INFORME DE TESTING - UNIÓN

Curso: Ingeniería de Software 1 (2016701)
Grupo:

- Daniel Alejandro Duitama Correa (dduitama@unal.edu.co)
- Edwin Felipe Pinilla Peralta
- Miguel Angel Martinez Fernandez (miamartinezfe@unal.edu.co)
- Juan Sebastián Umaña Camacho (juumanac@unal.edu.co)

Introducción

UNión es una plataforma de comunicación académica diseñada para la comunidad de la Universidad Nacional de Colombia, que centraliza y optimiza la interacción entre estudiantes y profesores. Su objetivo principal es resolver la dispersión de información en múltiples canales (correos, redes sociales, etc.) mediante un entorno seguro y organizado, enfocado en cursos específicos.

Funcionalidades clave:

- Gestión de cursos: Creación, edición y eliminación de cursos por parte de profesores.
- Mensajería en tiempo real: Chat individual y grupal, notificaciones instantáneas.
- Roles y permisos: Acceso diferenciado para estudiantes y profesores.
- Seguridad integrada: Autenticación mediante cuentas institucionales y protocolos de privacidad.
- Herramientas académicas: Encuestas con fecha límite.

Beneficio principal:

Simplifica la colaboración académica, garantizando que toda la comunicación y recursos estén disponibles en un solo lugar, con un diseño intuitivo inspirado en Google Classroom.

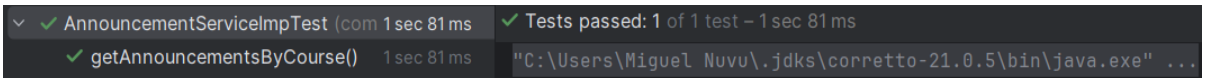
Resumen de Tests Realizados

- Miguel Martinez

- Tipo de prueba: Unitaria
- Componente probado: Servicio: AnnouncementService
- Herramienta: JUnit 5 + Mockito
- Código del test:

```java
@Test
  void getAnnouncementsByCourse() {
    Long courseId = 1L;
    Course course = new Course();
    course.setId(courseId);

    Announcement announcement1 = new Announcement();
    announcement1.setId(1L);
    announcement1.setCourse(course);

    Announcement announcement2 = new Announcement();
    announcement2.setId(2L);
    announcement2.setCourse(course);

    List<Announcement> announcements = Arrays.asList(announcement1,
announcement2);

    when(courseService.existsById(courseId)).thenReturn(true);

when(announcementRepository.findAllByCourseId(courseId)).thenReturn(announce
ments);

    List<Announcement> result =
announcementService.getAnnouncementsByCourse(courseId);

    assertNotNull(result);
    assertEquals(2, result.size());
    assertEquals(announcement1, result.get(0));
    assertEquals(announcement2, result.get(1));

    verify(courseService, times(1)).existsById(courseId);
    verify(announcementRepository, times(1)).findAllByCourseId(courseId);
  }
```

- Resultados:



Lecciones Aprendidas y Dificultades

Aprendizajes clave:

- Una de las ventajas de usar inyección por constructor es que en el test podemos incluir solo la anotación @InjectMocks y Mockito se encargara de inyectar las dependencias necesarias.

Dificultades:

- El diseño del test requiere tiempo incluyendo la creación e inicialización de instancias puntuales.

Mejoras futuras:

- Aumentar la cantidad de pruebas para cada servicio, en especial para funciones críticas del sistema.

- Juan Sebastian Umaña Camacho

- Tipo de prueba: Unitaria
- Componente probado: UserCard.tsx
- Herramienta: Jest + React Testing Library
- Código del test:

```
Unset
import { render, screen } from "@testing-library/react";
import UserCard from "@/components/chats/UserCard/UserCard";
import "@testing-library/jest-dom";

const mockUser = {
  userId: 1,
  userImage: "https://picsum.photos/id/202/48",
  userName: "John Doe",
  userEmail: "john@example.com",
  requestStatus: "pending",
};

describe("Componente UserCard", () => {
  it("muestra correctamente la información del usuario", () => {
    render(<UserCard {...mockUser} />);

    // Verifica que el nombre y el email están en el documento
    expect(screen.getByText(mockUser.userName)).toBeInTheDocument();
    expect(screen.getByText(mockUser.userEmail)).toBeInTheDocument();

    // Verifica que la imagen tiene el alt correcto
    const image = screen.getByRole("img", {
      name: `${mockUser.userName}'s avatar`,
    });
    expect(image).toBeInTheDocument();
  });
```

```
  it("muestra una imagen por defecto cuando no hay imagen de usuario", () =>
{
    render(<UserCard {...mockUser} userImage="" />);

    // Asegura que se renderiza una imagen con el alt correcto
    const image = screen.getByRole("img", {
      name: `${mockUser.userName}'s avatar`,
    });

    expect(image).toBeInTheDocument();
    expect(image).toHaveAttribute("src"); // No verificamos la URL porque
Next.js la transforma (next/Image)
  });

  it("maneja correctamente nombres y correos electrónicos largos", () => {
    const usuarioLargo = {
      ...mockUser,
      userName: "Johnathan Alexander Gomez García",
      userEmail: "johnathan.alexander.gomez.garcia@ejemplocorreolargo.com",
    };

    render(<UserCard {...usuarioLargo} />);

    // Verifica que el nombre y el email están presentes (truncate)
    expect(screen.getByText(usuarioLargo.userName)).toBeInTheDocument();
    expect(screen.getByText(usuarioLargo.userEmail)).toBeInTheDocument();
  });
});
```

- Resultados:

```
PASS __tests__/UserCard.test.tsx
  Componente UserCard
    √ muestra correctamente la información del usuario (88 ms)
    √ muestra una imagen por defecto cuando no hay imagen de usuario (77 ms)
    √ maneja correctamente nombres y correos electrónicos largos (17 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.745 s
Ran all test suites.
```

Lecciones Aprendidas y Dificultades

Aprendizajes clave:

- La prueba verifica correctamente la visualización de la información del usuario, incluyendo nombre, correo e imagen.
- Se comprobó que el componente maneja casos especiales, como la ausencia de imagen y nombres/correos largos.
- Testing Library facilita la validación de elementos en el DOM y sus atributos.

Dificultades:

- No se puede verificar directamente la URL de la imagen porque Next.js transforma las rutas al renderizar con next/image.

Mejoras futuras:

- Implementar pruebas visuales para verificar cómo se comporta el componente con distintos tamaños de texto e imágenes.
- Agregar validaciones adicionales para asegurar que el diseño responda adecuadamente en diferentes dispositivos y tamaños de pantalla.
- Incluir pruebas para verificar eventos interactivos, como botones dentro de la tarjeta de usuario, si los hubiera.

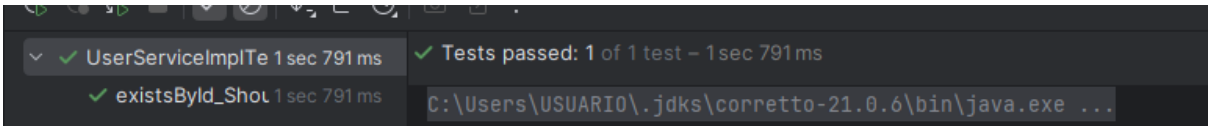
Tipo de prueba: Unitaria
Componente probado: Servicio: `UserServiceImpl`
Herramienta: JUnit 5 + Mockito

Código del test:

```
Unset
package com.union.unionbackend.services.impl;

import com.union.unionbackend.repositories.UserRepository;
import
com.union.unionbackend.services.userService.UserServiceImpl;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
```

```
class UserServiceImplTest {

  @Mock
  private UserRepository userRepository;

  @InjectMocks
  private UserServiceImpl userService;

  @Test
  void existsById_ShouldReturnTrue_WhenUserExists() {
    String userId = "123";
    when(userRepository.existsById(userId)).thenReturn(true);
    assertTrue(userService.existsById(userId));
  }

  @Test
  void existsById_ShouldReturnFalse_WhenUserDoesNotExist() {
    String userId = "456";
    when(userRepository.existsById(userId)).thenReturn(false);
    assertFalse(userService.existsById(userId));
  }
}
```

Resultados:



- Se validó que el servicio `UserServiceImpl` responde correctamente al verificar la existencia de un usuario.
- Se simularon diferentes casos usando Mockito.

Lecciones Aprendidas y Dificultades

Aprendizajes clave:

- El uso de Mockito simplifica la simulación de dependencias, evitando la necesidad de una base de datos real.
- Las pruebas unitarias ayudan a detectar posibles errores lógicos en el código antes de la implementación en producción.

Dificultades:

- La configuración inicial de Mockito tomó tiempo, especialmente al definir correctamente las anotaciones `@Mock` y `@InjectMocks`.

Mejoras futuras:

- Ampliar la cobertura de pruebas para métodos adicionales del servicio.
- Implementar pruebas para validar excepciones y comportamiento en escenarios de error.

Daniel Duitama

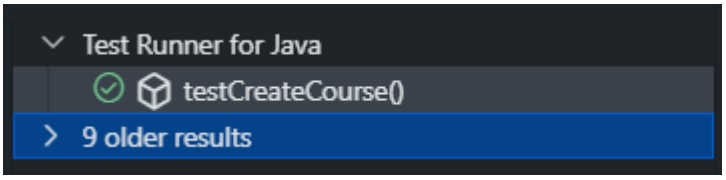
Tipo de prueba: Unitaria
Componente probado: Servicio: `CourseServiceImpl`
Herramienta: JUnit 5 + Mockito

Código del test:

```java
package com.union.unionbackend.services.courseService;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import static org.mockito.ArgumentMatchers.any;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import static org.mockito.Mockito.when;
import org.mockito.junit.jupiter.MockitoExtension;

import com.union.unionbackend.models.Course;
import com.union.unionbackend.repositories.CourseRepository;

@ExtendWith(MockitoExtension.class)
class CourseServiceTest {

    @Mock
    private CourseRepository courseRepository;

    @InjectMocks
```

```
    private CourseServiceImp courseService;



@Test
@DisplayName("🚀 Prueba la creación de un curso")
void testCreateCourse() {
    Course course = new Course();
    course.setId(1L);
    course.setName("Matemáticas");

    when(courseRepository.save(any(Course.class))).thenReturn(course);

    Course createdCourse = courseService.createCourse(course);

    assertNotNull(createdCourse);
    assertEquals(1L, createdCourse.getId());
    assertEquals("Matemáticas", createdCourse.getName());

    System.out.println("✅ Test de creación de curso exitoso!");
}

}
```

Resultados:



- Se validó que el servicio `CourseServiceImpl` en la creación de un curso funcionara correctamente.

Lecciones Aprendidas y Dificultades

Aprendizajes clave:

- Fue fácil y sencillo al menos para este método crear el test con mockito y JavaUnit.
- Validación de la correcta creación de un curso mediante **JUnit 5**.
- Implementación de pruebas unitarias con **inyección de dependencias** en los servicios.

Dificultades:

- Necesidad de garantizar que los datos simulados en **Mockito** reflejen el comportamiento real del servicio.

Mejoras futuras:

- Agregar pruebas para **actualización y eliminación** de cursos.
- Incluir validaciones para manejar **casos de error**, como curso no encontrado (`Optional.empty()`).
- Implementar pruebas para verificar la integración con `EnrollmentService`.