

Tests unitaires avec JUNIT sur une application Web réalisée avec Spring .

Auteur : Beuve Johnny.

Table des matières

Remerciements.....	2
Introduction.....	2
Framework Spring.....	2
JUnit.....	2
Pré- requis.....	2
Rappel de l'architecture de l'application.....	3
Test unitaire avec JUNIT.....	4
Exemple simple :	4
La classe à tester :	4
La classe qui teste :	4
Test positif :	5
Test négatif :	5
Exemple avec Spring :	6
Classe qui teste le serviceTodo :	7
Bonnes pratiques :	8
Conclusion.....	8

Remerciements

Merci à Christophe Jollivet pour ses précieux conseils. Je vous invite à voir ses articles :
<http://christophej.developpez.com/>

Introduction

Dans cet article, on montrera, comment faire des tests unitaires d'une application Web réalisée avec le framework Spring, JSF et Hibernate. Les tests porteront essentiellement sur les services, mais il est aussi possible de tester toutes les classes.

Framework Spring

Le framework Spring est un conteneur léger, simple, non intrusif, extensible qui permet un réel découplage des couches grâce à son approche basée sur les notions d'inversion de contrôle et de programmation par aspect.

Pour en savoir plus, il est vivement recommandé de lire le document écrit par Serge Tahé expliquant l'inversion de contrôle avec **SPRING** (<http://tahe.developpez.com/java/springioc/>).

Pour la description de tous les services offerts par le framework Spring, veuillez consulter le site de référence :<http://www.springframework.org/>

Junit

Junit est un framework qui permet de créer des tests unitaires.

<http://junit.sourceforge.net/>

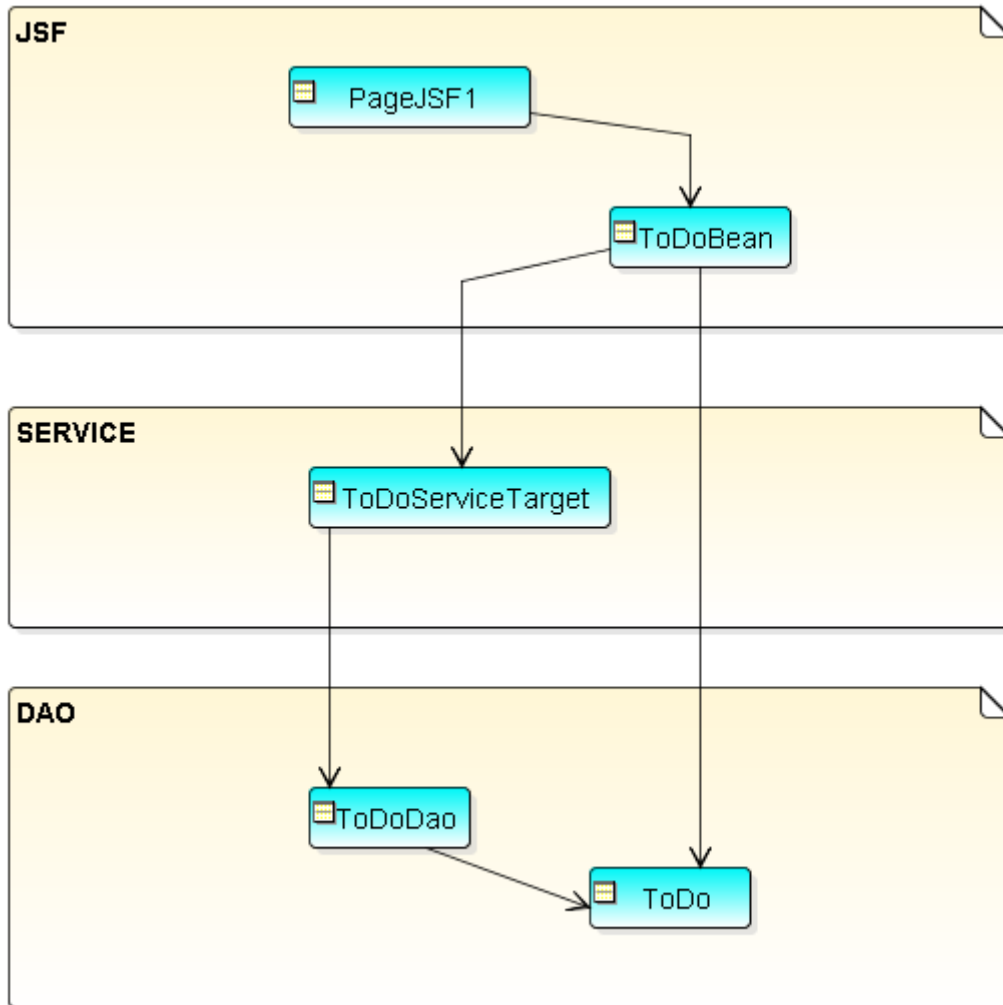
<http://www.junit.org/>

Pré- requis

Cet exemple s'appuie sur une application Web déjà existante. Un premier article « **Exemple CRUD avec le Framework Spring / JSF / Hibernate sur HSQL.** » explique l'application elle même et les pré-requis nécessaires pour faire tourner l'application.

Rappel de l'architecture de l'application.

Pour rappel l'architecture générale :



Nous testerons uniquement la partie service.

Test unitaire avec JUNIT

Exemple simple :

La classe à tester :

```
/**
 * Class with two attributs int a,b. <br>Addition and menux.
 */
public class ClassToTest {
    private int a;
    private int b;

    /**
     * Default constructor
     */
    public ClassToTest() {
        super();
        a = b = 0;
    }

    /**
     * Constructor
     */
    public ClassToTest(int a, int b) {
        super();
        this.a = a;
        this.b = b;
    }

    public int add() {
        return a + b;
    }

    public int menus() {
        return a - b;
    }
}
```

La classe qui teste :

```
/**
 */
package test;

import junit.framework.TestCase;

/**
 */
public class TestClassToTest extends TestCase {

    private ClassToTest test;

    /**
     * @see TestCase#setUp()
     */
    protected void setUp() throws Exception {
        super.setUp();
        test = new ClassToTest(3, 4);
    }

    public void testAdd() {
        assertEquals(7, test.add());
        // assertEquals("Ici le test est volontairement faux pour la démo", 8, test.add());
    }

    public void testMenus() {
        assertEquals(-1, test.menus());
    }
}
```

On hérite de TestCase.

Dans la méthode `setUp()` on initialise le contexte de test.

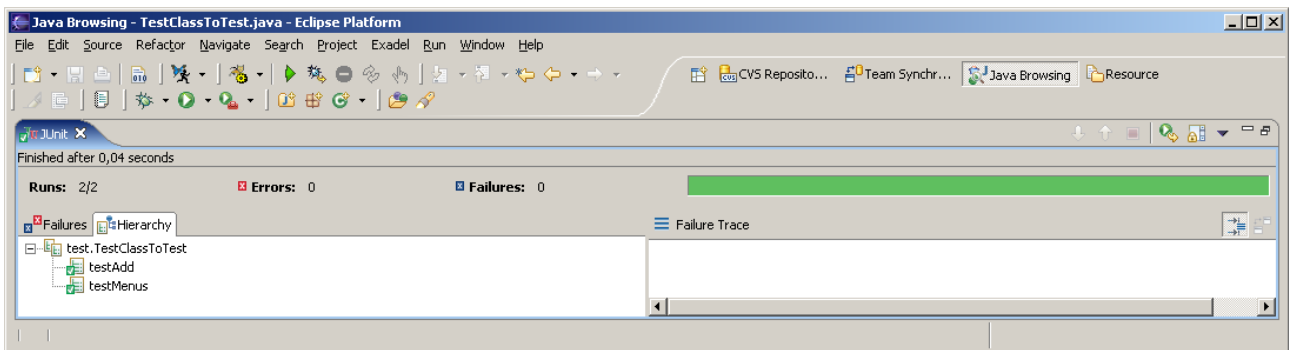
Ensuite on teste chaque méthode de la classe à tester : `testAdd()` pour tester la méthode `add()`.

Une série de méthodes fournies par Junit permettent de valider que le résultat attendu correspond bien au résultat réel. Ici nous utilisons par exemple `assertEquals(int, int)`.

Doc sur assertXXX : <http://www.junit.org/junit/javadoc/3.8.1/junit/framework/Assert.html>

Test positif :

Dans eclipse nous obtenons :

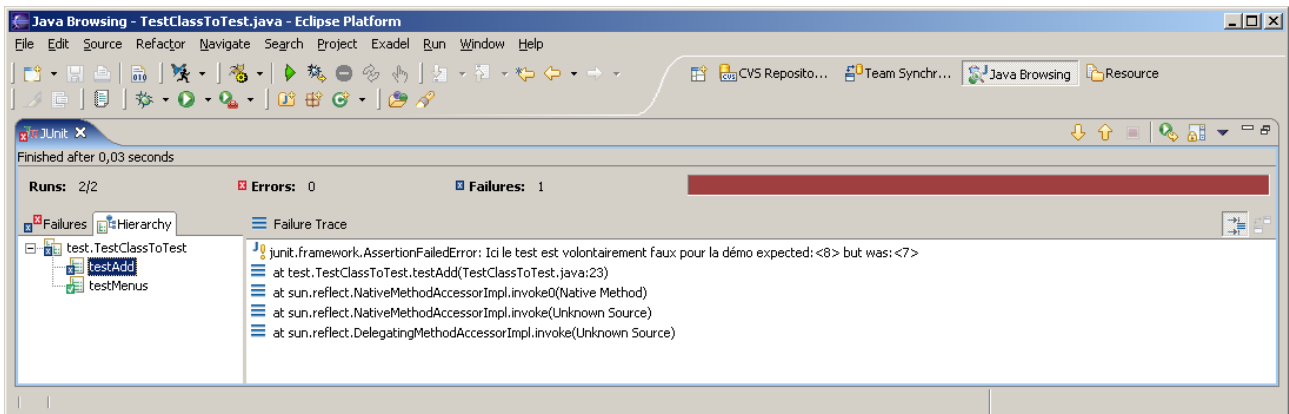


Les tests sont tous positifs.

Test négatif :

Retirons les commentaires de la ligne :

```
// assertEquals("Ici le test est volontairement faux pour la démo", 8, test.add());
```



Très rapidement on comprend que le résultat attendu est 8 mais que le programme donne 7 :

```
junit.framework.AssertionFailedError: Ici le test est volontairement faux pour la démo expected:<8>  
but was:<7>
```

Lien sur AssertionError :

<http://www.junit.org/junit/javadoc/3.8.1/junit/framework/AssertionFailedError.html>

Exemple avec Spring :

Pour initialiser le contexte de test nous utilisons la beanFactory du framework Spring. Le framework Spring se charge de mettre en mémoire et en relation les Services avec les Daos, d'initialiser le pool de connections, de mettre en place les transactions avec Hibernate, etc.

Il faut noter que l'on doit initialiser le pool de connections qui est normalement démarré avec le serveur d'application Tomcat. Il suffit donc de rajouter les librairies concernant ce pool normalement géré par Tomcat : commons-pool-1.1.jar

Classe qui teste le serviceToDo :

```
/*
 */
package test;

import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import jotodo.biz.bo.ToDo;
import jotodo.biz.bs.ToDoServiceImpl;
import jotodo.biz.exception.JoTestException;
import junit.framework.TestCase;

/**
 * Example of unit test of Spring Service.
 */
public class TestToDoServiceImpl extends TestCase {

    private XmlBeanFactory bf;
    private ToDo todo;
    private ToDo todo2;
    private ToDoServiceImpl toDoService;

    /**
     * Constructor.
     */
    public TestToDoServiceImpl() {
        System.out.println("#JUNIT## CONSTRUCTOR");
    }

    /**
     * @see TestCase#setUp()
     */
    protected void setUp() throws Exception {
        super.setUp();
        // Load Application context.
        System.out.println("#JUNIT## SETUP");
        bf = new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));
        System.out.println("#JUNIT## Init du contexte de l'application.");
        toDoService = (ToDoServiceImpl) bf.getBean("toDoServiceTarget");
        System.out.println("#JUNIT## Get a reference on singleton toDoService");

        todo = new ToDo();
        todo.setBody("#JUNIT## UNIT TEST BODY");
        todo.setTitle("#JUNIT## TITLE UNIT TEST");
    }

    /**
     * Scenarios Save Get Update Delete todo.
     */
    public void testSaveGetUpdateDeleteToDo() {
        try {
            // SAVE ...
            toDoService.saveToDo(todo);
            System.out.println("#JUNIT## SaveToDo() With ID " + todo.getId());
            assertTrue(true);
            // GET ...
            todo2 = toDoService.getToDo(todo.getId());
            System.out.println("#JUNIT## GetToDo() With ID " + todo.getId()
                + "\n    todo title : " + todo2.getTitle());
            assertTrue(true);
            // UPDATE ...
            todo2.setTitle("UPDATE TITLE UNIT TEST ");
            toDoService.updateToDo(todo2);
            System.out.println("#JUNIT##Update todo With ID " + todo2.getId()
                + "\n    todo title : " + todo2.getTitle());
            assertTrue(true);
            // DELETE ...
            toDoService.deleteToDo(todo2);
            assertTrue(true);
        } catch (JoTestException e) {
            assertTrue("#JUNIT## \n " + e.getMessage().toString(), false);
            e.printStackTrace();
        }
    }
}
```

```
bf = new XmlBeanFactory(new ClassPathResource("applicationContext.xml"));
```

- Initialise la beanFactory de Spring dans le setup.

```
todoService = (ToDoServiceImpl) bf.getBean("todoServiceTarget");
```

- Donne une référence sur le singleton todoService. C'est à ce moment que la beanFactory initialise tous les beans en rapport avec ce service.

```
todoService.saveToDo(todo);
```

- Début réel des tests.

Bonnes pratiques :

Il est impossible de connaître à l'avance l'ordre d'exécution des méthodes tests. Il est donc déconseillé de faire des méthodes tests ajout, modifications, suppression. Il faut plutôt monter un scénario entier.

Les effets de bords des tests doivent être contenus, pour rendre le système aussi propre que possible même après avoir exécuté les tests unitaires une centaine de fois. Ceci est d'autant plus vrai pour les tests influençant la base de données.

Enfin, il est inutile de déployer les tests. Il est donc préférable de prévoir un package spécifique.

Conclusion

Monter une série de tests de la sorte permet lorsque l'on effectue des modifications de vérifier très rapidement la validité des changements. On évite ainsi une bonne partie des bugs de régressions. Souvent il est préférable de réaliser la classe test avant ou en même temps que la classe elle-même. Ceci contraint le développeur à envisager tous les cas possibles et voire même à les valider avec le client. Le code ainsi produit est de bonne qualité.

Les tests unitaires avec Spring sont un jeu d'enfant. En effet l'inversion de contrôle permet de mettre tout en place en mémoire pour tester tranquillement nos classes.

Remarque : Nous pourrions utiliser log4j pour garder une trace de tous les tests effectués.