





Geeks México

BLOG DE PROGRAMACIÓN EN ESPAÑOL SOBRE JAVA,
FRAMEWORKS, BASES DE DATOS, CÓMPUTO EN LA NUBE, ETC.
EN ESPAÑOL Y EN INGLÉS.

[HOME](#)[ABOUT](#)[CONTACT](#)

Anuncios

 Grupo Santander	Hazte Cliente	<div>1/6 </div> <div>Este número es indicativo del riesgo del producto, siendo 1/6 indicativo del menor riesgo y 6/6 del mayor riesgo.</div> <div>Entidad adherida al Fondo de Garantía de Depósitos de Entidades de Crédito Español. Para depósitos en dinero, el importe garantizado es de 100.000€ por depositante en cada entidad de crédito.</div>
--	-------------------------------	---

[Report this ad](#)

Pruebas unitarias parte 2:

JUnit y Mockito primeros pasos

📅 [HACE 4 DÍAS](#) 💬 [DEJA UN COMENTARIO](#)

1 Vote

Cuando hablamos de **unit tests** debemos hablar de “**Mocking**”, este es uno de los skills principales que debemos tener a la hora de hacer tests en nuestras aplicaciones.

Al momento de desarrollar aplicaciones las dividimos en capas, web, negocio y datos, entonces al escribir tests unitarios en una capa, no queremos preocuparnos por otra, así que la simulamos, a este proceso se le conoce como Mocking.

Test unitarios

Un test unitario debe probar un componente aislado de nuestra aplicación, los errores o efectos secundarios de otros componentes deben ser eliminados. La pregunta normal sería ¿Cómo aislar mi componente de sus dependencias para así probarlo? la respuesta es a través del uso de “**test doubles**”, estos se clasifican del siguiente modo:

- Dummy objects : Este tipo de objetos son asignados al componente pero nunca se llaman y por lo general están vacíos.

- Fake objects: Este tipo de objetos tienen implementaciones funcionales pero simplificadas, normalmente utilizan datos que no vienen de la base de datos principal, sino que se tienen en cache u otra fuente más simple.
- Stub classes : Es una implementación parcial de una interfaz o clase con el propósito de utilizarla durante el test, normalmente solo cuentan con los métodos implementados que serán utilizados durante el test.
- Mock objects: Es una implementación Dummy de una interfaz o clase en la cual se define la salida que tendrá la llamada de un método.

Estos “**test doubles**” aseguran que las pruebas se ejecuten solo sobre el componente que deseamos probar asegurando que no se tendrá ningún efecto secundario de ninguna otra clase.

Uso de Mocks

Es posible crear mocks manualmente, pero ya existen frameworks que pueden hacerlo por nosotros, estos permiten crear objetos mock en tiempo de ejecución y definir su comportamiento.

El ejemplo clásico de un objeto mock es un proveedor de datos, cuando se ejecuta la aplicación el componente se conectará a una base de datos y proveerá datos reales, pero cuando se ejecuta un test unitario lo que buscamos es aislarlo y para esto necesitamos un objeto mock que simulará la fuente de datos, esto asegurará que las condiciones de prueba sean siempre las mismas.

Un punto importante que se debe considerar es diseñar nuestros componentes para que sean probados fácilmente, si quieres saber más sobre esto te

recomendamos el post [Pruebas Unitarias Parte 1: ¿Cómo escribir código "Testeable"? \(https://geeks-mexico.com/2018/03/15/como-escribir-codigo-codigo-testeable/\)](https://geeks-mexico.com/2018/03/15/como-escribir-codigo-codigo-testeable/).

Caso práctico con Mockito

Una vez que entendemos los conceptos básicos el siguiente paso es crear un ejemplo práctico, para hacerlo utilizaremos el framework de Mockito.

Configuración

Para incluir el soporte de Mockito en nuestra aplicación es necesario definir las siguientes dependencias:

[Maven dependencies \(https://github.com/raidentrance/mockito-example/blob/master/pom.xml\)](https://github.com/raidentrance/mockito-example/blob/master/pom.xml)

Como se puede ver se incluye lo siguiente al proyecto:

- Dependencia de Mockito : Brinda soporte para la creación de mocks
- Dependencia de JUnit : Brinda soporte para la creación de tests unitarios
- Maven compiler plugin : Define la versión de Java a utilizar

Ejemplo a realizar

En el ejemplo a realizar se crearán dos interfaces CalculatorService y DataService, con los siguientes propósitos:

- DataService : Contendrá el método `int[] getListOfNumbers()`; que devolverá una lista de números de una fuente de datos.

- CalculatorService : Contendrá el método `double calculateAverage()`; que calculará el promedio de la lista de números devuelta por el servicio DataService.

A demás se creará un test unitario que ayudará a probar el CalculatorService.

DataService.java

```
1  /**
2   * @author raidentrance
3   *
4   */
5  public interface DataService {
6      int[] getListOfNumbers();
7  }
```

CalculatorService.java

```
1  /**
2   * @author raidentrance
3   *
4   */
5  public interface CalculatorService {
6      double calculateAverage();
7  }
```

Una vez definidas las interfaces veamos la implementación de CalculatorService.

CalculatorServiceImpl.java

```
1  import com.raidentrance.services.CalculatorService;
2  import com.raidentrance.services.DataService;
3
4  /**
5   * @author raidentrance
6   *
7   */
8  public class CalculatorServiceImpl implements CalculatorService {
9      private DataService dataService;
10
11      @Override
12      public double calculateAverage() {
13          int[] numbers = dataService.getListOfNumbers();
14          double avg = 0;
15          for (int i : numbers) {
16              avg += i;
17          }
18          return (numbers.length > 0) ? avg : 0;
19      }
20  }
```

```
21     public void setDataService(DataService
22         this.dataService = dataService;
23     }
24 }
```

En este caso no escribiremos implementación del DataService ya que haremos un mock del mismo para nuestras pruebas.

Probando nuestro componente

Una vez escritas las implementaciones de nuestros servicios el siguiente punto es probarlas, para esto crearemos un test unitario junto con mocks que permitan simular diferentes respuestas basadas en los tests, veamos el test de ejemplo:

```
1  import static org.junit.Assert.assertEquals;
2  import static org.mockito.Mockito.when;
3
4  import org.junit.Test;
5  import org.junit.runner.RunWith;
6  import org.mockito.InjectMocks;
7  import org.mockito.Mock;
8  import org.mockito.junit.MockitoJUnitRunner;
9
10 import com.raidentrance.services.DataService;
11 import com.raidentrance.services.impl.CalculatorService;
12
13 /**
14  * @author raidentrance
15  */
16
17 @RunWith(MockitoJUnitRunner.class)
18 public class CalculatorServiceTest {
19     @InjectMocks
20     private CalculatorServiceImpl calculatorService;
21
22     @Mock
23     private DataService dataService;
24
25     @Test
26     public void testCalculateAvg_simpleInput() {
27         when(dataService.getListOfNumbers())
28             .thenReturn(Arrays.asList(1, 2, 3, 4, 5));
29         assertEquals(3.0, calculatorService.calculateAvg());
30     }
31
32     @Test
33     public void testCalculateAvg_emptyInput() {
34         when(dataService.getListOfNumbers())
35             .thenReturn(new ArrayList());
36         assertEquals(0.0, calculatorService.calculateAvg());
37     }
38 }
```

```
37     @Test
38     public void testCalculateAvg_singleInp
39         when(dataService.getListOfNumbers(
40             assertEquals(1.0, calculatorServic
41         }
42     }
```

Del código anterior podemos analizar lo siguiente:

- **@RunWith(MockitoJUnitRunner.class)** : Se utiliza para definir que se utilizará el Runner de Mockito para ejecutar nuestras pruebas.
- **@Mock** : Se utiliza para informar a Mockito que un objeto mock será inyectado en la referencia `dataService`, como se puede ver no es necesario escribir la clase implementación, Mockito lo hace por nosotros.
- **@InjectMocks** : Se utiliza para informar a Mockito que los mocks serán inyectados en el servicio definido, es necesario que se cuente con métodos setters en los que se coloca.
- **when(dataService.getListOfNumbers()).thenReturn(new int[] { 1, 2, 3, 4, 5 })** : Define que cuando se ejecute el método `getListOfNumbers` se devolverá el resultado `new int[] { 1, 2, 3, 4, 5 }`.

Con lo anterior seremos capaces de probar nuestros componentes de forma aislada asegurando que las condiciones de nuestros tests siempre serán las mismas.

Síguenos en nuestras redes sociales https://twitter.com/geeks_mx (https://twitter.com/geeks_mx) y <https://www.facebook.com/geeksJavaMexico/> (<https://www.facebook.com/geeksJavaMexico/>) para recibir los siguientes posts que hablarán de temas más avanzados sobre Testing, JUnit y Mockito.

Puedes encontrar el código completo en el siguiente enlace <https://github.com/raidentrance/mockito-example> (<https://github.com/raidentrance/mockito-example>).

Autor: Alejandro Agapito Bautista

Twitter: @raidentrance

(<https://geeksjavamexico.wordpress.com/mentions/raidentrance/>)

Contacto:raidentrance@gmail.com

Anuncios



Report this ad



Report this ad

