FRAMEWORK

Introducción

Módulo VI

harriague+asociados

Emilio Rey | María Agustina Aliaga

INCLUIT
empowering your business

Para que testear?

Problema

1. Asegurarnos que lo que estamos programando cumple con la funcionalidad esperada
2. Mantener la integridad de nuestra aplicación a través del tiempo
3. Ejecutar repetidamente las pruebas

Solución

Las librerias más populares en Java son **JUnit** y **TestNG**. JUnit 4 incorpora varias mejoras importantes sobre JUnit 3, que se basa en la clase base (es decir, TestCase) y la firma del método (es decir, métodos cuyos nombres comienzan con la palabra test) para identificar casos de prueba, un enfoque que carece de flexibilidad.

JUnit 4 le permite anotar sus métodos de prueba con la anotación JUnit **@Test**, por lo que un método público arbitrario se puede ejecutar como un caso de prueba.

TestNG es otro poderoso marco de prueba que hace uso de anotaciones. También proporciona un tipo de anotación **@Test** para que usted identificar casos de prueba.

Creación de pruebas unitarias y de Integración

Problema

Una técnica de prueba común es probar cada módulo de su aplicación de forma aislada y luego probarlos en combinación. Desea aplicar esta habilidad para probar sus aplicaciones Java.

Solución

Las pruebas unitarias se usan para probar una sola unidad de programación. En los lenguajes orientados a objetos, una unidad suele ser una clase o un método. El alcance de una prueba unitaria es una sola unidad, pero en el mundo real, la mayoría de las unidades no funcionarán de forma aislada. Ellos a menudo necesita cooperar con otros para completar sus tareas. Al probar una unidad que depende de otras unidades, una técnica común que puede aplicar es simular las dependencias de la unidad con partes de objetos simulados, los cuales pueden reducir la complejidad de las pruebas unitarias causadas por dependencias.

Un **stub** es un objeto que simula un objeto dependiente con la cantidad mínima de métodos necesarios para una prueba.

Los métodos se implementan de una manera predeterminada, generalmente con datos harcodeados. Un **Stub** también expone métodos para una prueba para verificar los estados internos del **stub**. A diferencia de un **stub**, un **objeto mockeado** falso generalmente sabe cómo son sus métodos espera que se llame en una prueba. El objeto simulado luego verifica los métodos realmente llamados en contra de los esperados.

Creación de pruebas unitarias y de Integración

En Java, hay varias librerías que pueden ayudar a crear objetos falsos, incluidos EasyMock y jMock.
La principal diferencia entre un **stub** y un **objeto mockeado** es que un **stub** se usa generalmente para la **verificación de estado**, mientras que un objeto simulado es utilizado para la **verificación del comportamiento**.
Las **pruebas de integración**, en contraste, se usan para **probar varias unidades en combinación como un todo**. Ellos prueban si la integración y la interacción entre las unidades es correcta. Cada una de estas unidades ya debería haber sido probada con pruebas unitarias, por lo que la prueba de integración generalmente se realiza después de la prueba unitaria.
Finalmente, tenga en cuenta que las aplicaciones desarrolladas utilizan el principio de "separar la interfaz de la implementación" y el patrón de inyección de dependencia es fácil de probar, tanto para pruebas unitarias como para pruebas de integración.
Esto es porque el principio y el patrón pueden reducir el acoplamiento entre las diferentes unidades de su aplicación.

Inyección de Test Fixtures dentro de tests de integración

Problema
Los test fixtures dentro de un test de integración para una aplicación de Spring son en su mayoría beans declarados en el contexto de la aplicación.
Es posible que desee que los dispositivos de prueba sean inyectados automáticamente por Spring a través de la inyección de dependencia, lo que le ahorra el problema de recuperarlos del contexto de la aplicación de forma manual.

Solución
Spring Testing pueden inyectar beans automáticamente desde el administrador de contexto de la aplicación en sus tests.

Inyección de Test Fixtures dentro de tests de integración

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = BankConfiguration.class)
public class AccountServiceJUnit4ContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

  @Autowired
  private AccountService accountService;

  @Before
   public void init() {
      accountService.createAccount(TEST_ACCOUNT_NO);
      accountService.deposit(TEST_ACCOUNT_NO, 100);
  }
}
```

Inyección de Test Fixtures dentro de tests de integración

Problema
Al crear tests de integración para una aplicación que accede a una base de datos, generalmente se preparan los datos de prueba en el método de inicialización. Después de ejecutar cada método de prueba, puede haber modificado los datos en la base de datos. Entonces, tienes que limpiar la base de datos para asegurarse de que el próximo método de test se ejecute desde un estado consistente. Como resultado, tienes que desarrollar muchas tareas de limpieza de base de datos.

Solución
Las configuraciones de test de prueba de Spring pueden crear y revertir una transacción para cada método de prueba, por lo que los cambios que realice en un método de prueba no afectará al siguiente. Esto también puede ahorrarle el problema de desarrollar tareas de limpieza para limpiar la base de datos.
A partir de Spring 2.5, el marco TestContext proporciona un oyente de ejecución de prueba relacionado con la transacción administración. Se registrará con un administrador de contexto de prueba por defecto si no especifica el suyo explícitamente.
• **TransactionalTestExecutionListener:** maneja la anotación @Transactional a nivel de clase o método y tiene los métodos a ejecutarse dentro de las transacciones automáticamente.

Inyección de Test Fixtures dentro de tests de integración

Su clase de prueba puede extender la clase de soporte TestContext correspondiente para su marco de prueba, como se muestra en la Tabla de abajo, para que sus métodos de prueba se ejecuten dentro de las transacciones. Estas clases se integran con un administrador de contexto de prueba y tiene @Transactional habilitado en el nivel de clase. Tenga en cuenta que también se requiere un administrador de transacciones en el bean archivo de configuración.

Testing Framework	TestContext Support Class*
JUnit 4	AbstractTransactionalJUnit4SpringContextTests
TestNG	AbstractTransactionalTestNGSpringContextTests