

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Consorcio de comidas a domicilio

Documento de Pruebas Unitarias

Nombre del Curso:

Fundamentos de ingeniería de software

Nombre del Equipo de Trabajo:

Si pierdo entro en prueba académica

Nombre de los Integrantes:

David Alexander Molina Díaz - 20202020084
e-mail: damolinad@correo.udistrital.edu.co

Jose Miguel Londoño Blandón - 20202020064
e-mail: josmlondonob@correo.udistrital.edu.co

Violeth Valmont Azahar - 20181020010
e-mail: vvalmonta@correo.udistrital.edu.co

Juan Esteban Torres - 20192020052
e-mail: jetorresa@correo.udistrital.edu.co

Las pruebas unitarias se dividieron en dos, el primer grupo aborda pruebas sobre la conexión a la DB, registro de usuarios y autenticación de los mismos, el segundo prueba los métodos usados para el cálculo de variables internas a la hora de cada compra independientes de la DB asumiendo que los valores entrantes son datos de tipo <Array> después de una modificación luego de su petición a la DB.

Prueba Insertar un registro con una PK conocida y nueva #1	
Dependencia	BD - restaurantes - usuarios- facturación
Criterios de inicialización	<pre> @Test public void testSaveCustomer_success() { Mockito.when(userRepository.save(Mockito.any())).thenReturn(validUserObject()); Mockito.when(customerRepository.save(Mockito.any())).thenReturn(validCustomerObject()); Customer _customer = customerServiceImpl.save(validCustomerRegisterObject()); Assertions.assertEquals(_customer.getUser().getEmail(), validCustomerObject().getUser().getEmail()); } </pre> <p>Pk no existente findBy(email/restaurante/id) registrar</p>
Criterios de finalización	registro en la base de datos

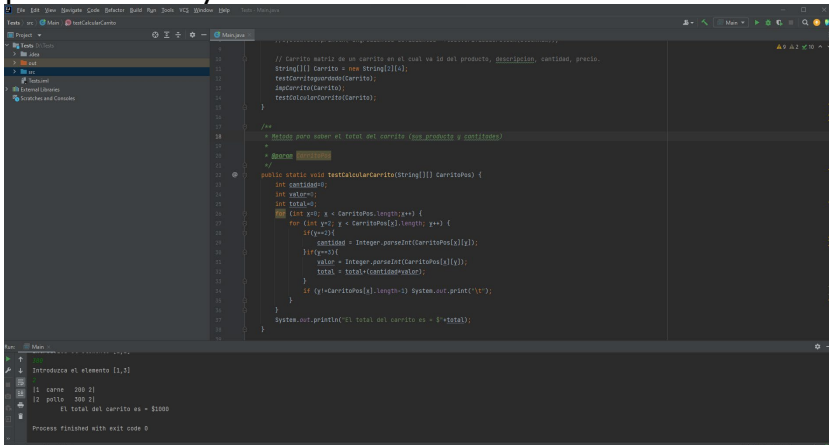
Prueba Insertar un segundo registro con la misma pk que el registro 1 #2	
Dependencia	BD - restaurantes - usuarios - facturación
Criterios de inicialización	<p>Pk preexistente findBy(email/restaurante/id)</p> <pre> @Test public void testSaveCustomer_success() { Mockito.when(userRepository.save(Mockito.any())).thenReturn(validUserObject()); Mockito.when(customerRepository.save(Mockito.any())).thenReturn(validCustomerObject()); Customer _customer = customerServiceImpl.save(validCustomerRegisterObject()); Assertions.assertEquals(_customer.getUser().getEmail(), validCustomerObject().getUser().getEmail()); } @Test(expected = DuplicateEntryException.class) public void testSaveAdministrator_duplicateEntry() { Mockito.when(administratorRepository.save(Mockito.any())).thenThrow(DataIntegrityViolationException.class); administratorServiceImpl.saveAdministrator(validAccountDTOObject()); } </pre>
Criterios de finalización	error en al ingresar los datos

Prueba Realizar la prueba y dar los resultados de la inserción realizada del primer y segundo registro #3	
---	--

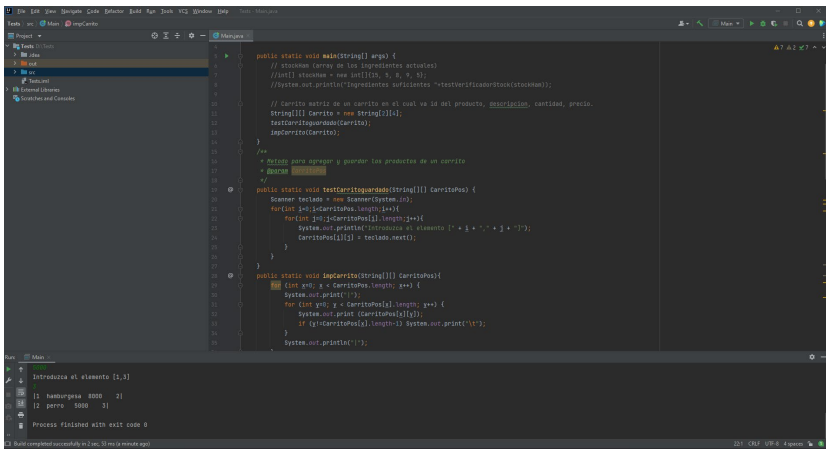
Dependencia	BD - restaurantes - ingredientes - facturación
Criterios de inicialización	Datos a consultar en la base de datos <pre>@Test public void testPlaceOrder_success() { Mockito.when(customerRepository.findById(Mockito.any())).thenReturn(Optional.of(validCustomerObject())); Mockito.when(restaurantRepository.findById(Mockito.any())).thenReturn(Optional.of(validRestaurantObject())); Mockito.when(pandaOrderRepository.save(Mockito.any())).thenReturn(validPandaOrderObject(OrderStatus.PENDING)); Mockito.when(stateRepository.findById(OrderStatus.PENDING)).thenReturn(State.builder().orderStatus(Mockito.when(foodRepository.findById(Mockito.any())).thenReturn(Optional.of(new Food())); Mockito.when(cartItemRepository.save(Mockito.any())).thenReturn(new CartItem()); CustomerServiceImpl customerService = Mockito.spy(customerServiceImpl); Mockito.doNothing().when(customerService).sendMail(Mockito.any(), Mockito.any(), Mockito.any());</pre>
Criterios de finalización	busqueda de registros, orden completada

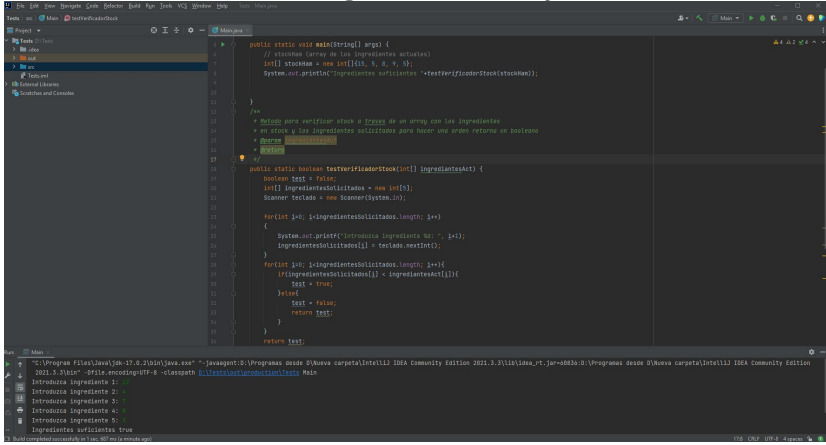
Prueba Autenticación		#4
Dependencia	BD - restaurantes - iusuarios - facturación	
Criterios de inicialización	Verificar por email roles y acceso a rutas <pre>@Test public void testAuthenticate_success() { Mockito.when(userRepository.findById(Mockito.any())).thenReturn(Optional.of(userObjectReturned())); Mockito.when(administratorRepository.findById(Mockito.any())).thenReturn(Optional.of(administratorObjectReturned())); Administrator _administrator = administratorServiceImpl.authenticate(validAccountDTOObject()); Assertions.assertEquals(_administrator.getUser().getEmail(), administratorObjectReturned().getUser().getEmail()); } @Test(expected = InvalidInputException.class) public void testAuthenticate_invalidInput() { administratorServiceImpl.authenticate(accountDTO: null); } @Test(expected = InvalidCredentialsException.class) public void testAuthenticate_notFound() { Mockito.when(userRepository.findById(Mockito.any())).thenReturn(Optional.empty()); administratorServiceImpl.authenticate(validAccountDTOObjectBadPassword()); } @Test(expected = InvalidInputException.class) public void testAuthenticate_invalidPassword() { Mockito.when(userRepository.findById(Mockito.any())).thenReturn(Optional.of(userObjectReturned())); administratorServiceImpl.authenticate(validAccountDTOObjectBadPassword()); } @Test(expected = InvalidInputException.class) public void testSaveAdministrator_invalidCredentials() { administratorServiceImpl.saveAdministrator(invalidAccountDTOObject()); } @Test(expected = DuplicateEntryException.class) public void testSaveAdministrator_duplicateEntry() { Mockito.when(administratorRepository.save(Mockito.any())).thenThrow(DataIntegrityViolationException.class); administratorServiceImpl.saveAdministrator(validAccountDTOObject()); } @Test public void testAuthenticate_success() { Mockito.when(userRepository.findById(Mockito.any())).thenReturn(Optional.of(validUserObject())); Mockito.when(customerRepository.findById(Mockito.any())).thenReturn(Optional.of(validCustomerObject())); Customer _customer = customerServiceImpl.authenticate(validAccountDTOObject()); Assertions.assertEquals(_customer.getUser().getEmail(), validCustomerObject().getUser().getEmail()); }</pre>	

	<pre> @Test(expected = InvalidInputException.class) public void testAuthenticate_invalidInput() { customerServiceImpl.authenticate(invalidAccountDTOObject()); } @Test(expected = RuntimeException.class) public void testAuthenticate_notFound() { Mockito.when(userRepository.findByEmail(Mockito.any())).thenReturn(Optional.empty()); customerServiceImpl.authenticate(validAccountDTOObject()); } @Test(expected = InvalidInputException.class) public void testAuthenticate_invalidPassword() { Mockito.when(userRepository.findByEmail(Mockito.any())).thenReturn(Optional.of(validUserObject())); customerServiceImpl.authenticate(validAccountDTOObjectBadPassword()); } </pre>
Criterios de finalización	busqueda de registros, verificación del error, devuelve el problema en una exception

Prueba Calculo total según artículos en carrito #5	
Dependencia	Services-Rotonda
Criterios de inicialización	<p>Ingresa un Array con los datos del artículo (nombre, cantidad, valor) suma sus valores totales por artículo y da un valor total a la cuenta</p> 
Criterios de finalización	Devuelve el total de los artículos según la cantidad seleccionada por cada uno de ellos

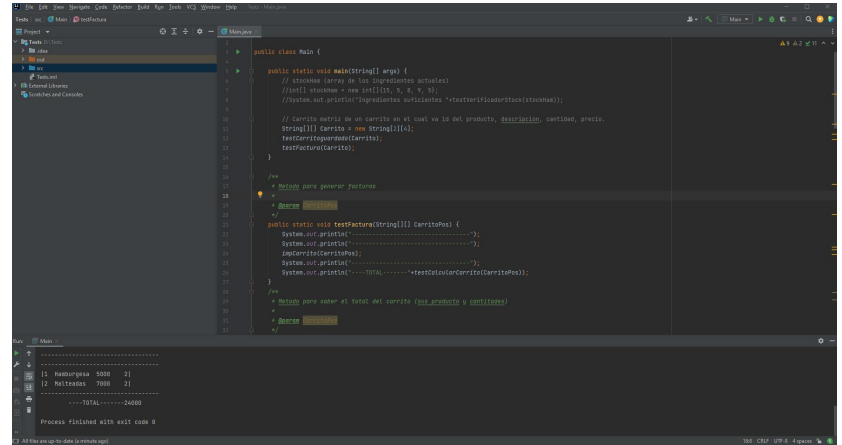
Prueba Carrito de compras guardado #6	
Dependencia	Services-Rotonda
Criterios de inicialización	Ingresa los datos que le son suministrados como array en una matriz

	
Criterios de finalización	Devuelve un array con los productos agregados al carrito

Prueba Verificar cantidad de ingredientes en stock	#7
Dependencia	Services-Rotonda
Criterios de inicialización	<p>Compara dos array en los que la posicion hace referencia al mismo ingrediente a consultar, si todos los ingredientes estan disponibles (su numero es menor o igual al stock) devuelve true</p> 
Criterios de finalización	Booleano con resultado true o false para la identificación de una falta de ingredientes.

Prueba Generar factura	#8
Dependencia	Services-Rotonda
Criterios de inicialización	<p>Ingresan articulos con descripcion, valor y cantidad por cada uno de los productos del carrito, imprime la factura como matriz y como texto en</p>

consola



The screenshot shows an IDE with a Java file named `Main.java`. The code defines a `Main` class with a `main` method and a `testFactura` method. The `main` method initializes an array of `Producto` objects and calls `testFactura`. The `testFactura` method prints the details of each product and the total amount. The output window shows the execution results, displaying the product details and the total amount of 24000.

```
public class Main {  
    // productos (array de los ingredientes actuales)  
    // tipo[] productos = new tipo[10]; // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10;  
    // System.out.println("Ingredientes actuales: " + Arrays.toString(productos));  
  
    // Carrito matriz de un carrito en el cual va la del producto, descripcion, cantidad, precio.  
    String[][] carrito = new String[1][0];  
    testCarritoIngrediente(carrito);  
    testFactura(carrito);  
}  
  
//  
// Metodo para generar facturas  
//  
// Metodo testFactura  
//  
public static void testFactura(String[][] carritoPlus) {  
    System.out.println("-----");  
    System.out.println("Factura de productos:");  
    for (String[] carritoPlus : carritoPlus) {  
        System.out.println("Producto: " + carritoPlus[0] + " Cantidad: " + carritoPlus[1] + " Precio: " + carritoPlus[2]);  
    }  
    System.out.println("-----");  
    System.out.println("TOTAL: " + testCalculaCarrito(carritoPlus));  
}  
  
//  
// Metodo para saber el total del carrito (por producto y cantidad)  
//  
// Metodo testCalculaCarrito  
//  
}
```

Run: Main

[0] Hamburguesa 5000 2]
[1] Helados 7000 2]

TOTAL: 24000

Process finished with exit code 0

Criterios de finalización

Array factura descripcion valor y cantidad