Testing ClickMunch

Introducción breve ClickMunch

Nuestra aplicación es una plataforma innovadora diseñada para transformar la experiencia gastronómica de los usuarios y optimizar el servicio en los restaurantes. Con un enfoque en la comodidad y la interactividad, la aplicación permite a los usuarios: Explorar Menús, Realizar Pedidos, Valorar y Recomendar.

En cuanto al desarrollo del proyecto en el back-end, utilizamos Java con el framework Spring Boot, el front-end está desarrollado con React Native, la base de datos que implementamos es PostgreSQL, un sistema de gestión de bases de datos relacional altamente escalable y confiable. Además, usamos Docker para la contenedorización y en términos de pruebas, usamos JUnit con Mockito para pruebas unitarias en el back-end.

Resumen de los tests:

1) **Nombre del integrante:** Johan Sebastian Roa Rodriguez
   Tipo de prueba realizada: Tres pruebas unitarias
   Descripción breve del componente utilizado: Me enfoque en hacer las pruebas del OrderController que maneja todo el tema de las ordenes en el backend, hice tres pruebas para los endpoints GET /api/orders, GET /api/orders/{username} y POST /api/orders/new-order
   Herramienta o framework usado: Use JUnit 5, Mockito y MockMvc
   Screenshot del código del test:
   Test 1

```
@Test
public void testFindOrdersByUser() throws Exception {
    // Crea una orden de prueba
    Order testOrder = new Order( id: 1,  userId: 1,  storeId: 1, Set.of(), Set.of(), Set.of(),  total: 10.0, Status.PENDING, Payment.CASH);

    // Define el comportamiento de los mocks
    when(userService.findByUsername("testUser")).thenReturn(testUser);
    when(orderService.findByUserId(1)).thenReturn(List.of(testOrder));

    // Realiza la petición GET y verifica la respuesta
    mockMvc.perform(get( uriTemplate: "/api/orders/testUser"))
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "$.length()").value( expectedValue: 1))
            .andExpect(jsonPath( expression: "$[0].id").value( expectedValue: 1));
}
```

   Test 2

```
@Test
public void testFindAllOrders() throws Exception {
    Order testOrder = new Order( id: 1,  userId: 1,  storeId: 1, Set.of(), Set.of(), Set.of(),  total: 20.0, Status.PENDING, Payment.CASH);
    when(orderService.findAll()).thenReturn(List.of(testOrder));

    mockMvc.perform(get( uriTemplate: "/api/orders"))
            .andExpect(status().isOk())
            .andExpect(jsonPath( expression: "$.length()").value( expectedValue: 1))
            .andExpect(jsonPath( expression: "$[0].id").value( expectedValue: 1));
}
```

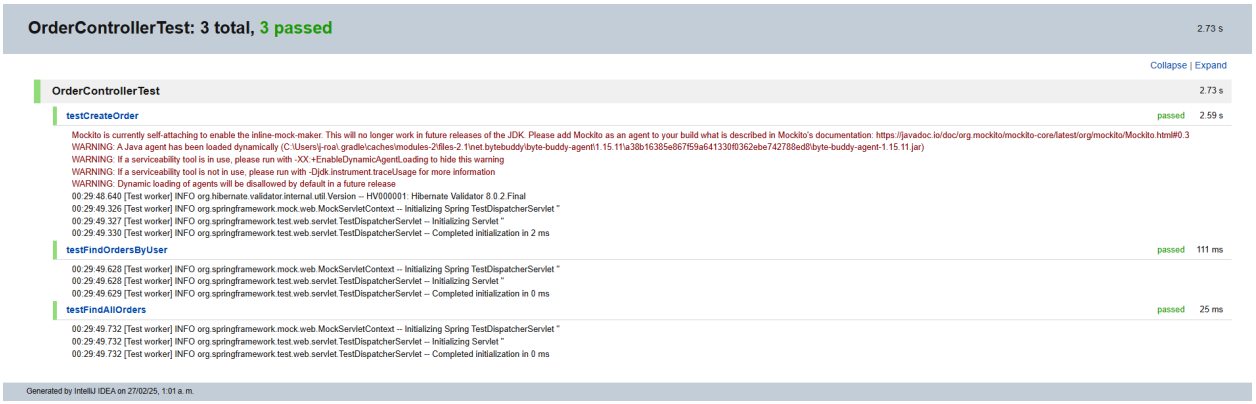
   Test 3

```java
@Test
public void testCreateOrder() throws Exception {
    // Ejemplo de JSON para crear una nueva orden
    String jsonRequest = """
            {
                "userId": 1,
                "storeId": 1,
                "plateIds": [],
                "drinkIds": [],
                "dessertIds": [],
                "total": 20.0,
                "paymentMethod": "CASH"
            }
            """;


    when(userService.findById(1)).thenReturn(testUser);

    mockMvc.perform(post( uriTemplate: "/api/orders/new-order")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(jsonRequest))
            .andExpect(status().isOk());
    }
}
```

Resultado de la ejecución:



Por si no se aprecia bien en la imagen los tres test fueron pasados de una manera exitosa

2) **Nombre del integrante:** Santiago Bejarano Ariza
Tipo de prueba realizada: Pruebas unitarias para el StoreController, encargado de la gestión de restaurantes.
Descripción breve del componente utilizado:
Las pruebas se enfocan en validar el correcto funcionamiento de los endpoints del controlador StoreController, que permite gestionar la información de los restaurantes en la plataforma. Se han realizado pruebas para los siguientes casos de uso:

GET /api/stores → Obtiene la lista de todos los restaurantes.
GET /api/stores/{id} → Busca un restaurante por su ID.
GET /api/stores/name/{name} → Busca restaurantes por su nombre.
POST /api/stores → Crea un nuevo restaurante, asegurando que no haya duplicados.

Herramientas o framework usado:
JUnit 5 para la ejecución de pruebas unitarias.
Mockito para simular el comportamiento de las dependencias (`StoreService`).
Spring Boot Test para facilitar la validación del comportamiento de los endpoints.

Código de testing:

```
@Test
void findAll_ShouldReturnListOfStores() {

when(storeService.findAll()).thenReturn(Arrays.asList(store1,
store2));
   List<Store> result = storeController.findAll();
   assertEquals(2, result.size());
   extracted(result);
}

private static void extracted(List<Store> result) {
     assertFalse(result.isEmpty(), "El resultado no debería
estar vacío");
   assertEquals("Restaurant A", result.getFirst().name());

}

@Test
void findAll_ShouldThrowException_WhenNoStoresFound() {
   when(storeService.findAll()).thenReturn(List.of());
                ResponseStatusException exception =
assertThrows(ResponseStatusException.class,
storeController::findAll);
       assertEquals("404 NOT_FOUND \"No stores found\"",
exception.getMessage());
}

@Test
void findById_ShouldReturnStore_WhenExists() {
   when(storeService.findById(1)).thenReturn(store1);
   Store result = storeController.findById(1);
   assertEquals("Restaurant A", result.name());
}

@Test
void findById_ShouldThrowException_WhenNotFound() {
                when(storeService.findById(3)).thenThrow(new
ResponseStatusException(HttpStatus.NOT_FOUND, "Store not
found"));
```

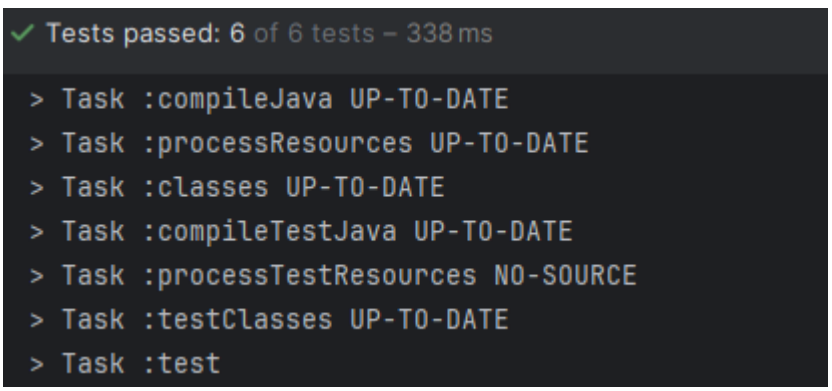
```
                ResponseStatusException exception =
assertThrows(ResponseStatusException.class, () ->
storeController.findById(3));
       assertEquals("404 NOT_FOUND \"Store not found\"",
exception.getMessage());
}

@Test
void findByName_ShouldReturnListOfStores() {
                    when(storeService.findByName("Restaurant
A")).thenReturn(Collections.singletonList(store1));
                        List<Store> result =
storeController.findByName("Restaurant A");
   assertEquals(1, result.size());
   assertEquals("Restaurant A", result.getFirst().name());
}

@Test
void create_ShouldThrowException_WhenStoreAlreadyExists() {
                    when(storeService.findByName("Restaurant
A")).thenReturn(Collections.singletonList(store1));
   Store newStore = new Store(
           3, "Restaurant A", "Alias A",
           "test@email.com", "Password123", "Calle 123",
           5.123456, -74.654321,
           List.of(), List.of(), List.of()
   );

                ResponseStatusException exception =
assertThrows(ResponseStatusException.class, () ->
storeController.create(newStore));
     assertEquals("403 FORBIDDEN \"Store already exists\"",
exception.getMessage());
}
```

Resultado del testing:



3. Nombre del Integrante: **Michael Stiven Betancourt Gelves**
Tipo de prueba realizada: Tests unitarios para el controlador del usuario

Descripción Breve del componente utilizado: Las pruebas se hicieron con la finalidad de probar el correcto funcionamiento de los endpoints del controlador de los usuarios, probando los métodos GET y POST.

Herramientas o framework usado:
JUnit 5 para la ejecución de pruebas unitarias.
Mockito para simular el comportamiento de las dependencias e inyectarlas en el test.
Spring Boot Test para la ejecución de la validación de los tests.

Código realizado:

```
package com.bestellen.click_munch.user;

import org.junit.jupiter.api.BeforeEach;
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 java.util.List;
import java.util.Set;

import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

@ExtendWith(MockitoExtension.class)
public class UserControllerTest {

    @Mock
    private UserService userService;

    @InjectMocks
    private UserController userController;

    private User userA, userB;

    @BeforeEach
    void setUp() {
        userA = new User(
                1, "userA", "testA@test.com",
                "passwordA", "passA", "3142589654", Set.of());
        userB = new User(
                2, "userB", "testB@test.com",
                "passwordB", "passB", "3142589654", Set.of());

    }

    @Test
    void shouldFindAll() {
        when(userService.findAll()).thenReturn(List.of(userA, userB));
        assertEquals(List.of(userA, userB), userController.findAll());
    }
```

```java
    @Test
    void shouldFindById() {
        when(userService.findById(1)).thenReturn(userA);
        assertEquals(userA, userController.findById(1));
    }

    @Test
    void shouldFindByUsername() {
        when(userService.findByUsername("userA")).thenReturn(userA);
        assertEquals(userA, userController.findByUsername("userA"));
    }

    @Test
    void shouldCreate() {
        userController.create(userA);
        verify(userService, times(1)).save(userA);
    }

}
```

Resultado de la ejecución:

Somos conscientes de que se requiere realizar muchas más pruebas y que las pruebas de integración serán fundamentales en el desarrollo de la aplicación, así que continuaremos con el testeo de los endpoints de la API para tener el mejor resultado posible, además, de acuerdo con la sesión de TDD, entendemos que es importante desarrollar los tests con la finalidad de acotar el alcance de la aplicación, entendiendo que las pruebas son necesarias durante todo el ciclo de vida del proyecto.