Unidades de pruebas

Unidades de pruebas **1**
- ¿Qué aprenderás? 2
- Introducción 2
- Pruebas con la librería JUnit 2
- Assert en JUnit 4
- Implementación JUnit a la clase MensajeServiceImpl 6

¡Comencemos!

¿Qué aprenderás?

- Entender la importancia de las unidades de pruebas.
- Aprender a utilizar la librería JUnit para hacer test unitarios.
- Aprender a utilizar Mockito para realizar test integrados.
- Definir e implementar los test unitarios en el desarrollo de la aplicación.

Introducción

Las unidades de pruebas, son una parte fundamental e importante en el desarrollo de cualquier aplicación, ya que servirán como documentación e iremos comprobando el código de nuestra aplicación a medida que la vamos desarrollando. Esto nos permite, al final del desarrollo, llegar con un software ya depurado y con una tasa muy baja de errores.

En este capítulo, se enseñará a crear unidades de prueba mediante la librería JUnit 4, la cual viene integrada dentro del conjunto de librerías de Spring Boot.

¡Vamos con todo!

Pruebas con la librería JUnit

Un test con JUnit, se debe pensar y crear en base a dos principales preguntas que debemos hacernos antes de crear el test:

1. ¿Qué se debe hacer antes de que se ejecute el test?

Para esto, se debe implementar la anotación de JUnit @Before, el cual es un método único dentro de la clase test en cuestión, en donde se coloca el código que se ejecutará antes de los test propiamente tal. Por ejemplo, este método nos sirve para:

- Instanciar el componente a probar.
- Hacer set a métodos del componente con algún valor que necesitemos probar.
- Declaraciones varias necesarias para los test.

2. ¿Cuáles son los test o pruebas relevantes que debemos hacer a nuestro componente?

En este caso, debemos usar la anotación @Test al inicio de cada prueba. En esta sección, podrán codificar todos los test necesarios para verificar el componente de manera correcta, por lo que necesitamos tener claridad de cuáles son las pruebas unitarias que deseemos y necesitemos certificar. Lo que se debe hacer, es escribir las pruebas unitarias antes de codificarlas, por ejemplo de la siguiente manera:

Caso 1:

El componente, al ser declarado, todos sus atributos deben tener el valor null.

Caso 2:

Al ejecutar el método setValor del Componente con un parámetro null, debería arrojar un error.

La estructura básica de un test unitario con JUnit, es la siguiente:

```
public class NombreClaseTest {
   @Before
   public void setUp() throws Exception {
      /*Código que se ejecutará antes de los test*/
   }
   @Test
   public void test1() {
      /* Código de prueba para el caso 1 */
   }
   @Test
   public void test2() {
      /* Código de prueba para el caso 2 */
   }
   @Test
   public void testn() {
      /* Código de prueba para el caso n */
   }
}
```

Como se puede apreciar, el ejemplo anterior nos muestra que existe solo un método @Before, no obstante existen tantos métodos @Test como casos necesitemos probar.

El método setUp del ejemplo anterior, se ejecutará siempre antes de cada test, es decir, en el ejemplo anterior se ejecuta setUp luego test1, luego setUp y luego test2 y así sucesivamente.

Assert en JUnit

Los assert en JUnit, sirven principalmente para realizar comparaciones en base a lo que se espera obtener de una prueba en particular, existen varios tipos de assert, de los cuales lo que más se destacan y utilizan son los siguientes:

- assertEquals, nos sirve para comparar un valor esperado en base a un resultado. Por ejemplo:

```
@Test
   public void resultadoEsCero() {
      assertEquals(0, 1-1);
   }
```

En el ejemplo anterior, el primer parámetro especifica lo esperado y el segundo parámetro especifica lo que se quiere probar.

- assertNotEquals, nos sirve para evaluar si un valor esperado no es igual al resultado. Por ejemplo:

```
@Test
   public void resultadoNoEsCero() {
      assertNotEquals(0, 1-2);
   }
```

- assertTrue, permite evaluar si una expresión es igual a verdadero. Por ejemplo:

```
@Test
   public void resultadoEsVerdadero() {
      assertTrue(1<2);
   }
```

- assertNull, sirve para evaluar si un objeto o algo en el código es null. Por ejemplo, si creamos una variable String pero no la hemos inicializado con null:

```
@Test
   public void resultadoEsNull() {
      assertNull(nombre);
   }
```

JUnit, es muy potente a la hora de realizar los test e implementa muchas y variadas herramientas que nos ayudan a generar y entender de mejor manera lo que queremos comprobar en un caso de prueba.

Cuando los test o casos de prueba son exitoso, JUnit avisa con una barra de progreso en color verde, como se muestra en la siguiente imagen:

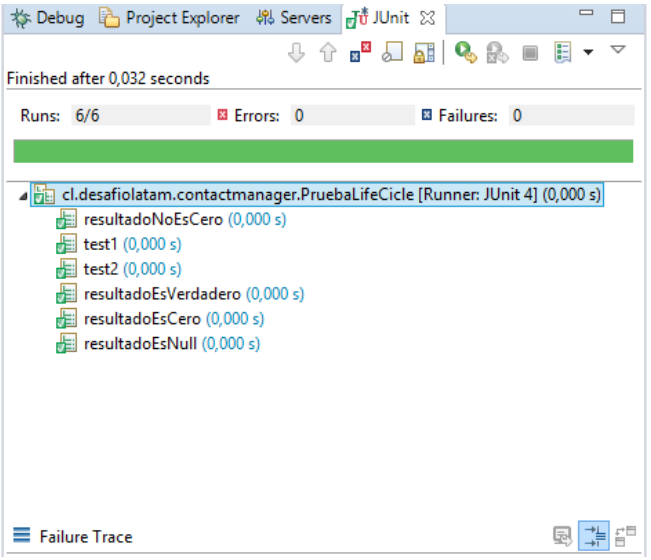

Imagen 1. Casos de prueba.
Fuente: Desafío Latam.

En la imagen 1, se muestra que todos los casos a probar terminaron de forma correcta. El resultado, en la parte superior nos indica el total de casos, cuántos fueron ejecutados, cuantos errores y cuantos casos exitosos se obtuvieron al ejecutar el test.

La idea, es que los casos de prueba resulten todos exitosos, de tal manera que se certifique el correcto funcionamiento del componente, en base a cómo necesitamos que este funcione. Cuando un caso resulta sin éxito, JUnit nos avisará con una barra de progreso en color rojo.

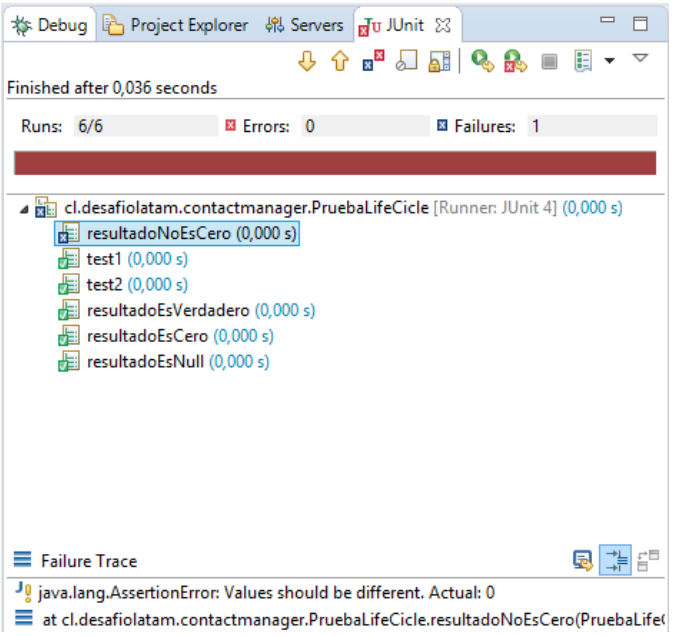

Imagen 2. Casos ejecutados, aprobados y fallados.
Fuente: Desafío Latam.

La imagen nos muestra que de 6 casos ejecutados, falló la comprobación de que si el resultado es igual a cero y hubieron 0 errores al ejecutar JUnit, estos aparecen cuando la ejecución del código da error y no la validación ingresada.

El test que se ejecutó para el resultado de la imagen anterior, fue el siguiente:

```
@Test
   public void resultadoNoEsCero() {
      assertNotEquals(0, 1-1);
   }
```

En este test, se está evaluando si 1-1 no es igual a cero, lo cual falla ya que esperamos que el resultado 1-1 tenga un valor distinto de cero, lo cual no es correcto.

Implementación JUnit a la clase MensajeServiceImpl

En el ejemplo siguiente, vamos a realizar las pruebas asociadas a nuestro componente MensajeServiceImpl.java de nuestro proyecto holamundospringmvc.

1. Antes que todo, debemos definir nuestros casos a probar:

Caso 1:

- **Descripción:** Probar que el método getDataMessageList en primera instancia, devuelve la instancia de una lista de Mensajes.
- **Método:** getDataMessageList().
- **Resultado esperado:** new ArrayList<Mensaje>().

Caso 2:

- **Descripción:** Probar que el método getDataMessageList en primera instancia, devuelva una lista ArrayList Vacía.
- **Método:** getDataMessageList().

.

- **Resultado esperado:** getDataMessageList().size = 0.

Caso 3:

- **Descripción:** Agregar un nuevo mensaje.
- **Método:** saveDataMessage() - getDataMessageList().
- **Resultado esperado:** Al ejecutar saveDataMessage y luego getDataMessageList, el resultado de este último debe ser el mismo objeto mensaje agregado.

2. Como ya tenemos definidos los casos a probar, debemos crear en cl.desafiolatam.holamundospringmvc la clase MensajeServiceImplTest.java. Siempre, las clases de test, por convención, se deben crear con el nombre de la clase a probar finalizando con la palabra Test.

3. Implementar el siguiente código en la clase test creada:

```
package cl.desafiolatam.holamundospringmvc;
import static org.junit.Assert.assertEquals;
import java.util.ArrayList;
import org.junit.Before;
import org.junit.Test;
import cl.desafiolatam.holamundospringmvc.model.Mensaje;
import cl.desafiolatam.holamundospringmvc.service.MensajeService;
import
cl.desafiolatam.holamundospringmvc.service.impl.MensajeServiceImpl;
public class MensajeServiceImplTest {
```

```
   private Mensaje mensaje;
   private MensajeService mensajeService;
   @Before
   public void setUp() throws Exception {
      System.out.println("setUp");
      mensaje = new Mensaje();
      mensajeService = new MensajeServiceImpl(mensaje);
   }
```

Caso 1:

- **Descripción:** Probar que el método getDataMessageList en primera instancia, devuelve la instancia de una lista de Mensajes.
- **Método:** getDataMessageList().
- **Resultado esperado:** new ArrayList<Mensaje>().

```
   @Test
   public void caso1_obtener_lista_mensajes() {
      assertEquals(new ArrayList<Mensaje>(),
mensajeService.getDataMessageList());
   }
```

Caso 2:

- **Descripción:** Probar que el método getDataMessageList en primera instancia, devuelva una lista ArrayList Vacía.
- **Método:** getDataMessageList().
- **Resultado esperado:** getDataMessageList().size = 0.

```
   @Test
   public void caso2_obtener_lista_contacto() {
      assertEquals(0, mensajeService.getDataMessageList().size());
   }
```

Caso 3:

- **Descripción:** Agregar un nuevo mensaje.
- **Método:** saveDataMessage() - getDataMessageList().
- **Resultado esperado:** Al ejecutar saveDataMessage y luego getDataMessageList, el resultado de este último debe ser el mismo objeto mensaje agregado

```
    @Test
    public void caso3_agregar_mensaje() {
        mensaje.setRemitente("Pablito");
        mensaje.setMensaje("clavo un clavito");
        mensajeService.saveDataMessage(mensaje);
        assertEquals(mensaje, mensajeService.getDataMessageList().get(0));
    }
}
```

4. Comprobar el test.

a. Ir a la clase creada, hacer click derecho sobre la clase, Run As -> click en 2 JUnit Test.

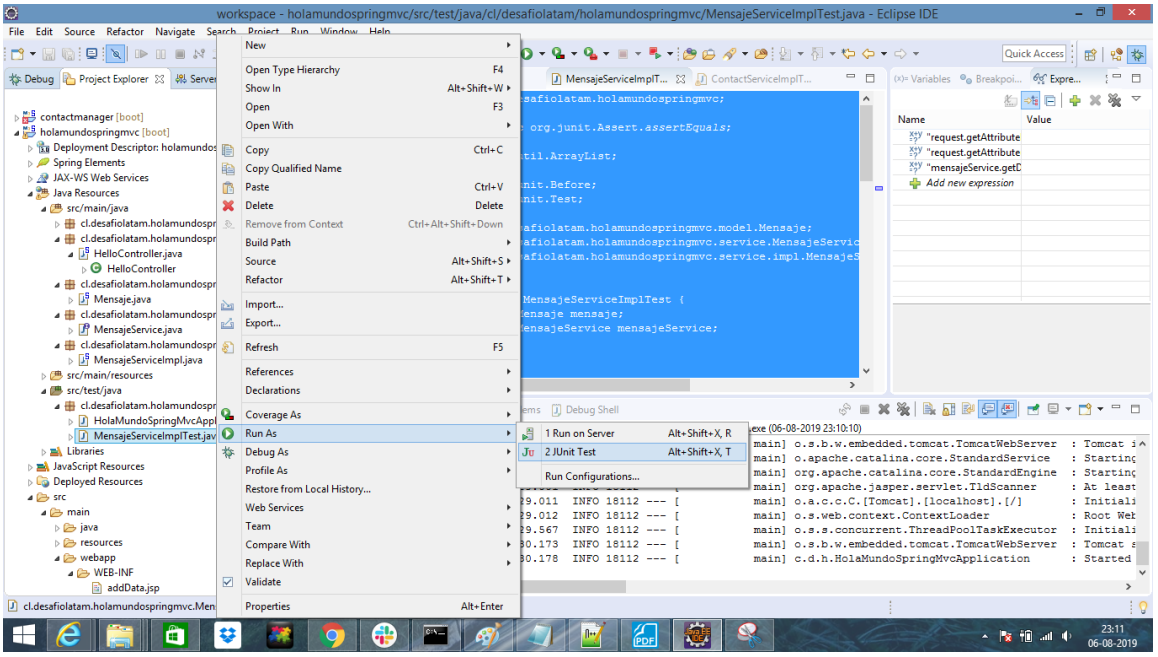

Imagen 3. Ejecutar casos de prueba.
Fuente: Desafío Latam.

b. Revisar los resultados.

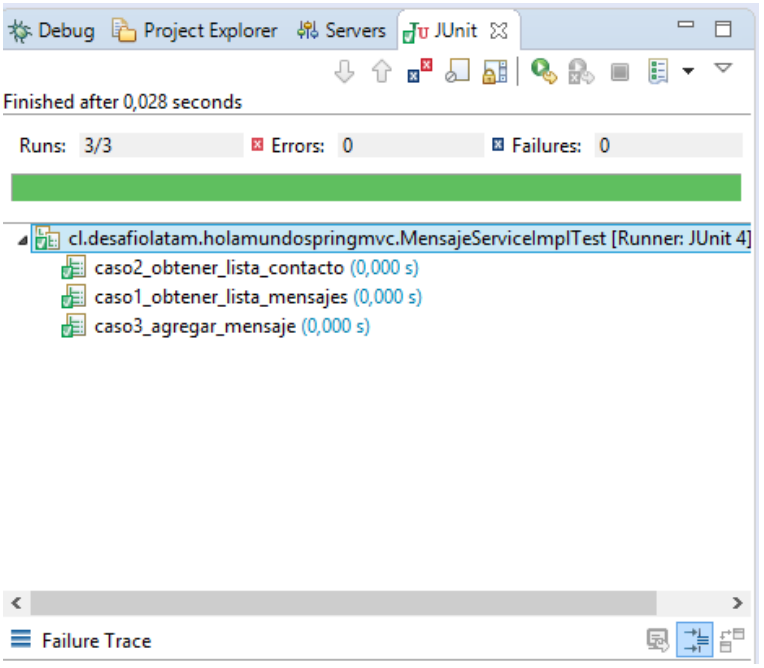

Imagen 4. Revisando los resultados.
Fuente: Desafío Latam.