Sprint 4: Test Unitarios backend equipo 9

**Configuraciones previas**

package com.grupo9.digitalBooking.music.model.controller;  
  
  
import com.grupo9.digitalBooking.music.model.DTO.BookingDTO;  
import com.grupo9.digitalBooking.music.model.DTO.BookingResponseDTO;  
import com.grupo9.digitalBooking.music.model.DTO.CategoryDTO;  
import com.grupo9.digitalBooking.music.model.entities.Booking;  
import com.grupo9.digitalBooking.music.model.service.InterfacesService.IBookingService;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.ResponseEntity;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.Collection;  
import java.util.Set;

**A. Class BookingControllerTest**

class BookingControllerTest {  
 @Mock  
 private IBookingService bookingService;  
  
 @InjectMocks  
 private BookingController bookingController;  
  
 @BeforeEach  
 void setUp() {  
 MockitoAnnotations.*openMocks*(this);  
 }

1. **createBooking\_ValidBooking\_ReturnsOkResponse()**

Este test verifica el comportamiento del método "createBooking" en la clase "BookingController" y valida la correcta creacion de una reserva dentro de la API.

@Test  
void createBooking\_ValidBooking\_ReturnsOkResponse() {  
 // Arrange  
 BookingDTO bookingDTO = new BookingDTO();  
 bookingDTO.setId(1L);  
 bookingDTO.setStartDate(LocalDate.*now*());  
 bookingDTO.setFinalDate(LocalDate.*now*().plusDays(1));  
  
 BookingResponseDTO bookingResponseDTO = new BookingResponseDTO();  
 bookingResponseDTO.setId(1L);  
 bookingResponseDTO.setStartDate(LocalDate.*now*());  
 bookingResponseDTO.setFinalDate(LocalDate.*now*().plusDays(1));  
  
 *when*(bookingService.createBooking(*any*(BookingDTO.class))).thenReturn(bookingResponseDTO);  
  
 // Act  
 ResponseEntity<?> response = bookingController.createBooking(bookingDTO);  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
 *assertEquals*(bookingResponseDTO, response.getBody());  
}

1. **createBooking\_ExistingBooking\_ReturnsBadRequestResponse()**

Este test verifica el comportamiento del método "createBooking" en la clase "BookingController" y valida lo que ocurre cuando no se crea correctamente la reserva.

@Test  
void createBooking\_ExistingBooking\_ReturnsBadRequestResponse() {  
 // Arrange  
 BookingDTO bookingDTO = new BookingDTO();  
  
 *when*(bookingService.createBooking(*any*(BookingDTO.class))).thenReturn(null);  
  
 // Act  
 ResponseEntity<?> response = bookingController.createBooking(bookingDTO);  
  
 // Assert  
 *assertEquals*(HttpStatus.*BAD\_REQUEST*, response.getStatusCode());  
 *assertEquals*("Message: The Reservation already exists", response.getBody());  
}

1. **getBooking\_ExistingId\_ReturnsOkResponse()**

Este test verifica el comportamiento del método "getBooking" en la clase "BookingController" y valida que este devuelva una reserva, utilizando el ID.

@Test  
void getBooking\_ExistingId\_ReturnsOkResponse() {  
 // Arrange  
 Long bookingId = 1L;  
  
 BookingResponseDTO bookingResponseDTO = new BookingResponseDTO();  
 bookingResponseDTO.setId(bookingId);  
  
 *when*(bookingService.readBooking(*anyLong*())).thenReturn(bookingResponseDTO);  
  
 // Act  
 ResponseEntity<?> response = bookingController.getBooking(bookingId);  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
 *assertEquals*(bookingResponseDTO, response.getBody());  
}

1. **getBooking\_NonExistingId\_ReturnsBadRequestResponse()**

Este test verifica el comportamiento del método "getBooking" en la clase "BookingController" y valida que se devuelva una respuesta negativa, si se busca una reserva con un numero de ID que no existe

@Test  
void getBooking\_NonExistingId\_ReturnsBadRequestResponse() {  
 // Arrange  
 Long bookingId = 1L;  
  
 *when*(bookingService.readBooking(*anyLong*())).thenReturn(null);  
  
 // Act  
 ResponseEntity<?> response = bookingController.getBooking(bookingId);  
  
 // Assert  
 *assertEquals*(HttpStatus.*BAD\_REQUEST*, response.getStatusCode());  
 *assertEquals*("Message: The Reservation with " + bookingId + " does not exist", response.getBody());  
}

1. **modifyBooking\_ExistingBooking\_ReturnsOkResponse()**

Este test verifica el comportamiento del método "modifyBooking" en la clase "BookingController" y valida que al modificar una reserva esta se guarde de manera exitosa

@Test  
void modifyBooking\_ExistingBooking\_ReturnsOkResponse() {  
 // Arrange  
 BookingDTO bookingDTO = new BookingDTO();  
 bookingDTO.setId(1L);  
  
 *when*(bookingService.modifyBooking(*any*(BookingDTO.class))).thenReturn(new BookingResponseDTO());  
  
 // Act  
 ResponseEntity<?> response = bookingController.modifyBooking(bookingDTO);  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
}

1. **removeBooking\_ExistingBooking\_ReturnsOkResponse()**

Este test verifica el comportamiento del método "removeBooking" en la clase "BookingController" y valida si al ejecutar el metodo, se elimina la reserva.

@Test  
void removeBooking\_ExistingBooking\_ReturnsOkResponse() {  
 // Arrange  
 Long bookingId = 1L;  
  
 *when*(bookingService.removeBooking(*anyLong*())).thenReturn(true);  
  
 // Act  
 ResponseEntity<?> response = bookingController.removeBooking(bookingId);  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
 *assertEquals*("Message: Booking was delete", response.getBody());  
}

1. **removeBooking\_NonExistingBooking\_ReturnsNotFoundResponse()**

Este test verifica el comportamiento del método "removeBooking" en la clase "BookingController" y valida que envie un error si la reserva que se esta intentando borrar no existe.

@Test  
void removeBooking\_NonExistingBooking\_ReturnsNotFoundResponse() {  
 // Arrange  
 Long bookingId = 1L;  
  
 *when*(bookingService.removeBooking(*anyLong*())).thenReturn(false);  
  
 // Act  
 ResponseEntity<?> response = bookingController.removeBooking(bookingId);  
  
 // Assert  
 *assertEquals*(HttpStatus.*NOT\_FOUND*, response.getStatusCode());  
 *assertEquals*("Message: The Reservation with " + bookingId + " does not exist", response.getBody());  
}

1. **getAllBookings\_ExistingBookings\_ReturnsOkResponse()**

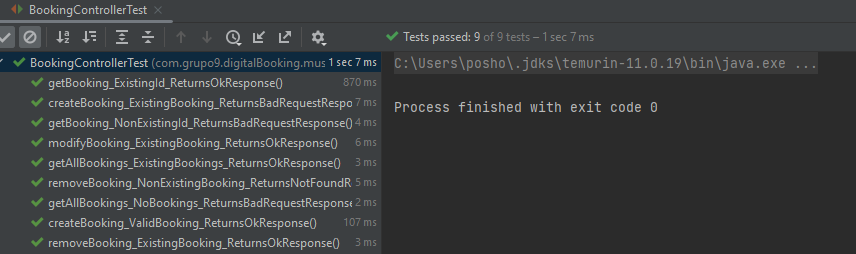
Este test verifica el comportamiento del método "getallBooking" en la clase "BookingController" y valida que este devuelva todas reservas creadas.

@Test  
void getAllBookings\_ExistingBookings\_ReturnsOkResponse() {  
 // Arrange  
 BookingResponseDTO bookingResponseDTO = new BookingResponseDTO();  
 bookingResponseDTO.setId(1L);  
  
 Set<BookingResponseDTO> bookings = Collections.*singleton*(bookingResponseDTO);  
  
 *when*(bookingService.getAll()).thenReturn(bookings);  
  
 // Act  
 ResponseEntity<?> response = bookingController.getallBookings();  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
 *assertEquals*(bookings, response.getBody());  
}

1. **getAllBookings\_NoBookings\_ReturnsBadRequestResponse()**

Este test verifica el comportamiento del método "getallBooking" en la clase "BookingController" y verifica cuando hay algun error en el envio de cada una de las reservas.

@Test  
void getAllBookings\_NoBookings\_ReturnsBadRequestResponse() {  
 // Arrange  
 *when*(bookingService.getAll()).thenReturn(Collections.*emptySet*());  
  
 // Act  
 ResponseEntity<?> response = bookingController.getallBookings();  
  
 // Assert  
 *assertEquals*(HttpStatus.*BAD\_REQUEST*, response.getStatusCode());  
 *assertEquals*("Message: There are not information", response.getBody());  
}



**Configuraciones Previas**

package com.grupo9.digitalBooking.music.controllerTest;  
  
import com.grupo9.digitalBooking.music.model.DTO.UserDTO;  
import com.grupo9.digitalBooking.music.model.DTO.UserResponseDTO;  
import com.grupo9.digitalBooking.music.model.controller.UserController;  
import com.grupo9.digitalBooking.music.model.entities.Rol;  
import com.grupo9.digitalBooking.music.model.service.UserServiceApi;  
import com.grupo9.digitalBooking.music.model.service.RolService;  
import com.grupo9.digitalBooking.music.model.repository.IUser;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import org.mockito.Mock;  
import org.mockito.MockitoAnnotations;  
import org.springframework.http.HttpStatus;  
import org.springframework.http.ResponseEntity;  
import org.mockito.InjectMocks;  
  
  
import java.util.Collections;  
import java.util.HashSet;  
import java.util.Set;  
  
import static org.junit.jupiter.api.Assertions.*assertEquals*;  
import static org.junit.jupiter.api.Assertions.*assertNotNull*;  
import static org.mockito.Mockito.\*;

**B. Class UserControllerTest**

public class UserControllerTest {  
 @InjectMocks  
 private UserController userController;  
  
 @Mock  
 private UserServiceApi userService;  
  
 @Mock  
 private IUser userRepository;  
  
 @Mock  
 private RolService rolService;  
  
 @BeforeEach  
 public void setUp() {  
 MockitoAnnotations.*openMocks*(this);  
 }

1. **createUser\_RolDoesNotExist\_ReturnsBadRequest()**

Este test verifica el comportamiento del método "createUser" en la clase "UserController" y verifica que arroje un error cuando el rol que se intenta implementar no existe.

@Test  
public void createUser\_RolDoesNotExist\_Returns400BadRequest() {  
 // Arrange  
 UserDTO userDTO = new UserDTO();  
 userDTO.setId(1L);  
 userDTO.setName("John");  
 userDTO.setLastName("Doe");  
 userDTO.setDni("123456789");  
 userDTO.setPassword("password");  
 userDTO.setAddress("123 Main St");  
 userDTO.setEmail("john.doe@example.com");  
 Rol rolDTO = new Rol();  
 rolDTO.setId(1L);  
 rolDTO.setName("RoleName");  
 userDTO.setRol(rolDTO);  
  
 *when*(rolService.existById(rolDTO.getId())).thenReturn(false);  
  
 // Act  
 ResponseEntity<?> response = userController.createUser(userDTO);  
  
 // Assert  
 *assertEquals*(HttpStatus.*BAD\_REQUEST*, response.getStatusCode());  
  
 String message = (String) response.getBody();  
 *assertEquals*("Message: The rol does not exist", message);  
  
}

1. **getUser\_ExistingUserId\_ReturnsOK()**

Este test verifica el comportamiento del método "getUser" en la clase "UserController" y verifica la correcta obtencion de un usuario por su ID.

@Test  
public void getUser\_ExistingUserId\_Returns200OK() {  
 // Arrange  
 Long userId = 1L;  
  
 UserDTO userDTO = new UserDTO();  
 userDTO.setId(userId);  
 userDTO.setName("John");  
 userDTO.setLastName("Doe");  
 userDTO.setDni("123456789");  
 userDTO.setPassword("password");  
 userDTO.setAddress("123 Main St");  
 userDTO.setEmail("john.doe@example.com");  
 Rol rolDTO = new Rol();  
 rolDTO.setId(1L);  
 rolDTO.setName("RoleName");  
 userDTO.setRol(rolDTO);  
  
 *when*(userService.readUser(userId)).thenReturn(userDTO);  
  
 // Act  
 ResponseEntity<?> response = userController.getUser(userId);  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
 *assertNotNull*(response.getBody());  
  
 UserDTO retrievedUser = (UserDTO) response.getBody();  
 *assertEquals*(userDTO.getName(), retrievedUser.getName());  
 *assertEquals*(userDTO.getLastName(), retrievedUser.getLastName());  
  
}

1. **getUser\_NonExistingUserId\_ReturnsBadRequest()**

Este test verifica el comportamiento del método "getUser" en la clase "UserController" y verifica que se arroje un error cuando se busca un usuario por medio de su ID que no existe.

@Test  
public void getUser\_NonExistingUserId\_Returns400BadRequest() {  
 // Arrange  
 Long userId = 1L;  
  
 *when*(userService.readUser(userId)).thenReturn(null);  
  
 // Act  
 ResponseEntity<?> response = userController.getUser(userId);  
  
 // Assert  
 *assertEquals*(HttpStatus.*BAD\_REQUEST*, response.getStatusCode());  
  
 String message = (String) response.getBody();  
 *assertEquals*("Message: The user with " + userId + " does not exist", message);  
  
}

1. **modifyUser\_ValidUserDTO\_ReturnsOK()**

Este test verifica el comportamiento del método "modifyUser" en la clase "UserController" y verifica que cuando se modifique un usuario este se guarde correctamente.

@Test  
public void modifyUser\_ValidUserDTO\_Returns200OK() {  
 // Arrange  
 UserDTO userDTO = new UserDTO();  
 userDTO.setId(1L);  
 userDTO.setName("John");  
 userDTO.setLastName("Doe");  
 userDTO.setDni("123456789");  
 userDTO.setPassword("password");  
 userDTO.setAddress("123 Main St");  
 userDTO.setEmail("john.doe@example.com");  
 Rol rolDTO = new Rol();  
 rolDTO.setId(1L);  
 rolDTO.setName("RoleName");  
 userDTO.setRol(rolDTO);  
  
 *when*(rolService.existById(rolDTO.getId())).thenReturn(true);  
 *when*(userService.modifyUser(userDTO)).thenReturn(userDTO);  
  
 // Act  
 ResponseEntity<?> response = userController.modifyUser(userDTO);  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
 *assertNotNull*(response.getBody());  
  
 UserDTO modifiedUser = (UserDTO) response.getBody();  
 *assertEquals*(userDTO.getName(), modifiedUser.getName());  
 *assertEquals*(userDTO.getLastName(), modifiedUser.getLastName());  
  
}

1. **modifyUser\_RolDoesNotExist\_ReturnsBadRequest()**

Este test verifica el comportamiento del método "modifyUser" en la clase "UserController" y verifica que si se modifica un usuario a un rol que no exista este arroje un error.

@Test  
public void modifyUser\_RolDoesNotExist\_ReturnsBadRequest() {  
 // Arrange  
 UserDTO userDTO = new UserDTO();  
 userDTO.setId(1L);  
 userDTO.setName("John");  
 userDTO.setLastName("Doe");  
 userDTO.setDni("123456789");  
 userDTO.setPassword("password");  
 userDTO.setAddress("123 Main St");  
 userDTO.setEmail("john.doe@example.com");  
 Rol rolDTO = new Rol();  
 rolDTO.setId(1L);  
 rolDTO.setName("RoleName");  
 userDTO.setRol(rolDTO);  
  
 *when*(rolService.existById(rolDTO.getId())).thenReturn(false);  
  
 // Act  
 ResponseEntity<?> response = userController.modifyUser(userDTO);  
  
 // Assert  
 *assertEquals*(HttpStatus.*BAD\_REQUEST*, response.getStatusCode());  
  
 String message = (String) response.getBody();  
 *assertEquals*("Message: The rol does not exist", message);  
  
}

1. **removeUser\_ExistingUserId\_ReturnsOK()**

Este test verifica el comportamiento del método "removeUser" en la clase "UserController" y comprueba que se elimine un usuario al buscarlo por el ID .

@Test  
public void removeUser\_ExistingUserId\_ReturnsOK() {  
 // Arrange  
 Long userId = 1L;  
  
 *when*(userService.removeUser(userId)).thenReturn(true);  
  
 // Act  
 ResponseEntity<?> response = userController.removeUser(userId);  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
 *assertNotNull*(response.getBody());  
  
 String message = (String) response.getBody();  
 *assertEquals*("Message: User was delete", message);  
  
}

1. **removeUser\_NonExistingUserId\_ReturnsBadRequest()**

Este test verifica el comportamiento del método "removeUser" en la clase "UserController" y prueba que arroje un error al intentar eliminar un usuario que no existe.

@Test  
public void removeUser\_NonExistingUserId\_ReturnsBadRequest() {  
 // Arrange  
 Long userId = 1L;  
  
 *when*(userService.removeUser(userId)).thenReturn(false);  
  
 // Act  
 ResponseEntity<?> response = userController.removeUser(userId);  
  
 // Assert  
 *assertEquals*(HttpStatus.*NOT\_FOUND*, response.getStatusCode());  
  
 String message = (String) response.getBody();  
 *assertEquals*("Message: The user " + userId + " does not exist", message);  
  
}

1. **getallUsers\_UsersExist\_ReturnsOK()**

Este test verifica el comportamiento del método "getallUsers" en la clase "UserController" y prueba que llame a todos los usuarios existentes.

@Test  
public void getallUsers\_UsersExist\_ReturnsOK() {  
 // Arrange  
 UserDTO userDTO = new UserDTO();  
 userDTO.setId(1L);  
 userDTO.setName("John");  
 userDTO.setLastName("Doe");  
 userDTO.setDni("123456789");  
 userDTO.setPassword("password");  
 userDTO.setAddress("123 Main St");  
 userDTO.setEmail("john.doe@example.com");  
 Rol rolDTO = new Rol();  
 rolDTO.setId(1L);  
 rolDTO.setName("RoleName");  
 userDTO.setRol(rolDTO);  
  
 Set<UserDTO> users = new HashSet<>(Collections.*singletonList*(userDTO));  
  
 *when*(userService.getAll()).thenReturn(users);  
  
 // Act  
 ResponseEntity<?> response = userController.getallUsers();  
  
 // Assert  
 *assertEquals*(HttpStatus.*OK*, response.getStatusCode());  
 *assertNotNull*(response.getBody());  
  
 Set<UserDTO> retrievedUsers = (Set<UserDTO>) response.getBody();  
 *assertEquals*(1, retrievedUsers.size());  
 UserDTO retrievedUser = retrievedUsers.iterator().next();  
 *assertEquals*(userDTO.getName(), retrievedUser.getName());  
 *assertEquals*(userDTO.getLastName(), retrievedUser.getLastName());  
  
}

1. **getallUsers\_NoUsersExist\_ReturnsBadRequest()**

Este test verifica el comportamiento del método "getallUsers" en la clase "UserController" y prueba lo que ocurre si no ha sido generado la coleccion de usuarios.

@Test  
public void getallUsers\_NoUsersExist\_ReturnsBadRequest() {  
 // Arrange  
 *when*(userService.getAll()).thenReturn(Collections.*emptySet*());  
  
 // Act  
 ResponseEntity<?> response = userController.getallUsers();  
  
 // Assert  
 *assertEquals*(HttpStatus.*BAD\_REQUEST*, response.getStatusCode());  
  
 String message = (String) response.getBody();  
 *assertEquals*("Message: There are not information", message);  
  
}

