usuarioId`), se espera que el método devuelve una lista de IDs de estudiantes (`usuarioId`). Esta prueba asegura que el componente puede extraer correctamente los IDs de estudiantes de un curso.

```
it('should return an empty array if cursoUsuarios is
```

```

undefined', () => {
  const course: Course = {
    nombre: 'Math',
    cursoUsuarios: undefined
  };

  const studentIds = component.getStudentsForCourse(course);

  expect(studentIds).toEqual([]);
});

```

Esta prueba verifica que el método `getStudentsForCourse` del componente maneje correctamente el caso en el que `cursoUsuarios` es `undefined`. Al pasar un curso sin una lista de `cursoUsuarios`, se espera que el método devuelva un arreglo vacío. Esta prueba asegura que el componente puede manejar adecuadamente cursos que no tienen estudiantes asociados y no causa errores cuando `cursoUsuarios` es `undefined`.

Figura 6

Captura de todas las pruebas completadas



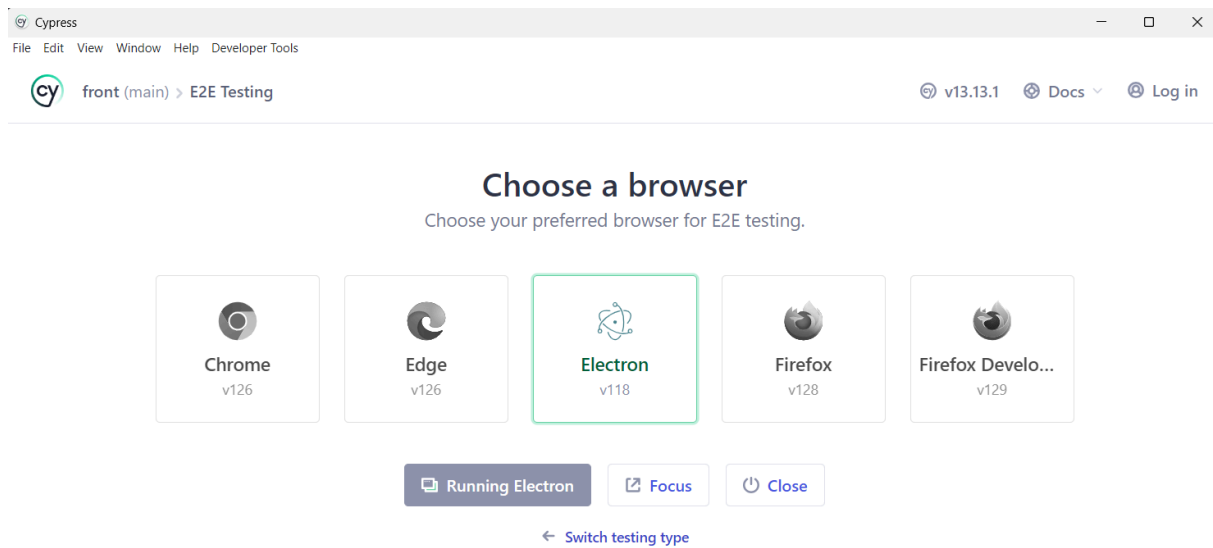
Una vez finalizada la ejecución de las pruebas de Jasmine, se abre en el navegador lo que se ha probado y si las pruebas fueron exitosas o no.

Pruebas E2E

Las pruebas end to end se realizaron con Cypress.

Figura 7

Configuración de Entorno



Se realizó un script en el cual se hicieron diversas pruebas:

```
describe('Student and Course Management', () => {
  beforeEach(() => {
    cy.visit('/');
  });

  it('agregar estudiante', () => {
    cy.get('input[name="nombreUsuario"]').type('Martin Perez');
    cy.get('input[name="email"]').type('mperez@example.com');
    cy.get('input[name="password"]').type('password123');
    cy.get('button').contains('Agregar Estudiante').click();
  });

  it('matricular estudiante en un curso', () => {
    cy.visit('/');
    cy.contains('li', 'Martin Perez').within(() => {
      cy.get('select').select('1 - Matematica');
    });
  });

  it('eliminar estudiante', () => {
    cy.visit('/');
    cy.contains('li', 'Martin Perez').within(() => {
      cy.get('button').contains('Eliminar').click();
    });
  });
});
```

```

    });
  });

  it('agregar estudiante2', () => {
    cy.get('input[name="nombreUsuario"]').type('Juan kings');
    cy.get('input[name="email"]').type('jlongs@example.com');
    cy.get('input[name="password"]').type('password123');
    cy.get('button').contains('Agregar Estudiante').click();
  });

  it('inscribir curso', () => {
    cy.visit('/');
    cy.get('button').contains('Ver Cursos').click();
    cy.get('input[name="nombre"]').type('Ingles');
    cy.get('button').contains('Agregar Curso').click();
  });

  it('matricular estudiante en un curso2', () => {
    cy.visit('/');
    cy.contains('li', 'Juan kings').within(() => {
      cy.get('select').select('8 - Ingles');
    });
  });

  it('verificar matricula', () => {
    cy.visit('/');
    cy.get('button').contains('Ver Cursos').click();
  });
});

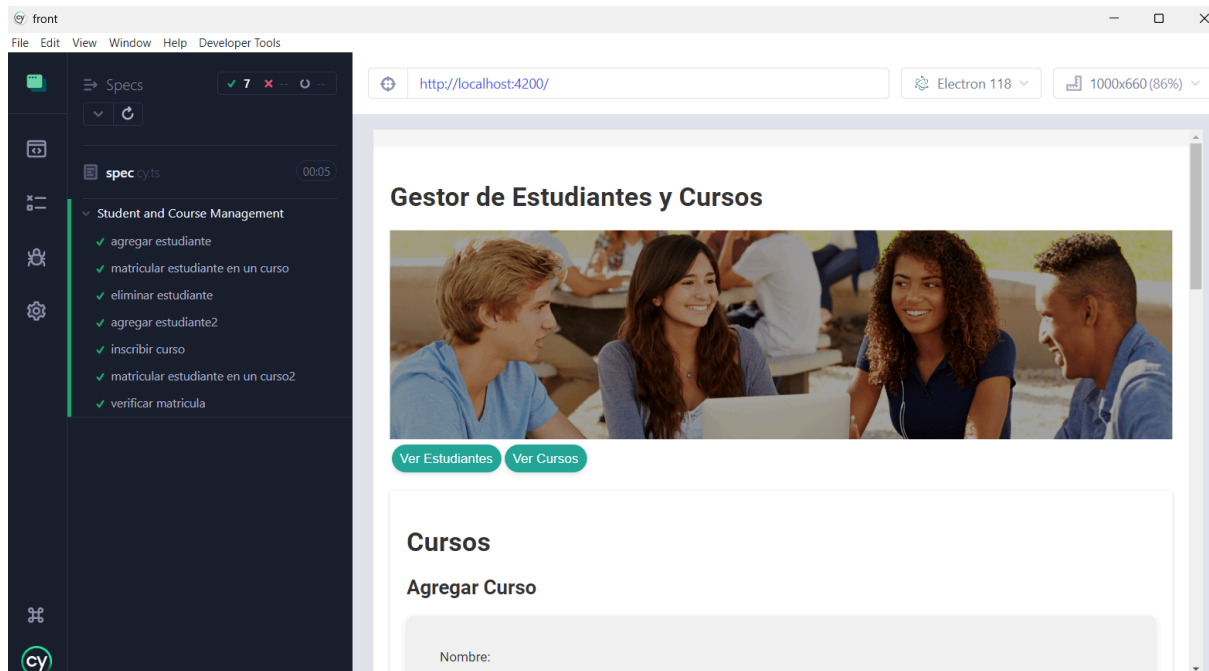
```

Se realizaron las siguientes pruebas con el script para pruebas end-to-end en Cypress:

1. **Agregar Estudiante:** Se añade un nuevo estudiante con el nombre "Martin Perez", email "mperez@example.com" y contraseña "password123".
2. **Matricular Estudiante en un Curso:** Se matricula a "Martin Perez" en el curso "1 - Matemática".
3. **Eliminar Estudiante:** Se elimina al estudiante "Martin Perez" de la lista.
4. **Agregar Estudiante 2:** Se añade un nuevo estudiante con el nombre "Juan Kings", email "jlongs@example.com" y contraseña "password123".
5. **Inscribir Curso:** Se agrega un nuevo curso llamado "Inglés".

6. **Matricular Estudiante en un Curso 2:** Se matricula a "Juan Kings" en el curso "8 - Inglés".
7. **Verificar Matrícula:** Se verifica la lista de cursos existentes.

Figura 8
Pruebas E2E



Conclusiones

Se ha demostrado que un sistema basado en microservicios para la gestión de usuarios y cursos es efectivo y escalable. La división de responsabilidades entre los diversos microservicios ha aumentado la flexibilidad y la facilidad de mantenimiento. El uso de validaciones y manejo de excepciones ha mejorado la experiencia del usuario y garantizado la integridad de los datos. Las pruebas realizadas han confirmado que el sistema es robusto y confiable.

Recomendaciones

1. Usar contenedores y orquestadores como Docker y Kubernetes para gestionar el despliegue y escalabilidad del sistema.
2. Implementar monitoreo continuo y herramientas de logging para identificar y resolver problemas de manera proactiva.

3. Continuar ampliando la cobertura de pruebas para incluir escenarios más complejos y garantizar la calidad del sistema.
4. Evaluar y adoptar nuevas tecnologías y prácticas emergentes en el campo de los microservicios para mantener el sistema actualizado y eficiente.

Referencias Bibliográficas

Carvajal, D., & J. Álvarez. (2020). Propuesta de modelo de arquitectura distribuida en microservicios para la gestión de prestaciones de salud en la empresa Conexia sede Bogotá. .

<https://www.semanticscholar.org/paper/Propuesta-de-modelo-de-arquitectura-distribuida-en-Carvajal-%C3%81lvarez/935f53fd0481e10c56648759571618f7f96459ce>

González Heredia, Octavio, J., de, M., & Karen Cortés Verdín. (2021). Prácticas de los equipos de desarrollo de microservicios: un mapeo sistemático de la literatura.

RECIBE, REVISTA ELECTRÓNICA de COMPUTACIÓN, INFORMÁTICA, BIOMÉDICA Y ELECTRÓNICA;

<https://www.semanticscholar.org/paper/Pr%C3%A1cticas-de-los-equipos-de-desarrollo-de-un-mapeo-Heredia-Hern%C3%A1ndez/3bf325038aa3da37491b32e2deb948a4ff4763ac>

Rios, C., & Fernando, D. (2020). Diseño de un prototipo de una arquitectura basada en microservicios para la integración de aplicaciones Web altamente transaccionales. Caso: Entidades financieras. .

<https://www.semanticscholar.org/paper/Dise%C3%B1o-de-un-prototipo-de-una-arquitectura-basada-Rios-Fernando/1995f50830b706beac1cc2547b22ed5f1c392292>

