CASO DE ESTUDIO N°2: EDUTECH INNOVATORS SPA

Maximiliano Hernández, Renato Valenzuela, Diego Zamora

Justificación de Herramientas y Framework utilizadas

JUnit5: Marco de pruebas unitarias estándar en Java.

- Justificación: Amplia comunidad, integración con Spring Boot, fácil de usar.
- Contribución: Asegura la ejecución adecuada de unidades individuales del código.

Mockito:

- Justificación: Permite la simulación de comportamientos en objetos (mocks).
- Contribución: Facilita pruebas unitarias aisladas al simular dependencias externas.

Postman:

- Justificación: Ideal para probar endpoints RESTful.
- Contribución: Verificación rápida y visual de la comunicación cliente-servidor.

Swagger (OpenAPI):

- Justificación: Estandariza y documenta APIs RESTful.
- Contribución: Permite a los desarrolladores y testers comprender e interactuar con los microservicios fácilmente.

MySQL Workbench:

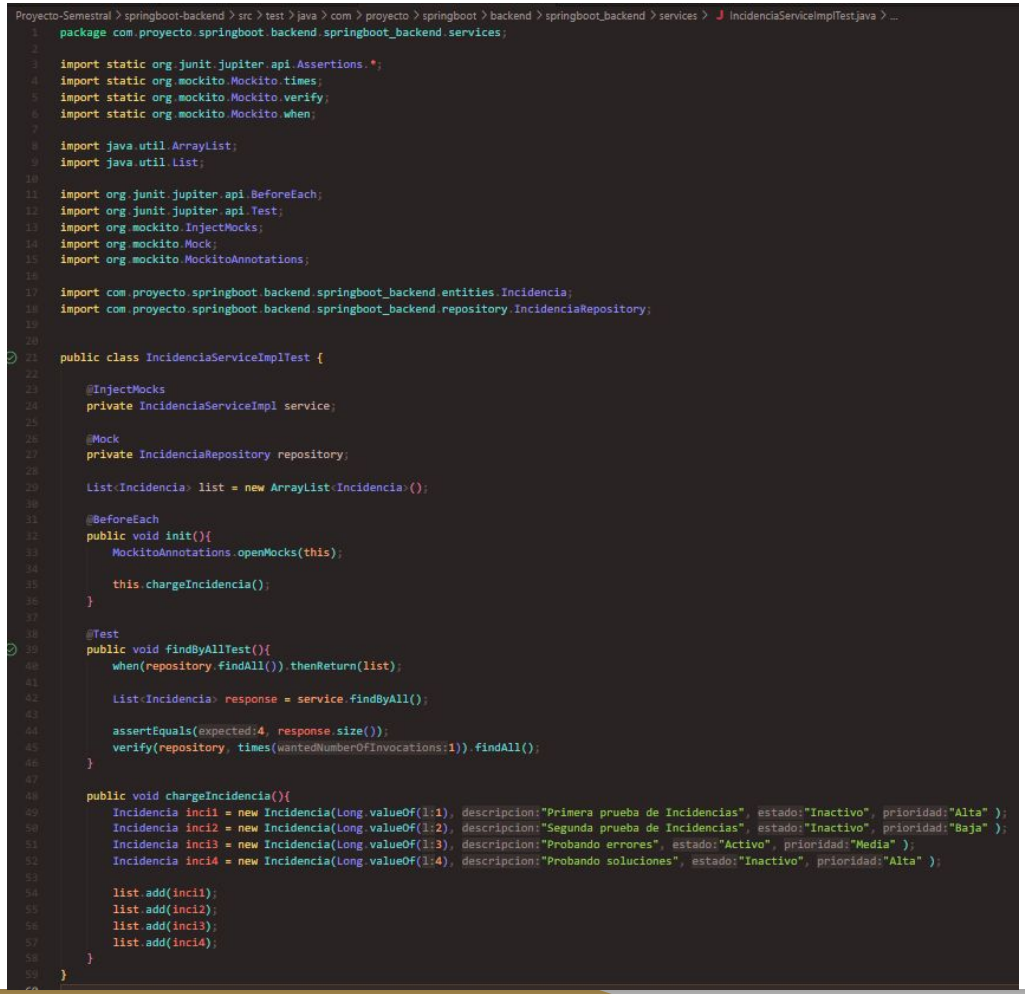
- Justificación: Diseño y administración de bases de datos.
- Contribución: Facilitó la definición y prueba de esquemas de datos.

Spring boot:

- Justificación: Framework compatible con el lenguaje de programación utilizado (Java)
- Contribución: nos facilitó la creación rápida y sencilla de aplicación web y de backend, especialmente APIs REST.

Pruebas unitarias

Para hacer las pruebas unitarias se usó el framework Mockito el cual nos sirve para simular objetos (como Usuarios, Incidencias, Cursos, etc) llamados “mocks” y ver si el CRUD funciona correctamente sin la necesidad de hacer conexiones reales (a la base de datos en este caso) y para ejecutar estos testeos se ocupa como motor de pruebas el framework de JUnit.



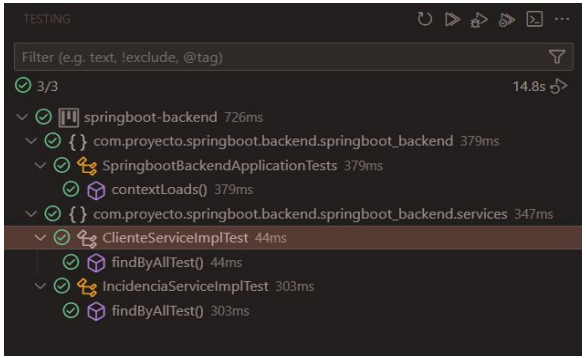
¿Cómo nos aseguramos de la calidad del código desarrollado?

Para asegurarnos de la calidad del código en microservicios con pruebas unitarias usamos JUnit 5 e implementamos Mockito y MockMvc.

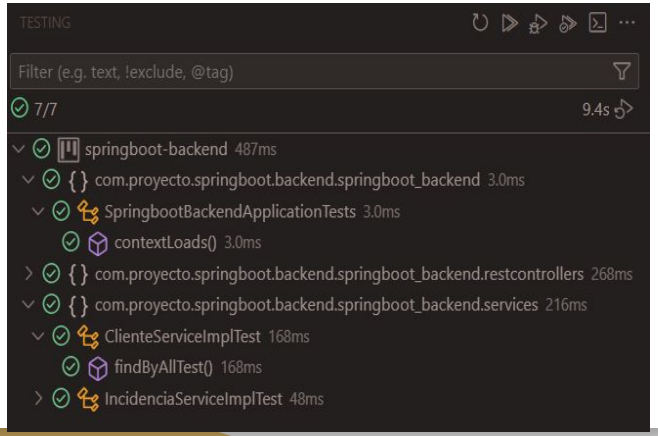
JUnit 5: es un framework de pruebas unitarias que nos ayudó a escribir y ejecutar pruebas automáticas para verificar que nuestro código funciona correctamente.

MockMvc: verifica nuestras rutas, respuestas y errores HTTP.

Mockito: simula las dependencias



Pruebas unitarias

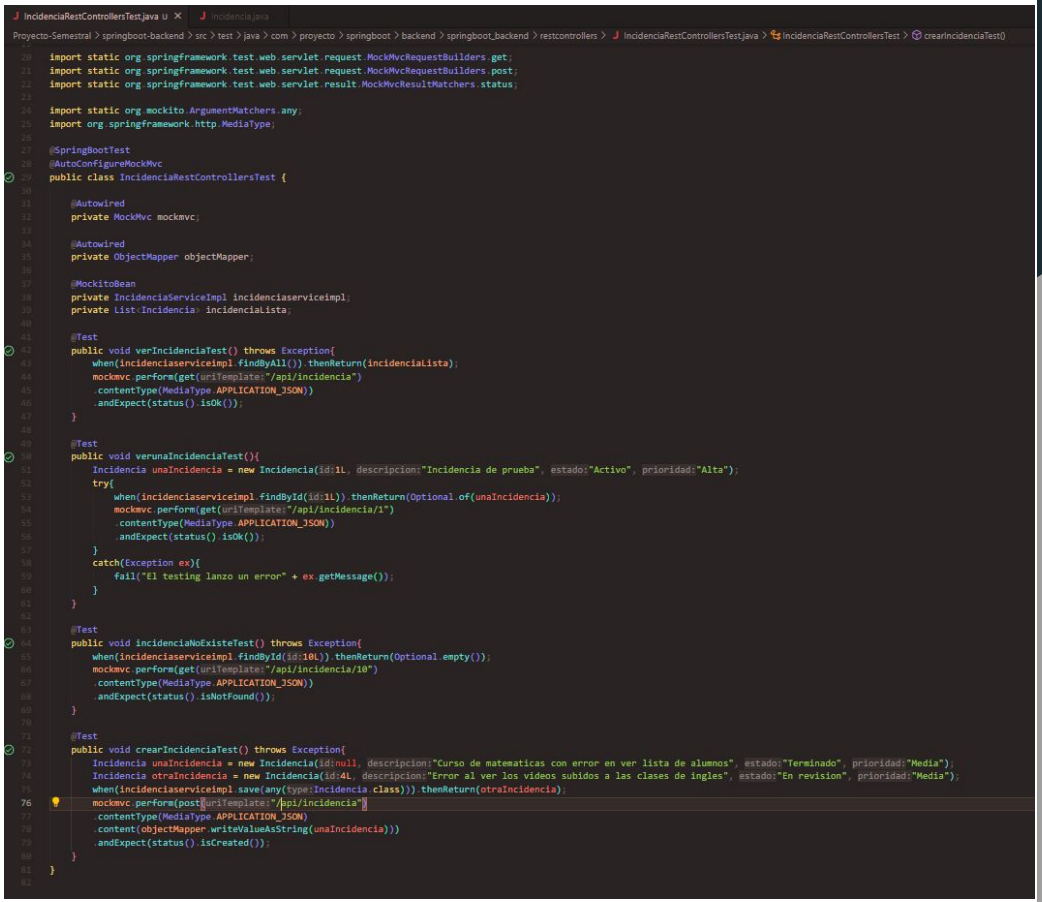


Prueba de integración

Prueba de integración

Para hacer las pruebas de integración se usó una clase utilitaria que viene con Spring-Test llamada MockMVC el cual nos sirve para probar nuestra API Rest, osea, hace simulaciones de peticiones HTTP (Get, Post, Put, etc) sin la necesidad de levantar un servidor real.

Estas pruebas permiten verificar que los controladores, servicios y otros componentes funcionen correctamente de forma integrada. Dependiendo de la configuración, las peticiones pueden interactuar con una base de datos real o en memoria. Para ejecutar estos testeos también se ocupa como motor de pruebas el framework de JUnit.



Componentes de microservicio con OAS

-Edición del pomp

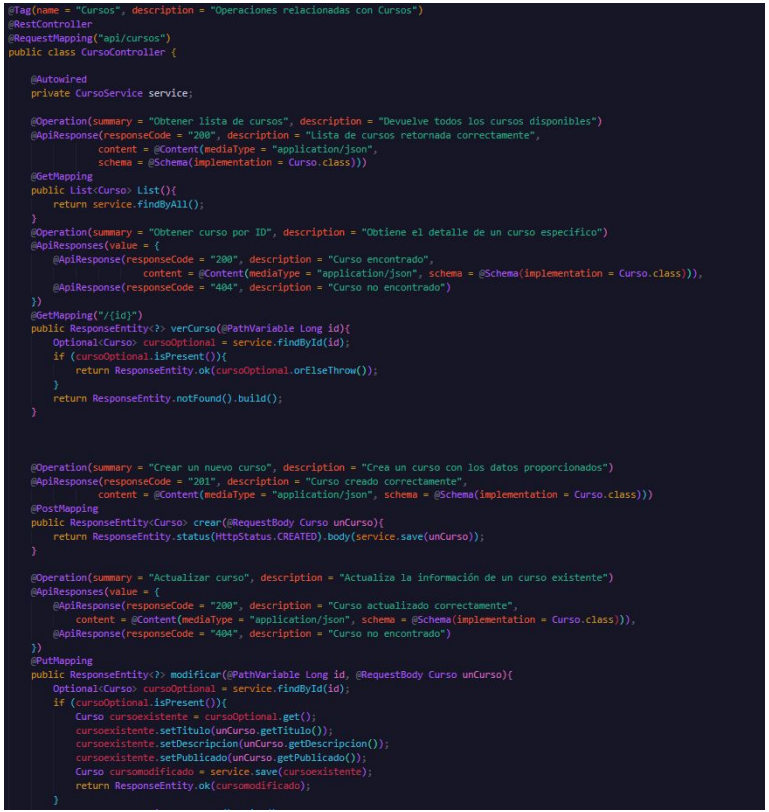
-Se realizó la importación del swagger

-Se agregó:

“Operation”:describe un endpoint en términos de su propósito, parámetros de entrada y respuestas

“ApiResponse”:documentar las posibles respuestas que un endpoint puede devolver

“Tag”: Categorías para agrupar endpoints



```java
@Tag(name = "Cursos", description = "Operaciones relacionadas con Cursos")
@RestController
@RequestMapping("api/cursos")
public class CursoController {

    @Autowired
    private CursoService service;

    @Operation(summary = "Obtener lista de cursos", description = "Devuelve todos los cursos disponibles")
    @ApiResponse(responseCode = "200", description = "Lista de cursos retornada correctamente",
                 content = @Content(mediaType = "application/json",
                 schema = @Schema(implementation = Curso.class)))
    @GetMapping
    public List<Curso> List(){
        return service.findByAll();
    }
    @Operation(summary = "Obtener curso por ID", description = "Obtiene el detalle de un curso especifico")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Curso encontrado",
                     content = @Content(mediaType = "application/json", schema = @Schema(implementation = Curso.class))),
        @ApiResponse(responseCode = "404", description = "Curso no encontrado")
    })
    @GetMapping("/{id}")
    public ResponseEntity<?> verCurso(@PathVariable Long id){
        Optional<Curso> cursoOptional = service.findById(id);
        if (cursoOptional.isPresent()){
            return ResponseEntity.ok(cursoOptional.orElseThrow());
        }
        return ResponseEntity.notFound().build();
    }



    @Operation(summary = "Crear un nuevo curso", description = "Crea un curso con los datos proporcionados")
    @ApiResponse(responseCode = "201", description = "Curso creado correctamente",
                 content = @Content(mediaType = "application/json", schema = @Schema(implementation = Curso.class)))
    @PostMapping
    public ResponseEntity<Curso> crear(@RequestBody Curso unCurso){
        return ResponseEntity.status(HttpStatus.CREATED).body(service.save(unCurso));
    }

    @Operation(summary = "Actualizar curso", description = "Actualiza la información de un curso existente")
    @ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Curso actualizado correctamente",
          content = @Content(mediaType = "application/json", schema = @Schema(implementation = Curso.class))),
        @ApiResponse(responseCode = "404", description = "Curso no encontrado")
    })
    @PutMapping
    public ResponseEntity<?> modificar(@PathVariable Long id, @RequestBody Curso unCurso){
        Optional<Curso> cursoOptional = service.findById(id);
        if (cursoOptional.isPresent()){
            Curso cursoexistente = cursoOptional.get();
            cursoexistente.setTitulo(unCurso.getTitulo());
            cursoexistente.setDescripcion(unCurso.getDescripcion());
            cursoexistente.setPublicado(unCurso.getPublicado());
            Curso cursomodificado = service.save(cursoexistente);
            return ResponseEntity.ok(cursomodificado);
        }
```

Visor del Swagger

(Imagen de referencia, servicio más detallado)