

Tutoriel sur le test d'applications Web avec Selenium

Par [Denis Thomas](#) 

Date de publication : 22 juillet 2013

Dernière mise à jour : 22 juillet 2013

CONFIRMÉ

Avec cet article, je vais présenter Selenium, un outil qui nous permettra de tester l'interface utilisateur des applications Web, comment écrire des tests simples à l'aide du pattern Page Object, et comment automatiser ces tests avec Maven.

Commentez

I - Présentation de Selenium.....	3
II - Application à tester.....	3
III - Utilisation de Selenium IDE.....	6
III-A - Installation.....	6
III-B - Création d'un cas de test.....	7
III-C - Insertion manuelle d'une commande.....	9
IV - Tests Selenium sous Eclipse.....	11
IV-A - Dépendances.....	12
IV-B - Export du cas de test.....	12
IV-B-1 - JUnit 4 et WebDriver.....	13
IV-B-2 - JUnit 4 et WebDriver embarqué.....	15
IV-C - Attente du chargement des pages.....	16
IV-D - Vérification du contenu des pages.....	16
IV-D-1 - L'en-tête.....	17
IV-D-2 - Le pied de page.....	18
IV-D-3 - Première page.....	19
IV-D-4 - Deuxième page.....	22
IV-D-5 - Troisième page.....	23
IV-E - Autres navigateurs.....	24
IV-E-1 - HtmlUnitDriver.....	24
IV-E-2 - Opera.....	26
IV-E-3 - Chrome.....	27
V - Page Object Pattern.....	27
V-A - Création des pages.....	28
V-B - La classe de test.....	32
V-C - Autres drivers.....	33
VI - Tests d'intégration automatisés.....	34
VI-A - Le plugin Maven Jetty.....	34
VI-B - Paramétrage des tests.....	35
VI-C - Adaptation de notre test.....	36
VII - Liens.....	37
VIII - Conclusion.....	37
IX - Remerciements.....	37

I - Présentation de Selenium

Dans le cadre du développement d'une application, quelle qu'elle soit, les tests sont indispensables, et prennent une part non négligeable du développement. Il en existe plusieurs types : unitaires, intégration, fonctionnels, qualification, etc. Aujourd'hui, la plupart sont automatisés, ce qui permet un gain de temps substantiel, ainsi qu'une plus grande fiabilité.

Selenium est un de ces outils d'automatisation, concernant les tests d'interface des applications Web. Il se compose de deux parties :

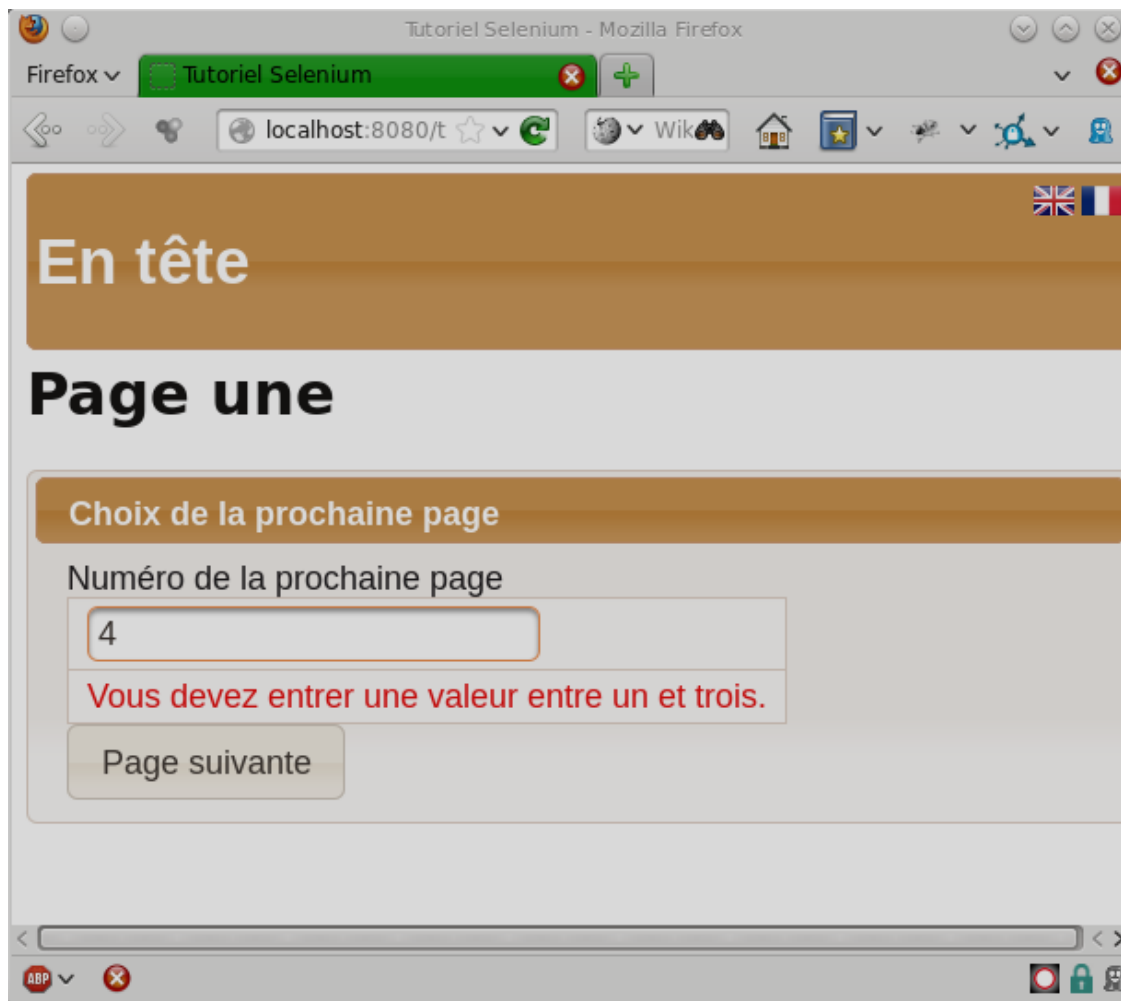
- Selenium IDE : c'est une extension de Firefox, qui permet d'enregistrer une suite d'actions, qu'il sera possible de rejouer à volonté ;
- Selenium WebDriver : il s'agit cette fois d'une API, disponible pour plusieurs langages, permettant de programmer des actions sur l'interface, et à vérifier les réponses. Les actions à réaliser peuvent être exportées depuis Selenium IDE.

Selenium est un projet distribué sous la **licence Apache 2.0**, et peut être téléchargé librement depuis <http://seleniumhq.org/>.

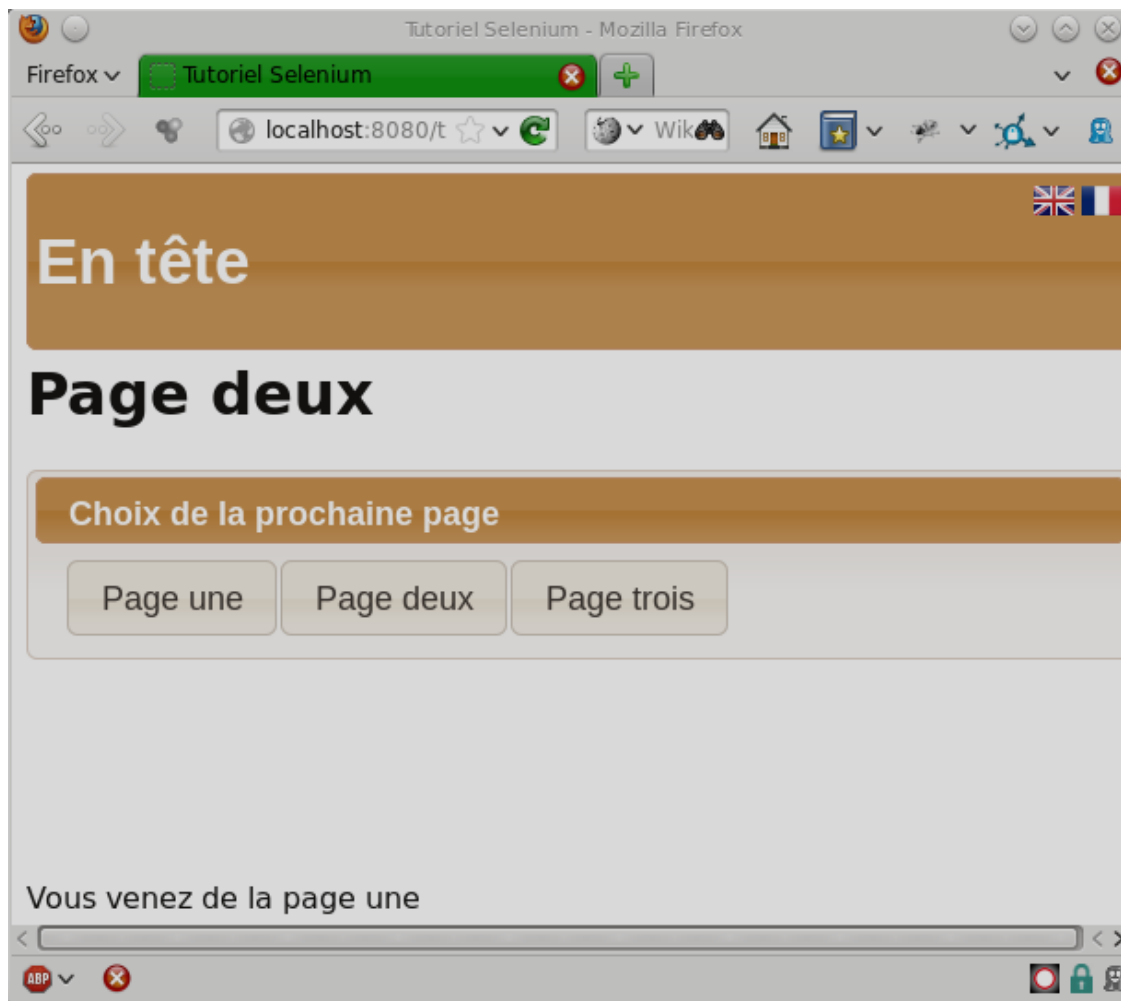
II - Application à tester

Pour les besoins de cet article, j'ai utilisé une petite application basique, composée de trois pages.

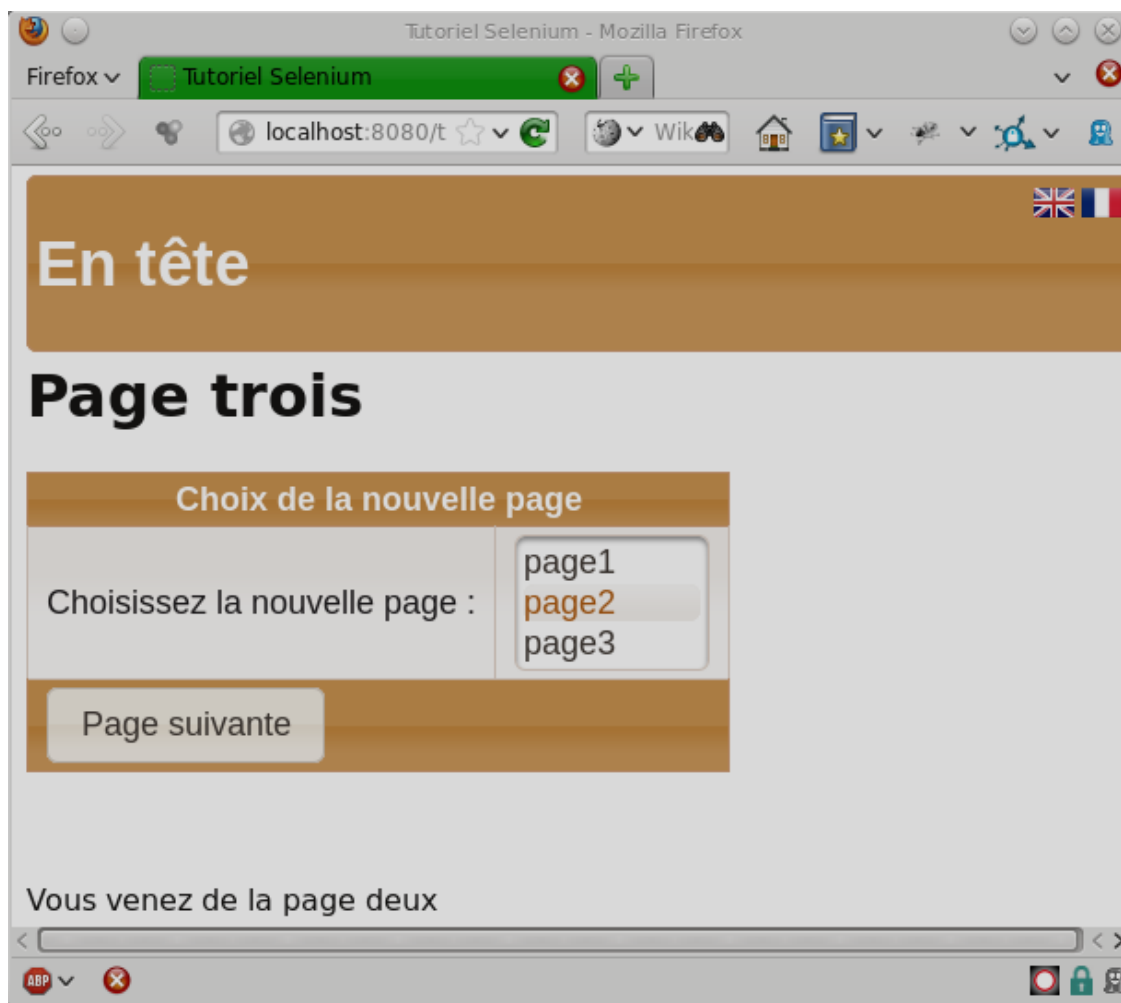
La page une nous permet de choisir la prochaine page en entrant son numéro, compris impérativement entre 1 et 3 :



La page deux nous présente trois boutons pour choisir directement la prochaine page :



Enfin la page 3 affiche la liste des pages disponibles :



Chacune de ces pages se compose :

- d'un en-tête, avec une barre d'outils nous permettant de choisir la langue ;
- d'un pied de page, affichant la page d'où on vient ;
- du contenu spécifique à la page.

Dans les liens à la fin de l'article, vous trouverez le projet complet, comprenant les fichiers de tests Selenium. Je ne décrirai pas le fonctionnement de l'application, ce n'est pas le but ici. Sachez seulement qu'il s'agit d'une application JSF, réalisée avec le framework Primefaces.

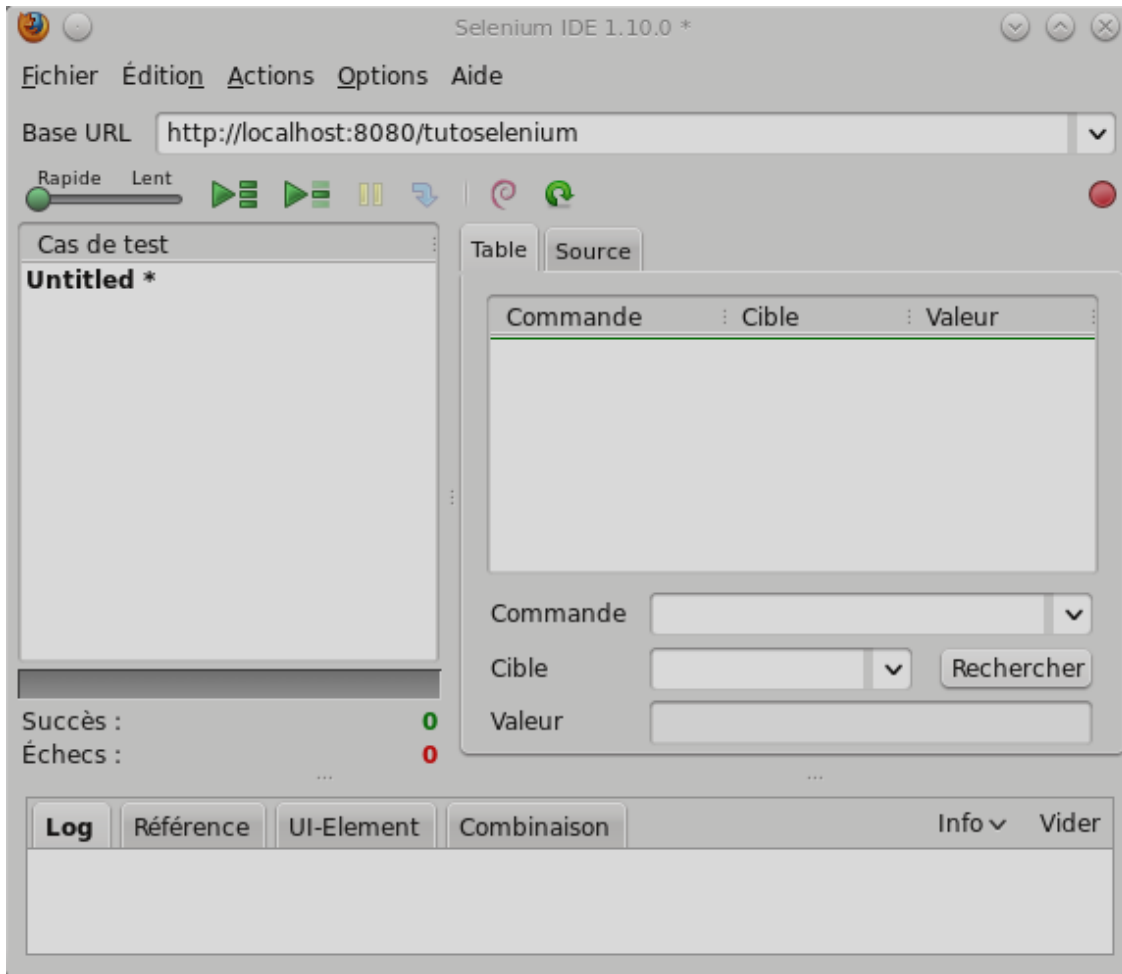
Une fois le projet dézippé et importé dans Eclipse, vous pouvez lancer l'application et vous amuser avec elle quelques instants avant que nous commençons avec Selenium.

III - Utilisation de Selenium IDE

III-A - Installation

Selenium IDE peut être téléchargé et installé depuis [cette page](#). Il s'agit d'une extension pour Firefox. En utilisant ce dernier, l'installation est automatique, moyennant un éventuel petit message d'avertissement à propos de l'installation de plugin et un redémarrage du navigateur. Il est également fortement conseillé d'installer **Firebug**, qui nous permettra d'inspecter la page de l'application.

Une fois Selenium IDE installé, allez dans le menu Développement Web, et lancez-le. Vous devez voir apparaître cette fenêtre :



III-B - Création d'un cas de test

Maintenant que Selenium IDE est installé, nous pouvons commencer la création d'un cas de test. Ceci consiste simplement à enregistrer une suite d'actions.

Lançons l'application dans Eclipse, et attendons que la page d'accueil s'affiche dans Firefox. Dans Selenium IDE, cliquons sur le bouton rouge en haut à droite (ou passons par le menu « Actions »).

La suite d'actions que nous voulons enregistrer est la suivante :

- 1 Page 1 : entrer 2 dans la zone de texte, puis cliquer sur le bouton ;
- 2 Page 2 : cliquer sur le bouton « Page 3 » ;
- 3 Page 3 : sélectionner « Page 1 », puis cliquer sur le bouton « Page suivante » ;
- 4 Page 1 : cliquer sur le drapeau anglais, puis recommencer les étapes 1 à 3.

Au fur et à mesure de vos actions, la table va s'enrichir avec les commandes exécutées :


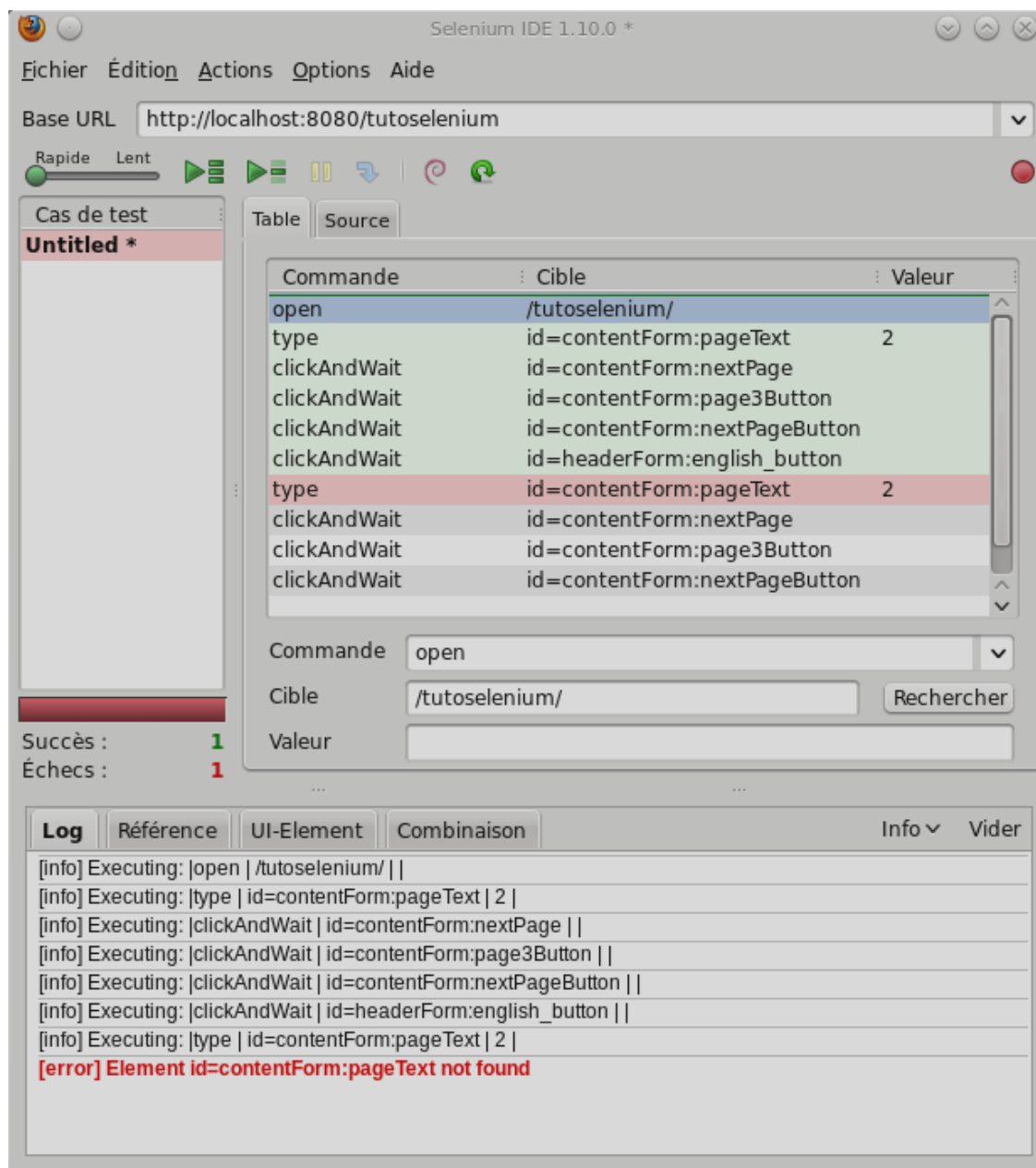


Table	Source	
Commande	Cible	Valeur
open	/tutoselenium/	
type	id=contentForm:pageText	2
clickAndWait	id=contentForm:nextPage	
clickAndWait	id=contentForm:page3Button	
clickAndWait	id=contentForm:nextPageButton	
clickAndWait	id=headerForm:english_button	
type	id=contentForm:pageText	2
clickAndWait	id=contentForm:nextPage	
clickAndWait	id=contentForm:page3Button	
clickAndWait	id=contentForm:nextPageButton	

Et c'est tout, il ne nous reste plus qu'à exécuter autant de fois que nous voulons ce cas de test. On peut évidemment l'enregistrer pour le rejouer plus tard. Mais si nous le jouons en l'état, nous obtenons une erreur :



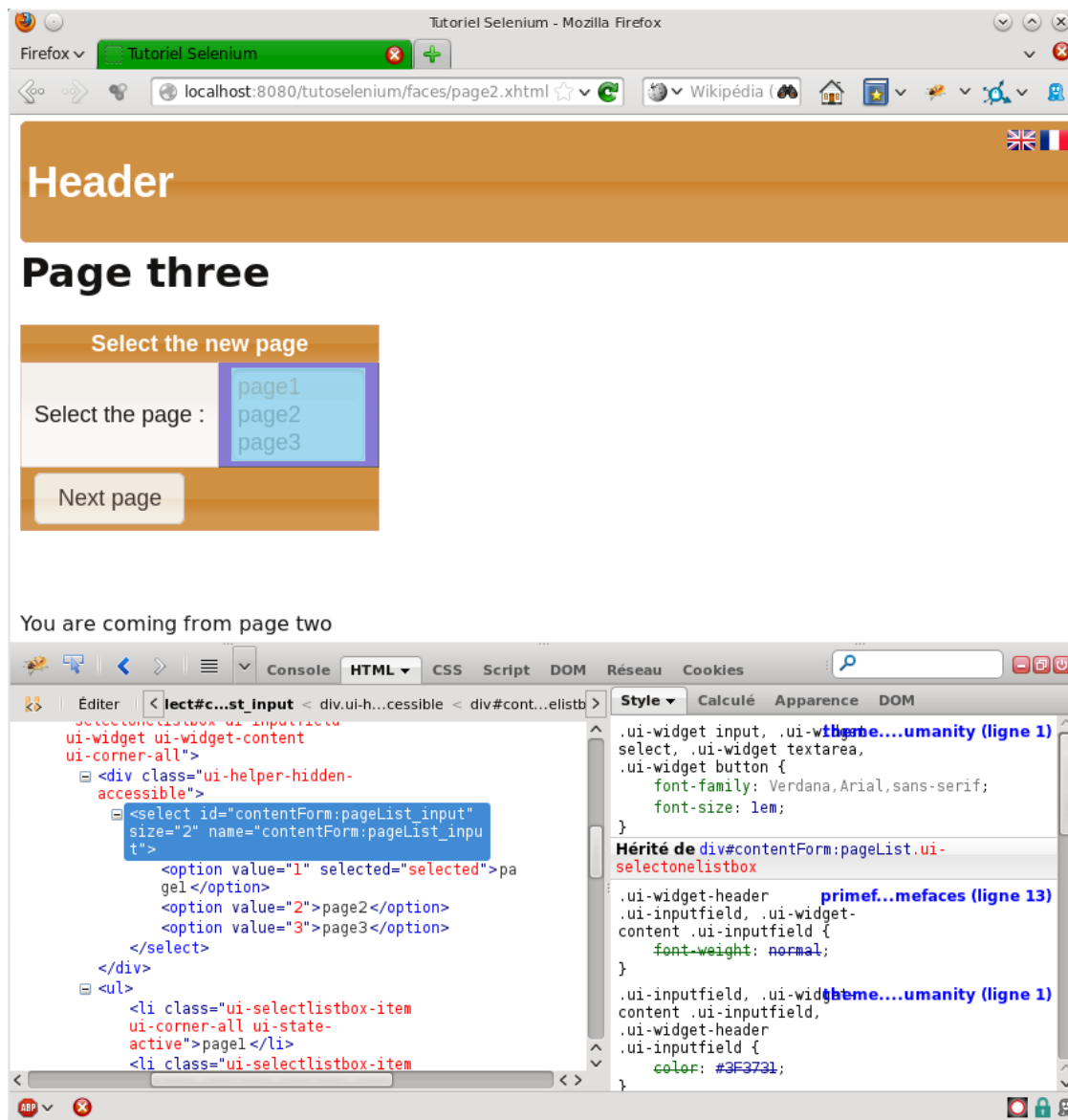
Après la page 3, Selenium IDE ne nous a pas renvoyé sur la page 1 comme prévu, mais sur la page 2. Aussi, quand on cherche l'élément avec l'id « contentForm:pageText », il ne le trouve pas, ce dernier est sur la page 1, et nous sommes sur la page 2...

En arrivant sur la page 3, c'est la page d'où nous venons qui est sélectionnée, c'est-à-dire la page 2. Pour une raison que j'ignore, Selenium n'a pas enregistré notre sélection de la page 1. Donc, quand on clique sur le bouton page suivante, on retourne page 2, et non page 1.

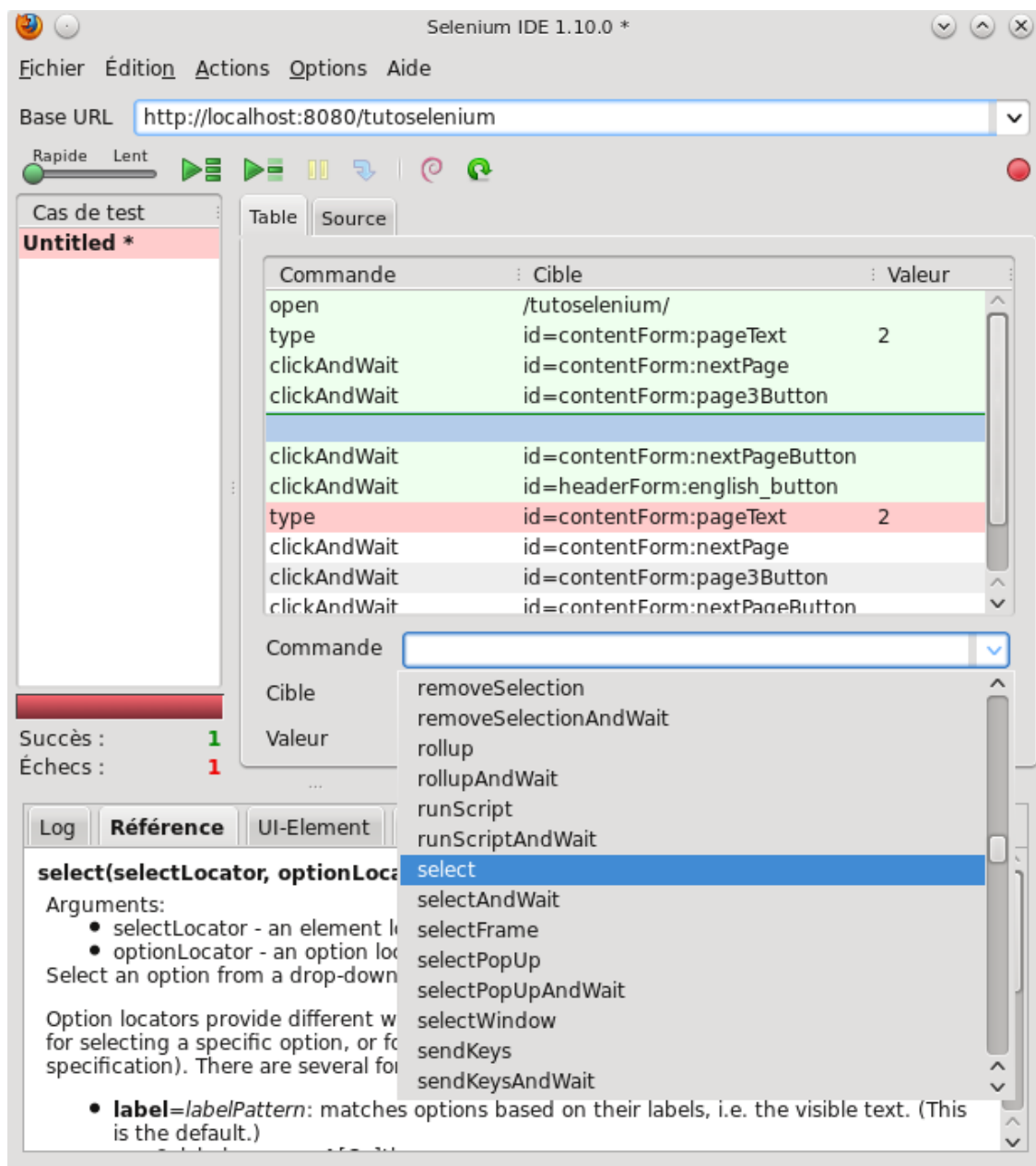
Corrigeons ce problème.

III-C - Insertion manuelle d'une commande

Utilisons Firebug pour inspecter l'arbre DOM de la page 3 :



Nous obtenons ainsi l'ID de l'élément à sélectionner : « contentForm:pageList_input ». Muni de cette information, rendons-nous dans Selenium IDE, sélectionnons la troisième ligne clickAndWait, celle avec la cible « id=contentForm:nextPageButton », puis insérons une nouvelle commande à l'aide du menu contextuel. Pour cette commande, allons dans la liste déroulante Commande, et choisissez « select » :



Dans cible, mettons « id=contentForm:pageList_input », puis dans valeur « value=1 ». Nous devons recommencer la même opération pour la seconde visite de la page 3, celle en anglais. L'onglet « Référence » nous présente la documentation de la commande en cours. Pour la valeur, nous pouvons mettre également « label=page1 », ou utiliser un index.

Exécutons le cas de test, tout fonctionne ! Il ne nous reste plus qu'à le sauvegarder pour le rejouer plus tard si on en a besoin. Mais pour notre plus grand plaisir, on peut aussi en créer une version Java, Python, C# ou Ruby au choix. Voyons ceci sans plus tarder.

IV - Tests Selenium sous Eclipse

Nous avons quelques contraintes pour utiliser les tests Selenium par programmation. La principale est qu'il nous faut disposer du navigateur pour lequel nous voulons utiliser le driver. Selenium IDE est une extension pour le navigateur de Mozilla, et à ma connaissance, il n'en existe pas de version pour les autres navigateurs. Nous disposons donc déjà du navigateur de Mozilla.

Pour les autres drivers, on devra donc installer les navigateurs correspondant : Opera, Chrome (ou Chromium), Internet Explorer. Il existe un driver spécial, ne possédant pas de navigateur associé. Il s'agit de HTMLUnit, dépourvu d'interface graphique. Il est donc tout désigné, par exemple, pour utiliser Selenium sur un serveur d'intégration continu sans interface graphique. Nous verrons cependant qu'il n'est pas parfait. Aucun ne l'est d'ailleurs, chaque navigateur interprétant parfois à sa sauce le code des pages Web, particulièrement les styles CSS.

Je ne présente dans cet article que des bouts de code, qui suffiront, j'espère, à la compréhension de l'ensemble des tests. Vous trouverez le projet Eclipse complet en téléchargement parmi les liens à la fin de l'article.

IV-A - Dépendances

Depuis Selenium IDE, le plus simple pour disposer d'un projet de test est d'exporter notre cas de test. Préparons donc le terrain à l'export de notre cas de test sous Eclipse, en ajoutant les dépendances suivantes dans le pom :

Dépendances Selenium

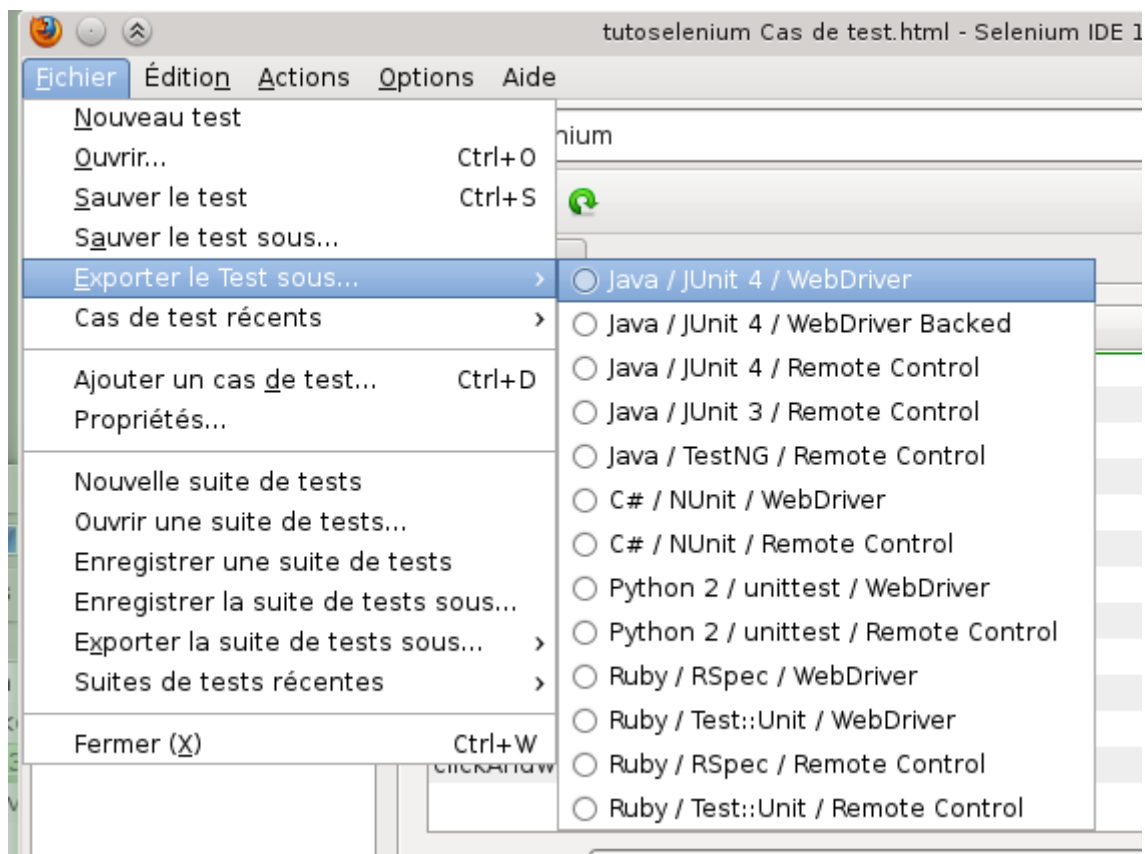
```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>2.27.0</version>
  <scope>test</scope>
</dependency>
```

Créons aussi le package `fr.atatorus.tutoselenium`, dans le répertoire `src/test/java`.

Voilà, nous sommes prêt pour l'export.

IV-B - Export du cas de test

Selenium IDE nous laisse le choix entre plusieurs langages et types d'export. Les langages disponibles sont Java, C#, Python et Ruby, chacun associé avec divers frameworks de tests. Rien que pour Java, nous avons cinq choix possibles :



Nous n'allons pas détailler toutes les possibilités que nous offre Selenium. Attardons-nous seulement sur les deux premières.

IV-B-1 - JUnit 4 et WebDriver

Comme emplacement, choisissons le package créé dans Eclipse, et exportons notre cas de test sous le nom SeleniumTest.java. Rafraichissons le projet Eclipse, nous voyons surgir notre classe de test. J'ai abrégé le code et remplacé les commentaires de Selenium par d'autres, plus génériques. Le code produit par Selenium est plutôt assez fruste.

Selenium : cas de test exporté

```
1. public class Selenium {
2.     private WebDriver driver;
3.     private String baseUrl;
4.     private boolean acceptNextAlert = true;
5.     private StringBuffer verificationErrors = new StringBuffer();
6.
7.     @Before
8.     public void setUp() throws Exception {
9.         // On instancie notre driver, et on configure notre temps d'attente
10.        driver = new FirefoxDriver();
11.        baseUrl = "http://localhost:8080/tutoselenium";
12.        driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
13.    }
14.
15.    @Test
16.    public void testSelenium() throws Exception {
17.        // On se connecte au site
18.        driver.get(baseUrl + "/tutoselenium/");
19.
20.        // On se rend page 1
21.        driver.findElement(By.id("contentForm:pageText")).clear();
22.        driver.findElement(By.id("contentForm:pageText")).sendKeys("2");
```

Selenium : cas de test exporté

```
23.     driver.findElement(By.id("contentForm:nextPage")).click();
24.
25.     // On est page 2, on va page 3
26.     driver.findElement(By.id("contentForm:page3Button")).click();
27.
28.     // On sélectionne notre prochaine page dans la liste
29.
30.     new Select(driver.findElement(By.id("contentForm:pageList_input"))).selectByVisibleText("1");
31.     driver.findElement(By.id("contentForm:nextPageButton")).click();
32.
33.     // On est de retour page 1, on passe en anglais
34.     driver.findElement(By.id("headerForm:english_button")).click();
35.
36.     // Et on recommence le même enchainement
37.     ...
38. }
39.
40. @After
41. public void tearDown() throws Exception {
42.     driver.quit();
43.     String verificationErrorString = verificationErrors.toString();
44.     if (!"".equals(verificationErrorString)) {
45.         fail(verificationErrorString);
46.     }
47.
48.     private boolean isElementPresent(By by) {
49.         try {
50.             driver.findElement(by);
51.             return true;
52.         } catch (NoSuchElementException e) {
53.             return false;
54.         }
55.     }
56.
57.     private String closeAlertAndGetItsText() {
58.         try {
59.             Alert alert = driver.switchTo().alert();
60.             if (acceptNextAlert) {
61.                 alert.accept();
62.             } else {
63.                 alert.dismiss();
64.             }
65.             return alert.getText();
66.         } finally {
67.             acceptNextAlert = true;
68.         }
69.     }
70. }
```

Corrigeons vite fait le package pour qu'il corresponde au nôtre, ainsi que le nom de la classe. Mais attendons un peu avant de lancer Tomcat et notre test, car nous n'irons pas bien loin. L'export comporte en effet quelques menues erreurs.

Tout d'abord, dans la méthode setUp(), on définit baseUrl, qui finit par « /tutoselenium » tout à fait normalement. Il ne faut donc pas l'ajouter une seconde fois quand on demande au driver de se connecter. Le début du test devient :

```
@Test
public void testSelenium() throws Exception {
    // On se connecte au site
    driver.get(baseUrl);
    ...
}
```

Ensuite page 3, pour sélectionner la page une dans la liste, on utilise la méthode selectByVisibleText("1"). Or ce n'est pas par le texte affiché que nous voulons sélectionner la page, mais par la valeur. Il faut donc utiliser selectByValue() :

```
new Select(driver.findElement(By.id("contentForm:pageList_input"))).selectByValue("1");
```

On peut maintenant exécuter le test. Lançons Tomcat depuis Eclipse, puis le test (avec Run As > JUnit Test). Il doit être vert.

Si nous regardons le code, la première chose que nous voyons, c'est que tout passe par l'utilisation de la classe **org.openqa.selenium.WebDriver**. Il s'agit de l'interface qui nous permet de manipuler la page Web. Nous utilisons ici l'implémentation Firefox, nous verrons plus tard les autres implémentations ⁽¹⁾ et leurs différences.

La deuxième chose qu'on remarque, dans la méthode setUp(), est l'appel à implicitlyWait(). Nous parlerons un peu plus loin de son utilité, quand nous aborderons le chapitre sur l'attente du chargement des pages Web.

Pratiquement tout notre test consiste à trouver un élément de la page, avec la méthode findElement(), et appeler une méthode sur cet élément pour simuler une action de l'utilisateur. Les éléments sont sélectionnés par leur id, mais d'autres modes sont possibles : par classe CSS ou par balise notamment.

Selenium a également généré quelques méthodes privées, non utilisées pour l'instant, mais qui peuvent se révéler utiles. En effet, si ce test passe vert, il ne fait aucune vérification. Il se contente d'enchaîner les actions, sans vérifier le contenu des pages. Selenium étant bien évidemment incapable de deviner à quoi doivent ressembler nos pages Web, nous allons faire tout ceci à la main.

Passons maintenant au test suivant, avec le driver « embarqué ».

IV-B-2 - JUnit 4 et WebDriver embarqué

Cette fois, nous n'utilisons pas directement le driver, mais à travers la classe **com.thoughtworks.selenium.Selenium**. Et cette fois, le code fonctionne directement sans modification :

```
public class selenium {
    private Selenium selenium;

    @Before
    public void setUp() throws Exception {
        WebDriver driver = new FirefoxDriver();
        String baseUrl = "http://localhost:8080/tutoselenium";
        selenium = new WebDriverBackedSelenium(driver, baseUrl);
    }

    @Test
    public void testSelenium() throws Exception {
        // Connexion au site
        selenium.open("/tutoselenium/");

        // On est page 1, on va page 2
        selenium.type("id=contentForm:pageText", "2");
        selenium.click("id=contentForm:nextPage");
        selenium.waitForPageToLoad("30000");

        // puis page 3
        selenium.click("id=contentForm:page3Button");
        selenium.waitForPageToLoad("30000");

        // et retour page 1
        selenium.select("id=contentForm:pageList_input", "value=1");
        selenium.click("id=contentForm:nextPageButton");
        selenium.waitForPageToLoad("30000");

        // On passe en anglais
        selenium.click("id=headerForm:english_button");
    }
}
```

(1) À l'exception de Internet Explorer : Windows ne me sert que pour les jeux, je n'ai aucun outil de dev sur ma machine Windows. Je vous laisse les tests avec IE à titre d'exercice.

```
// etc.  
}  
  
@After  
public void tearDown() throws Exception {  
    selenium.stop();  
}  
}
```

Le code est très proche, la seule différence est que plutôt que passer par le driver pour sélectionner un élément, nous utilisons la classe **com.thoughtworks.selenium.Selenium**. On remarque aussi cette fois que nous attendons désormais à chaque changement de page.

En ce qui nous concerne, nous utiliserons un mélange des deux méthodes. Nous utiliserons le driver pour sélectionner un élément et interagir avec lui, mais nous aurons besoin de l'objet Selenium pour attendre le chargement de la page complète. Voyons donc comment gérer le temps d'attente de la mise à disposition des pages par le serveur Web.

IV-C - Attente du chargement des pages

Avant de tester une page, nous devons nous assurer que le serveur a fini de l'envoyer. Selenium offre plusieurs moyens d'attendre que la page soit disponible. Le premier est d'attendre le chargement de la page complète :

```
selenium.waitForPageToLoad("30000");
```

Par une bizarrerie que je ne m'explique pas, le paramètre à passer est le temps d'attente en millisecondes sous forme de String... Une autre manière d'attendre est celle que nous avons vue plus haut, avec une attente implicite :

```
driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
```

Ici, nous demandons au driver d'attendre au plus 30 secondes la disponibilité d'un élément quand il interroge le DOM. Une fois ce délai écoulé, si l'élément n'est toujours pas disponible, il lance l'exception **NoSuchElementException**.

Enfin, nous avons l'attente explicite :

```
WebDriverWait wait = new WebDriverWait(driver, 30);  
WebElement element =  
    wait.until(ExpectedConditions.presenceOfElementLocated(By.id("id_element")));
```

Ici, l'attente ne concerne qu'un élément, celui dont on précise l'identifiant. À priori, l'attente explicite fait doublon avec l'attente implicite, celle-ci s'appliquant à tous les éléments.

Ces trois attentes ne font pas forcément double emploi. Alors que la première sert à attendre la page, les deux autres peuvent avoir leur utilité dans le cas où le DOM serait modifié sans recharger la page complète, par exemple avec une requête AJAX. Quant à l'attente explicite, elle sera utile après un événement particulier, dont on sait par avance qu'il prendra plus de temps que la normale.

N'ayant pas de code Ajax, ni de modification dynamique du DOM, nous nous contenterons d'attendre simplement le chargement complet de la page. Nous aurions pu utiliser l'attente implicite, mais cela provoque quelques problèmes aléatoires avec le driver pour Opera.

IV-D - Vérification du contenu des pages

Maintenant que nous savons enchaîner nos pages, nous devons penser à les vérifier.

Commençons par ajouter quelques méthodes pour contrôler nos pages quand nous arrivons dessus. Notre nouvelle méthode de test devient :

```
@Test
public void testSelenium() throws Exception {

    // Connection
    driver.get(baseUrl);
    selenium.waitForPageToLoad(PAGE_TO_LOAD_TIMEOUT);
    // Vérification
    checkHeader(Locale.FRENCH);
    checkPageUne(false, Locale.FRENCH);

    checkFooter(); // On vient de se connecter, on n'affiche donc pas le numéro de la page précédente

    // Avant d'aller page 2, on provoque une erreur
    driver.findElement(By.id("contentForm:pageText")).clear();
    driver.findElement(By.id("contentForm:pageText")).sendKeys("4");
    driver.findElement(By.id("contentForm:nextPage")).click();
    checkPageUne(true, Locale.FRENCH);

    // On va page 2
    driver.findElement(By.id("contentForm:pageText")).clear();
    driver.findElement(By.id("contentForm:pageText")).sendKeys("2");
    driver.findElement(By.id("contentForm:nextPage")).click();
    selenium.waitForPageToLoad(PAGE_TO_LOAD_TIMEOUT);
    // vérification
    checkHeader(Locale.FRENCH);
    checkPageDeux(Locale.FRENCH);
    checkFooter("page une", Locale.FRENCH); // On vient de la page 1 en français

    // On va page 3
    driver.findElement(By.id("contentForm:page3Button")).click();
    selenium.waitForPageToLoad(PAGE_TO_LOAD_TIMEOUT);
    // vérification
    checkHeader(Locale.FRENCH);
    checkPageTrois(Locale.FRENCH);
    checkFooter("page deux", Locale.FRENCH);

    // on retourne page 1
    new Select(driver.findElement(By.id("contentForm:pageList_input"))).selectByValue("1");
    driver.findElement(By.id("contentForm:nextPageButton")).click();
    selenium.waitForPageToLoad(PAGE_TO_LOAD_TIMEOUT);
    checkFooter("page trois", Locale.FRENCH);

    // On passe en anglais
    driver.findElement(By.id("headerForm:english_button")).click();
    selenium.waitForPageToLoad(PAGE_TO_LOAD_TIMEOUT);

    // Et on recommence
    ...
}
```

Rien de bien compliqué : après chaque action, on attend le chargement de la page (100 ms suffisent amplement) et on vérifie que tout est correct. Il nous reste à voir ces diverses méthodes de contrôle.

IV-D-1 - L'en-tête

Notre en-tête contient seulement un label, et deux boutons permettant de changer la langue :

header.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/
html" xmlns:p="http://primefaces.org/ui">

<p:toolbar>
```

header.xhtml

```
<p:toolbarGroup align="left">
  <h1>
    <h:outputText value="#{msg['header.title']}" />
  </h1>
</p:toolbarGroup>

<p:toolbarGroup align="right">
  <h:commandButton id="english_button" styleClass="flags english" action="#{prefs.setEnglishLocale()}" />
  <h:commandButton id="french_button" styleClass="flags french" action="#{prefs.setFrenchLocale()}" />
</p:toolbarGroup>

</p:toolbar>
</html>
```

Nous avons deux choses à vérifier : le titre de l'entête, et les drapeaux. Pour connaître le titre de l'en-tête, il suffit de regarder les fichiers messages. Nous n'avons qu'une seule balise <H1>, aussi la sélection se fera par balise :

```
private void checkHeader(Locale locale) {
    WebElement title = driver.findElement(By.tagName("h1"));
    assertThat(title.getText(), is(locale == Locale.FRENCH ? "En tête" : "Header"));
    ...
}
```

Pour ce qui est des drapeaux, ils sont obtenus par CSS, avec une image de fond après une petite manipulation de FacesServlet :

```
.english {
    background-image: url("#{resource['images:drapeau_anglais.png']}");
}

.french {
    background-image: url("#{resource['images:drapeau_francais.png']}");
}
```

Il nous faut donc vérifier que notre servlet en frontal fait correctement la substitution avec notre image (on peut en être raisonnablement sûr), et que nous ne nous sommes pas trompés (on peut l'être un peu moins cette fois). Comme nos drapeaux sont inclus dans un formulaire, nous devons ajouter l'id du formulaire, « headerForm », à l'id de nos boutons :

```
private void checkHeader(Locale locale) {
    ...
    String drapeau =
driver.findElement(By.id("headerForm:english_button")).getCssValue("background-image");
    assertThat(drapeau, is(buildUrl("drapeau_anglais.png")));
    drapeau = driver.findElement(By.id("headerForm:french_button")).getCssValue("background-
image");
    assertThat(drapeau, is(buildUrl("drapeau_francais.png")));
}

private String buildUrl(String flag) {
    return "url(\"" + baseUrl + "/faces/javax.faces.resource/" + flag + "?ln=images\")";
}
```

Le principe est à chaque fois le même : on trouve notre élément, par sa balise ou son id, puis on vérifie une de ses caractéristiques. Ici, le texte pour le titre, et la valeur CSS background-image pour les drapeaux.

IV-D-2 - Le pied de page

Le pied de page est encore plus simple que l'en-tête :

footer.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:ui="http://java.sun.com/jsf/facelets" >

    <ui:fragment rendered="#{empty navigator.previousPage ? 'false' : 'true'}">
        <h:outputText id="previousPage" value="#{navigator.previousPageMessage}" />
    </ui:fragment>

</html>
```

Il ne contient qu'un simple texte, et encore pas toujours. On a donc deux vérifications à faire, avec et sans :

```
private void checkFooter() {
    assertThat(isElementPresent(By.id("previousPage")), is(false));
}

private void checkFooter(String fromPage, Locale locale) {
    if (locale == Locale.FRENCH) {
        checkElement("previousPage", "Vous venez de la " + fromPage);
    } else {
        checkElement("previousPage", "You are coming from " + fromPage);
    }
}

private void checkElement(String elementId, String expected) {
    assertThat(driver.findElement(By.id(elementId)).getText(), is(expected));
}

private void checkElement(String parentId, String elementId, String expected) {
    checkElement(parentId + ":" + elementId, expected);
}
```

C'est toujours le même principe, on récupère un élément, puis on vérifie sa caractéristique qui nous intéresse, ici le texte. On en a profité pour créer la méthode `checkElement()`, car nous l'utiliserons assez souvent.

IV-D-3 - Première page

Commençons les choses sérieuses, avec la vérification du contenu des pages elles-mêmes. Ces pages sont incluses dans un fichier `layout.xhtml` :

layout.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/
html" xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets"
>

<h:head>
    <title>Tutoriel Selenium</title>
    <h:outputStylesheet name="styles.css" library="css" />
</h:head>

<h:body>
    <f:view locale="#{prefs.locale}">

        <div id="header">
            <h:form id="headerForm">
                <ui:include src="header.xhtml" />
            </h:form>
        </div>

        <div id="content">
```

layout.xhtml

```
<h:form id="contentForm">
    <ui:insert name="content">
        <h2>Content</h2>
    </ui:insert>
</h:form>
</div>

<div id="footer">
    <ui:include src="footer.xhtml" />
</div>

</f:view>
</h:body>
</html>
```

C'est la balise `<div id="content" />` qui va contenir notre page :

page1.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:p="http://primefaces.org/ui"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core" template="layout.xhtml">

    <ui:define name="content">
        <h1>
            <h:outputText id="pageTitle" value="#{msg['page1.title']}" />
        </h1>

        <p:panel id="panel" header="#{msg['page1.panel_title']}">
            <h:outputLabel id="label" for="nextPage" value="#{msg['page1.text_page_label']}" />

            <p:panelGrid columns="1">
                <p:inputText id="pageText" value="#{navigator.nextPage}"
                    converterMessage="#{msg['error.conversion_to_integer']}"
                    validatorMessage="#{msg['error.page_number_range']}"
                    <f:validateLongRange minimum="1" maximum="3" />
                </p:inputText>
                <h:message id="pageError" for="pageText" style="color:red" />
            </p:panelGrid>

            <p:commandButton id="nextPage"
                value="#{msg['page1.next_page_button']}" action="#{navigator.go}"
                ajax="false" />
        </p:panel>
    </ui:define>
</ui:composition>
```

Notre page est placée à l'intérieur d'un formulaire d'id « contentForm », nous devons ajouter cet id pour obtenir ceux de nos éléments :

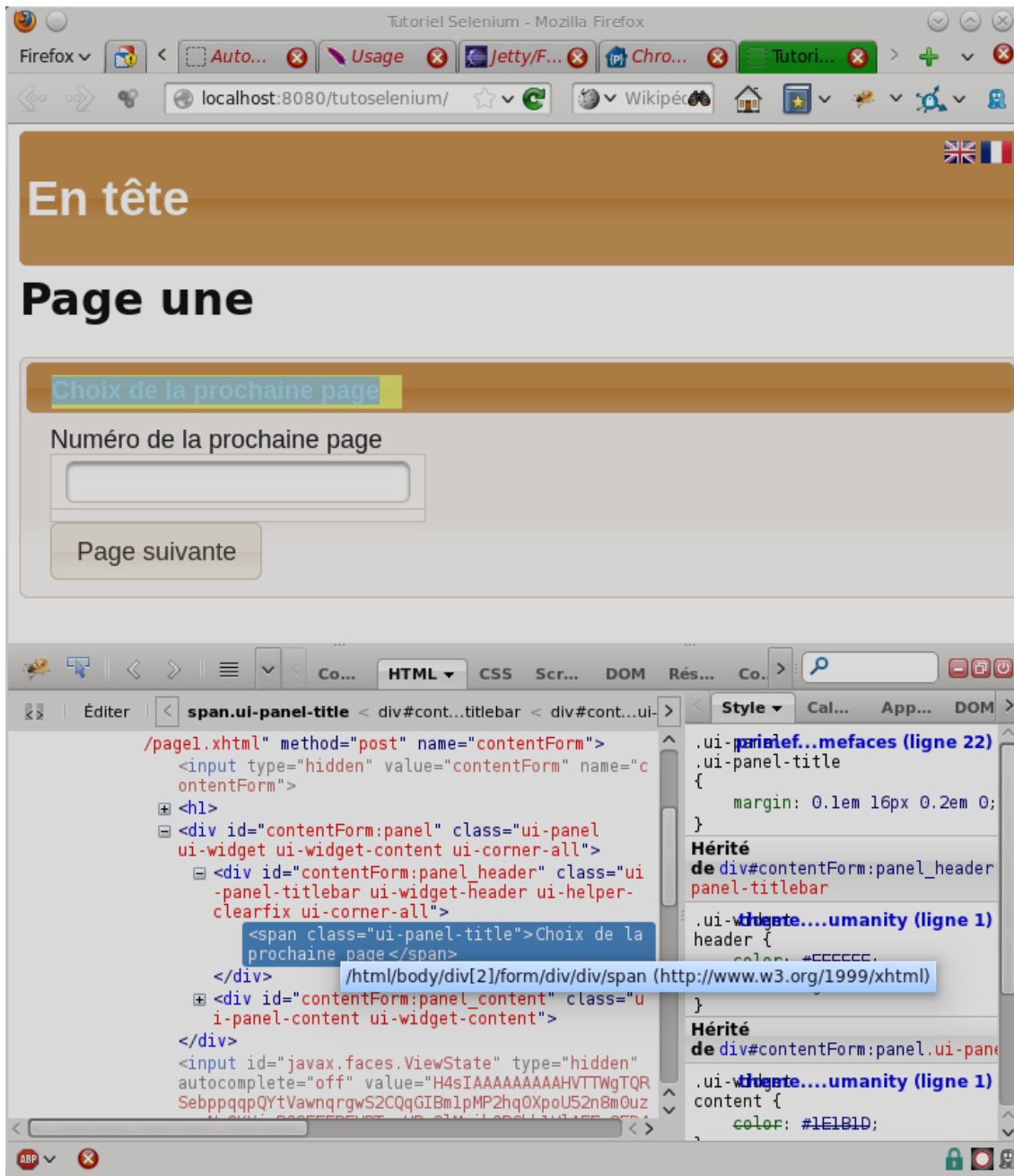
```
private void checkPageUne(boolean errorMessage, Locale locale) {
    checkElement("contentForm", "pageTitle", locale == Locale.FRENCH ? "Page une"
: "Page one");
    checkPanelTitle("contentForm",
        "panel",
        "ui-panel-title",
        locale == Locale.FRENCH ? "Choix de la prochaine page"
: "Select the new page");

    checkElement("contentForm", "label", locale ==
Locale.FRENCH ? "Numéro de la prochaine page"
: "Number of next page :");
    checkElement("contentForm", "nextPage", locale == Locale.FRENCH ? "Page suivante"
: "Next page");
}
```

```
WebElement errorMessageElement = driver.findElement(By.id("contentForm:pageError"));
assertThat(errorMessageElement.isDisplayed(), is(errorMessage));
if (errorMessage) {
    checkElement("contentForm",
        "pageError",
        locale == Locale.FRENCH ? "Vous devez entrer une valeur entre 1 et 3."
        : "You must enter a number between one and three.");
    String color =
driver.findElement(By.id("contentForm:pageError")).getCssValue("color");
    assertThat(color, is("red"));
}

private void checkPanelTitle(String parentId, String panelId, String titleClass, String
expectedTitle) {
    WebElement panel = driver.findElement(By.id(parentId + ":" + panelId));
    WebElement panelTitle = panel.findElement(By.className(titleClass));
    assertThat(panelTitle.getText(), is(expectedTitle));
}
```

Nous introduisons une nouvelle méthode pour vérifier le titre de notre panneau. Si on examine notre page avec Firebug, nous remarquons que le titre de notre panel ne peut être sélectionné que par sa classe CSS :



Mais ce n'est pas grave du tout, on y arrive très bien comme vous le voyez. Exécutons le test, et c'est le drame : il ne passe plus... Firefox a eu, en effet, l'idée de remplacer le style de notre message d'erreur :

```
style="color:red"
```

par :

```
style="color:rgba(255, 0, 0, 1)"
```

Corrigons notre test (ce ne sera pas la dernière fois...), et tout rentrera dans l'ordre.

IV-D-4 - Deuxième page

La page deux est beaucoup plus simple à tester, nous n'avons que trois boutons :

page2.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:p="http://primefaces.org/ui"
    xmlns:ui="http://java.sun.com/jsf/facelets" template="layout.xhtml">

    <ui:define name="content">
        <h1>
            <h:outputText id="pageTitle" value="#{msg['page2.title']}" />
        </h1>

        <p:panel id="panel" header="#{msg['page2.panel_title']}">
            <p:commandButton id="page1Button"
                value="#{msg['page2.page1_button']}" action="#{navigator.page1()}"
                ajax="false" />

            <p:commandButton id="page2Button"
                value="#{msg['page2.page2_button']}" action="#{navigator.page2()}"
                ajax="false" />

            <p:commandButton id="page3Button"
                value="#{msg['page2.page3_button']}" action="#{navigator.page3()}"
                ajax="false" />

        </p:panel>
    </ui:define>
</ui:composition>
```

Et voici le code de notre méthode :

```
private void checkPageDeux(Locale locale) {
    checkElement("contentForm", "pageTitle", locale == Locale.FRENCH ? "Page deux"
: "Page two");
    checkPanelTitle("contentForm",
        "panel",
        "ui-panel-title",
        locale == Locale.FRENCH ? "Choix de la prochaine page"
: "Select the new page");

    checkElement("contentForm", "page1Button", locale == Locale.FRENCH ? "Page une"
: "Page one");
    checkElement("contentForm", "page2Button", locale == Locale.FRENCH ? "Page deux"
: "Page two");
    checkElement("contentForm", "page3Button", locale == Locale.FRENCH ? "Page trois"
: "Page three");
}
```

IV-D-5 - Troisième page

Notre page trois est légèrement plus complexe, avec une liste :

page3.xhtml

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:p="http://primefaces.org/ui"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core" template="layout.xhtml">

    <ui:define name="content">
        <h1>
            <h:outputText id="pageTitle" value="#{msg['page3.title']}" />
        </h1>

        <p:panelGrid id="panel" columns="2">
```

page3.xhtml

```
<f:facet name="header">
    #{msg['page3.panel_title']}
</f:facet>

<p:column>
    <p:outputLabel id="label" for="pageList" value="#{msg['page3.select']}" />
</p:column>

<p:column>
    <p:selectOneListbox id="pageList" value="#{navigator.nextPage}">
        <f:selectItems value="#{navigator.pages}" />
    </p:selectOneListbox>
</p:column>

<f:facet name="footer">
    <p:commandButton id="nextPageButton"
        value="#{msg['page3.next_page_button']}" action="#{navigator.go}"
        ajax="false" />
</f:facet>

</p:panelGrid>
</ui:define>
</ui:composition>
```

Voici notre méthode :

```
private void checkPageTrois(Locale locale) {
    checkElement("contentForm", "pageTitle", locale == Locale.FRENCH ? "Page trois"
: "Page three");
    checkPanelTitle("contentForm",
        "panel",
        "ui-panelgrid-header",
        locale == Locale.FRENCH ? "Choix de la nouvelle page"
: "Select the new page");

    checkElement("contentForm", "label", locale ==
Locale.FRENCH ? "Choisissez la nouvelle page :"
: "Select the page :");

    checkElement("contentForm", "nextPageButton", locale == Locale.FRENCH ? "Page suivante"
: "Next page");
    Select select = new Select(driver.findElement(By.id("contentForm:pageList_input")));
    List<WebElement> options = select.getOptions();
    assertThat(options.get(0).getText(), is("page1"));
    assertThat(options.get(1).getText(), is("page2"));
    assertThat(options.get(2).getText(), is("page3"));
}
```

Ce n'est pas beaucoup plus compliqué que nos méthodes précédentes, si ce n'est ici que nous devons contrôler les options de notre liste.

Maintenant que nous savons que le contenu de nos trois pages est exactement celui attendu, voyons ce qu'il en est avec d'autres navigateurs.

IV-E - Autres navigateurs

IV-E-1 - HtmlUnitDriver

Comme expliqué précédemment, ce driver ne possède pas d'interface graphique. De plus, il émule le comportement JavaScript des autres navigateurs. Il n'est donc pas idéal pour vérifier le parfait comportement de notre application avec un véritable navigateur, mais son absence d'interface graphique peut être un atout si on utilise un serveur d'intégration sans interface graphique. De plus, il est légèrement plus rapide que les autres drivers, absence d'interface graphique oblige. Modifions légèrement notre classe de test :


```
@Before
public void setUp() throws Exception {
    baseUrl = "http://localhost:8080/tutoselenium";
}

@Test
public void firefoxTest() throws Exception {
    driver = new FirefoxDriver();
    testSelenium();
}

@Test
public void htmlUnitTest() throws Exception {
    driver = new HtmlUnitDriver(true);
    testSelenium();
}

private void testSelenium() throws Exception {
    driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
    // Aucun changement dans le code du test
}
```

Voilà, nous avons maintenant deux méthodes de test, une pour Firefox une pour HtmlUnit, et chacune appelle la même méthode de test. Exécutons le test, nous obtenons une première erreur pour HtmlUnitTest :

```
java.lang.AssertionError:
Expected: is <false>
but: was <true>
at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
at org.junit.Assert.assertThat(Assert.java:865)
at org.junit.Assert.assertThat(Assert.java:832)
...
```

Eh oui, HtmlUnitdriver ne se comporte pas exactement comme Firefox... Pour une raison que j'ignore, et que j'ai pour l'instant renoncé à chercher, HtmlUnit considère que le message d'erreur est toujours affiché. Dans le cas de Firefox, nous testons comme d'habitude, mais dans le cas de HTMLUnit, nous devons vérifier que le message est vide.

Corrigeons l'erreur et continuons. Nous tombons sur une autre erreur :

```
java.lang.AssertionError:
Expected: is "url(\"http://localhost:8080/tutoselenium/faces/javax.faces.resource/
drapeau_anglais.png?ln=images\")"
but: was "url(/tutoselenium/faces/javax.faces.resource/drapeau_anglais.png?ln=images)"
```

HtmlUnit a supprimé l'adresse de L'URL de nos images de drapeaux... Corrigons cette nouvelle erreur, et recommençons, nous en avons une dernière :

```
java.lang.AssertionError:
Expected: is "rgba(255, 0, 0, 1)"
but: was "red"
```

HtmlUnit ne change pas la description du code couleur de notre message d'erreur. Voici finalement le code corrigé :

```
private void checkPageUne(boolean errorMessage, Locale locale) {
    ...

    if (errorMessage) {
        ...
        String color =
driver.findElement(By.id("contentForm:pageError")).getCssValue("color");
        switch (currentDriver) {
            case FIREFOX_DRIVER:
                assertThat(color, is("rgba(255, 0, 0, 1)"));
                break;
        }
    }
}
```

```

        default:
            assertThat(color, is("red"));
            break;
    }
} else {
    switch (currentDriver) {
        case FIREFOX_DRIVER:

assertThat(driver.findElement(By.id("contentForm:pageError")).isDisplayed(), is(false));
            break;
        default:
            checkElement("contentForm", "pageError", "");
    }
}
}

private Object buildUrl(String resource) {
    switch (currentDriver) {
        case FIREFOX_DRIVER:
            return "url(\"" + baseUrl + "/faces/javax.faces.resource/" + resource + "?
ln=images\")";
        default:
            return "url(/tutoselenium/faces/javax.faces.resource/" + resource + "?
ln=images)";
    }
}
}

```

currentDriver est une simple variable qui est initialisée avec des constantes représentant les différents drivers. Voyons les surprises que nous réserve Opera.

IV-E-2 - Opera

Outre l'installation du navigateur Opera lui-même, nous devons ajouter la dépendance suivante à notre pom :

```

<dependency>
  <groupId>com.opera</groupId>
  <artifactId>operadriver</artifactId>
  <version>1.1</version>
  <scope>test</scope>
</dependency>

```

Ensuite, notre petit bout de code de test :

```

@Test
public void operaTest() throws Exception {
    driver = new OperaDriver();
    currentDriver = OPERA_DRIVER;
    testSelenium();
}

```

Pour que nos tests passent, nous devons tenir compte des spécificités d'Opera. Par chance, ce dernier se comporte presque comme Firefox. Donc, partout où nous avons mis :

```

case FIREFOX_DRIVER:
    // Vérifications

```

nous pouvons mettre :

```

case FIREFOX_DRIVER:
case OPERA_DRIVER:
    // Vérifications

```

Le driver Opera m'a posé quelques problèmes aléatoires avec le temps de chargement des pages. En général, ça passe avec 100 ms, mais pas toujours. J'ai résolu le problème en mettant un temps d'attente de 250 ms. C'est d'autant plus étrange que ce driver est le plus véloce de tous, presque autant que HtmlUnitDriver.

Après avoir vu ce comportement d'Opera, remettons notre code pour attendre le chargement des pages, et terminons par Chrome.

IV-E-3 - Chrome

Outre le navigateur (Chrome ou Chromium), nous avons besoin du driver associé, qui ne vient pas forcément avec. Pour ceux qui utilisent une distribution Linux, il est certainement disponible dans les dépôts officiels avec Chromium. Pour les autres, vous pouvez librement le [télécharger](#).

Pour instancier ce driver, la propriété système webdriver.chrome.driver doit indiquer le chemin où trouver le driver chrome :

```
private static final String CHROME_DRIVER_PATH = "/usr/lib64/chromium/chromedriver";

private static final String CHROME_DRIVER_PATH = "/usr/lib64/chromium/chromedriver";

@Test
public void chromeTest() throws Exception {
    System.setProperty("webdriver.chrome.driver", CHROME_DRIVER_PATH);
    driver = new ChromeDriver();
    currentDriver = CHROME_DRIVER;
    selenium = new WebDriverBackedSelenium(driver, baseUrl);
    testSelenium();
}
```

Si on lance les tests, nous nous apercevons bien vite que Chrome lui aussi a ses propres habitudes, et ne se comporte pas exactement comme ses confrères.

Pour les drapeaux, ça ressemble à du Firefox ou de l'Opera, mais ce n'est pas tout à fait pareil, il manque les guillemets (le défaut est toujours pour HtmlUnitWebDriver) :

```
java.lang.AssertionError:
Expected: is "url(/tutoselenium/faces/javax.faces.resource/drapeau_anglais.png?ln=images) "
but: was "url(http://localhost:8080/tutoselenium/faces/javax.faces.resource/
drapeau_anglais.png?ln=images) "
```

Pour la couleur du message d'erreur, cette fois, c'est comme Firefox ou Opera :

```
java.lang.AssertionError:
Expected: is "red"
but: was "rgba(255, 0, 0, 1) "
```

Ajoutons ceci à notre méthode buildUrl() :

```
case CHROME_DRIVER:
    return "url(" + baseUrl + "/faces/javax.faces.resource/" + resource + "?
ln=images)";
```

Et si nous exécutons nos quatre tests, tout fonctionne enfin !

V - Page Object Pattern

Comme nous venons de le voir, les tests avec Selenium ne sont pas très compliqués dans leur principe. Par contre, pour ce qui est de leur mise en œuvre, c'est une autre paire de manches. Ici, nous avons trois pages, très simples,

et on voit que c'est déjà assez lourd, même en ne montrant que des extraits de code. Pour des applications réelles, avec plusieurs dizaines de pages, chacune avec tout autant de cas de tests, ça va vite devenir infernal à coder.

C'est ici que le **Page Object Pattern** va nous être très utile, si ce n'est indispensable. En résumé, il consiste à créer une classe par page. Toutes les interactions avec les pages passeront par ces classes. En donnant des noms clairs aux méthodes, le test redevient d'un coup beaucoup plus lisible, et surtout réutilisable. Pour ce qui est de la facilité d'écriture, ce pattern s'accorde très bien avec le framework **FluentLenium**. Pour l'importer, il suffit d'ajouter cette dépendance dans notre pom.xml :

```
<dependency>
  <groupId>org.fluentlenium</groupId>
  <artifactId>fluentlenium-core</artifactId>
  <version>0.8.0</version>
  <scope>test</scope>
</dependency>
```

Une autre remarque concernant l'écriture des tests. Jusqu'à présent, nous n'avions qu'un seul test, qui déroule toute notre séquence. Une bonne pratique serait plutôt de découper notre test en petite séquence : page 1 à page 2, page 2 à page 3, etc. Les tests seront plus simples, plus compréhensibles. En contrepartie, leur temps d'exécution va considérablement augmenter, puisque nous devons instancier le driver à chaque fois.

Voyons donc comment tout ceci fonctionne.

V-A - Création des pages

Une première approche nous conduirait à créer trois classes, Page1, Page2 et Page3. Cependant, nous savons que ces pages ont beaucoup en commun, dont l'en-tête et le pied de page, ainsi que le titre. Elles peuvent donc dériver d'une classe commune, BasePage.

Essayons de voir à quoi peut ressembler cette classe de base :

```
public abstract class BasePage {

    // //////////////////////////////////////
    // HEADER

    private WebElement headerTitle;
    private WebElement englishFlag;
    private WebElement frenchFlag;

    // //////////////////////////////////////
    // BODY

    private WebElement pageTitle;

    // //////////////////////////////////////
    // FOOTER

    private WebElement previousPage;

    // getters et setters

}
```

Nous avons rendu cette classe abstraite, car elle n'a pas vocation à être utilisée telle quelle. Elle ne représente qu'une partie de la page, et doit donc être complétée. Nous pouvons bâtir les classes représentant nos pages de la même manière, mais cela n'est pas vraiment satisfaisant. Nous devons toujours récupérer les divers éléments de la page, et les y placer avec les setters. Une meilleure solution serait de les créer dans le constructeur, qui aurait alors besoin du driver utilisé. Mais FluentLenium nous offre une solution encore plus élégante : des annotations.

De plus, telle qu'elle est, les éléments sont bien dans la page, mais leur vérification est toujours à l'extérieur. En laissant la page vérifier elle-même, elle peut être réutilisée dans plusieurs tests.

Pour utiliser les annotations, il suffit pour ceci que notre classe hérite de **FluentPage**, et lors de son instantiation, tous les attributs de type **FluentWebElement** seront recherchés sur la page. Par défaut, la recherche se base sur la correspondance entre le nom de l'attribut Java et la valeur de l'attribut id ou name de l'élément de la page Web. Il est aussi possible d'utiliser l'annotation **@FindBy**. Voici donc notre nouvelle classe de base :

```
public abstract class BasePage extends FluentPage {

    private static final String PAGE_TO_LOAD_TIMEOUT = "250";
    protected static Locale locale = Locale.FRENCH;

    public static void resetLocale() {
        locale = Locale.FRENCH;
    }

    private static final String[] pagesFR = { "", "page une", "page deux", "page trois" };
    private static final String[] pagesEN = { "", "page one", "page two", "page three" };

    // ////////////////////////////////////////
    // HEADER

    @FindBy(tagName = "h1")
    private FluentWebElement headerTitle;
    @FindBy(id = "headerForm:english_button")
    private FluentWebElement englishFlag;
    @FindBy(id = "headerForm:french_button")
    private FluentWebElement frenchFlag;

    // ////////////////////////////////////////
    // FOOTER

    @FindBy(id = "previousPage")
    private FluentWebElement footerText;
    private Selenium selenium;

    // ////////////////////////////////////////

    @Override
    public String getBaseUrl() {
        return "http://127.0.0.1:8080/tutoselenium/";
    }

    public void clickOnFrenchFlags() {
        locale = Locale.FRENCH;
        frenchFlag.click();
        waitPageToLoad();
    }

    public void clickOnEnglishFlags() {
        locale = Locale.ENGLISH;
        englishFlag.click();
        waitPageToLoad();
    }

    protected void checkHeader() {
        assertThat(headerTitle.getText(), is(locale == Locale.FRENCH ? "En tête" : "Header"));
        String url = buildUrl(getBaseUrl(), "drapeau_anglais.png");
        assertThat(englishFlag.getElement().getCssValue("background-image"), is(url));
        url = buildUrl(getBaseUrl(), "drapeau_francais.png");
        assertThat(frenchFlag.getElement().getCssValue("background-image"), is(url));
    }

    protected void checkFooter(int previousPage) {
        String page = "";
        if (locale == Locale.FRENCH) {
            page = pagesFR[previousPage];
        } else {

```

```

        page = pagesEN[previousPage];
    }
    assertThat(footerText.getText(), is(locale == Locale.FRENCH ? "Vous venez de la " + page
                                                                    : "You are coming from " +
page));
}

protected boolean hasPreviousPage() {
    FluentWebElement previousPage = getPreviousPageElement();
    if (previousPage == null) {
        return false;
    }
    return previousPage.isDisplayed();
}

private FluentWebElement getPreviousPageElement() {
    FluentList<FluentWebElement> elements = find("#previousPage");
    if (!elements.isEmpty()) {
        return elements.get(0);
    }
    return null;
}

protected void waitPageToLoad() {
    if (selenium == null) {
        selenium = new WebDriverBackedSelenium(getDriver(), getBaseUrl());
    }
    selenium.waitForPageToLoad(PAGE_TO_LOAD_TIMEOUT);
}

private String buildUrl(String baseUrl, String resource) {
    return "url(\"" + baseUrl + "faces/javax.faces.resource/" + resource + "?ln=images\"");
}
}

```

Nous avons créé deux méthodes, `checkHeader()` et `checkFooter()`, destinées à être appelées par les classes concrètes représentant les pages. Nous avons ensuite des méthodes publiques, pour « cliquer » sur les drapeaux. Quant à la méthode `getBaseUrl()`, elle doit renvoyer l'URL de base, pour toutes les URL relatives.

Mais examinons plus attentivement la méthode `hasPreviousPage()`. Nous aurions pu utiliser l'attribut `footerText`, plutôt qu'une variable locale. Si nous avons fait ainsi, c'est parce que l'élément n'est pas toujours présent sur la page. La classe `FluentWebElement` agit comme un proxy, et n'est jamais nulle, car initialisée à l'instanciation de la classe par le biais de son annotation. Mais quand nous demandons à vérifier l'absence de l'élément, elle ne le trouve pas (et pour cause), et nous obtenons une exception :

```

org.openqa.selenium.NoSuchElementException: Unable to locate element:
{"method":"name","selector":"previousPage"}

```

Nous sommes obligés de le rechercher seulement lors de son utilisation, et pas avant. Ce problème ne se pose pas lors de la vérification du texte de l'élément.

Voyons maintenant le code de la page une :

```

public class Page1 extends BasePage {

    // ////////////////////////////////////////
    // BODY

    @FindBy(id = "contentForm:pageTitle")
    private FluentWebElement pageTitle;
    @FindBy(className = "ui-panel-title")
    private FluentWebElement panelTitle;
    @FindBy(id = "contentForm:label")
    private FluentWebElement label;
    @FindBy(id = "contentForm:pageText")
    private FluentWebElement textField;
}

```

```
@FindBy(id = "contentForm:pageError")
private FluentWebElement errorMessage;
@FindBy(id = "contentForm:nextPage")
private FluentWebElement button;

public void setNextPage(int page) {
    textField.clear();
    textField.text("" + page);
}

public void buttonClick() {
    button.click();
    waitPageToLoad();
}

@Override
public void isAt() {
    assertThat(pageTitle.getText(), is(locale == Locale.FRENCH ? "Page une" : "Page one"));
}

public void checkPage(boolean errorExpected, int previousPage) {
    checkHeader();
    checkBody(errorExpected);
    assertThat(hasPreviousPage(), is(true));
    checkFooter(previousPage);
}

public void checkPage(boolean errorExpected) {
    checkHeader();
    checkBody(errorExpected);
    assertThat(hasPreviousPage(), is(false));
}

public void gotoPage(int page) {
    setNextPage(page);
    buttonClick();
}

private void checkBody(boolean errorExpected) {
    assertThat(panelTitle.getText(), is(locale ==
Locale.FRENCH ? "Choix de la prochaine page"
: "Select the new page"));
    assertThat(label.getText(), is(locale == Locale.FRENCH ? "Numéro de la prochaine page"
: "Number of next page :"));
    assertThat(button.getText(), is(locale == Locale.FRENCH ? "Page suivante"
: "Next page"));
    String errorText = errorMessage.getText();
    assertThat(errorMessage.getText().equals(""), is(!errorExpected));
    if (errorExpected) {
        assertThat(errorText,
is(locale ==
Locale.FRENCH ? "Vous devez entrer une valeur entre un et trois."
: "You must enter a value between one and three"));
        assertThat(errorMessage.getElement().getCssValue("color"), is("rgba(255, 0, 0, 1)"));
    }
}
}
```

La méthode `isAt()` est dédiée à la vérification que nous sommes sur la bonne page, rien de plus. Son contenu est laissé à la libre appréciation de chacun. Pour notre part, nous vérifions le titre de la page.

Ensuite, nous avons une méthode `checkPage()`, qui prend en paramètres les valeurs qui peuvent changer, à savoir si nous attendons un message d'erreur, et quelle est la page précédente. Nous avons aussi quelques méthodes pour manipuler notre page.

Je ne présente pas les autres pages, vous aurez deviné qu'elles sont tout aussi simples.

Voyons donc maintenant ce que devient notre test.

V-B - La classe de test

Nous allons commencer par Firefox, avant de l'adapter aux autres drivers.

Là aussi, nous utiliserons FluentLenium, et notre test va hériter de **FluentTest** :

```
public class TutorielSeleniumTest extends FluentTest {

    private static final String PAGE_TO_LOAD_TIMEOUT = "250";

    private final WebDriver driver;
    private String baseUrl;
    private Selenium selenium;

    @Page
    protected Page1 page1;
    @Page
    protected Page2 page2;
    @Page
    protected Page3 page3;

    @Override
    public WebDriver getDefaultDriver() {
        return driver;
    }

    @Before
    public void setUp() throws Exception {
        driver = new FirefoxDriver();
        baseUrl = "http://127.0.0.1:8080/tutoselenium/";
        selenium = new WebDriverBackedSelenium(driver, baseUrl);
        BasePage.resetLocale();
    }

    @After
    public void tearDown() throws Exception {
        driver.quit();
    }

    @Test
    public void page1to1WithErrorFrench() throws Exception {
        connect();

        page1.isAt();
        page1.checkPage(false);

        page1.setNextPage(4);
        page1.buttonClick();

        page1.isAt();
        page1.checkPage(true);
    }

    @Test
    public void page1to2French() throws Exception {
        ...
    }

    @Test
    public void page2to3French() throws Exception {
        ...
    }

    @Test
    public void page3to1French() throws Exception {
```



```

    ...
}

// Les mêmes, en anglais

private void connect() {
    goTo(baseUrl + "faces/page1.xhtml");
    selenium.waitForPageToLoad(PAGE_TO_LOAD_TIMEOUT);
}

private void connectAndGo(int page) {
    connect();
    if (page != 1) {
        page1.gotoPage(page);
    }
}
}

```

Nous n'avons pas à nous occuper d'instancier nos pages, l'annotation `@Page` et la classe `FluentTest` s'en chargent pour nous. Cependant, pour ceci il est impératif que nous disposions du driver à utiliser, d'où la méthode `getDefaultDriver()` qui sera appelée lors de l'instanciation de nos pages.

Nous avons découpé notre test en petites séquences indépendantes. Nous commençons nos tests par `connect()`, ou `connectAndGo()` pour se rendre directement sur une page. Ensuite, chaque test suit la même structure. Du coup, tout devient beaucoup plus simple et lisible.

V-C - Autres drivers

Nous devons exécuter notre test à l'identique quatre fois, une fois par driver. Pour ceci, nous utiliserons les tests paramétrés de JUnit. Ensuite, nous savons que nos drivers ont quelques différences minimales. Le problème est que nos pages ne connaissent pas le driver en cours d'utilisation. Elles pourraient, mais ce n'est pas vraiment dans leur responsabilité. Elles sont conçues pour représenter les pages et interagir avec elles, rien de plus. Nous pourrions les spécialiser, en créant une version pour chaque driver, qui hériterait de nos pages. Ce qui multiplierait le nombre de classes au-delà du raisonnable : avec nos trois malheureuses pages, nous nous retrouvons avec 12 classes à créer ! À la place de l'héritage, nous allons donc utiliser la délégation.

```

@RunWith(Parameterized.class)
public class TutorielSeleniumTest extends FluentTest {

    private static final String PAGE_TO_LOAD_TIMEOUT = "250";

    @Parameters(name = "{0}")
    public static Collection<Object[]> data() throws IOException {
        return Arrays.asList(new Object[][] { { HTML_UNIT, new HtmlUnitDelegate() },
        { FIREFOX, new FirefoxDelegate() },
        { OPERA, new OperaDelegate() },
        { CHROME, new ChromeDelegate() } });
    }

    private final WebDriver driver;
    private final BrowserDelegate delegate;
    private String baseUrl;
    private Selenium selenium;

    public TutorielSeleniumTest(WebDriverFactory.Type driverType, BrowserDelegate
delegate) throws InterruptedException {
        super();
        this.driver = WebDriverFactory.getDriver(driverType);
        this.delegate = delegate;
        createPage(Page1.class);
    }

    @Page
    protected Page1 page1;

```

```
@Page
protected Page2 page2;
@Page
protected Page3 page3;

@Override
public WebDriver getDefaultDriver() {
    return driver;
}

@Before
public void setUp() throws Exception {
    baseUrl = "http://127.0.0.1:8080/tutoselenium/";
    selenium = new WebDriverBackedSelenium(driver, baseUrl);
    BasePage.resetLocale();
    page1.setDelegate(delegate);
    page2.setDelegate(delegate);
    page3.setDelegate(delegate);
}

...
}
```

Concernant les tests paramétrés, leur principe est très simple : deux annotations suffisent. La première sur la classe, pour dire que le test est à exécuter plusieurs fois avec différents jeux de paramètres, et une autre pour indiquer la méthode produisant ces jeux de paramètres.

Examinons le premier paramètre. Il ne s'agit pas du driver lui-même, mais d'une enum qui nous donne son type. Le driver sera instancié à travers une factory. Si nous faisons ainsi, c'est simplement que le driver doit être instancié au début de chaque test, et non au début de l'ensemble des tests.

Concernant le second paramètre, le délégué, il s'agit d'une interface reprenant les méthodes qui divergent d'un browser à l'autre, avec une implémentation spécifique à chacun. Ceci nous permet en plus de réutiliser ces délégués pour d'autres tests.

Je ne présente pas ici le code de la factory ou des délégués. Ils sont suffisamment simples pour ne pas demander d'explication, et vous les trouverez dans l'archive du projet en fin d'article. Je pense que vous en aurez compris le principe.

VI - Tests d'intégration automatisés

Si nos tests s'exécutent dans Eclipse (ou dans tout autre IDE à votre convenance), ils ne sont pas encore totalement automatisés : nous devons lancer Tomcat, puis les exécuter à la main. Voyons comment donc obtenir une automatisation complète avec Maven. Nous allons les placer dans la phase d'intégration, ce qui permet de bien les distinguer des tests unitaires, tout en testant l'ensemble des composants de notre application, même si la nôtre n'en utilise pas beaucoup.

VI-A - Le plugin Maven Jetty

Pour nos tests, nous allons remplacer Tomcat par le plugin Jetty, que nous pouvons lancer avant les tests, et arrêter à l'issue.

```
<properties>
...
<jetty.port>8080</jetty.port>
<jetty.host>127.0.0.1</jetty.host>
</properties>

<build>
...
<plugins>
```

```

...
<!-- Lancement et arrêt de jetty pour les tests d'intégration -->
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>8.1.8.v20121106</version>
  <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
    <stopKey>foo</stopKey>
    <stopPort>9999</stopPort>
    <connectors>
      <connector
        implementation="org.eclipse.jetty.server.nio.SelectChannelConnector">
          <host>${jetty.host}</host>
          <port>${jetty.port}</port>
          <maxIdleTime>60000</maxIdleTime>
        </connector>
      </connectors>
    </configuration>
    <executions>
      <execution>
        <id>start-jetty</id>
        <phase>pre-integration-test</phase>
        <goals>
          <goal>run</goal>
        </goals>
        <configuration>
          <scanIntervalSeconds>0</scanIntervalSeconds>
          <daemon>true</daemon>
        </configuration>
      </execution>
      <execution>
        <id>stop-jetty</id>
        <phase>post-integration-test</phase>
        <goals>
          <goal>stop</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
  ...
</plugins>
...
</build>

```

Nous lançons Jetty à la phase pre-integration-test, pour l'arrêter à la phase post-integration-test.

VI-B - Paramétrage des tests

Pour lancer les tests d'intégration, nous ne pouvons pas utiliser le plugin Surefire des tests unitaires : au premier échec, le build s'arrête, et Jetty continuerait de tourner. Nous allons donc utiliser le plugin Failsafe.

```

<build>
  ...
  <plugins>
    ...
    <!-- Exclusion des tests selenium de surefire -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.12.4</version>
      <configuration>
        <excludes>
          <exclude>*/selenium/**/*.java</exclude>
        </excludes>
      </configuration>
    </plugin>
    ...
  </plugins>
  ...
  <!-- tests selenium avec failsafe -->

```

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>2.12.4</version>
  <configuration>
    <reportsDirectory>${basedir}/target/surefire-reports</reportsDirectory>
    <includes>
      <include>**/selenium/**/*.*.java</include>
    </includes>
    <systemPropertyVariables>
      <base.url>http://${jetty.host}:${jetty.port}</base.url>
      <jetty.port>${jetty.port}</jetty.port>
      <jetty.context>${project.artifactId}</jetty.context>
    </systemPropertyVariables>
  </configuration>
  <executions>
    <execution>
      <id>integration-test</id>
      <goals>
        <goal>integration-test</goal>
      </goals>
    </execution>
    <execution>
      <id>verify</id>
      <goals>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
...
</plugins>
...
</build>

```

Pour Surefire, nous excluons tous les tests des packages comprenant selenium dans leur chemin. Inversement, pour Failsafe, nous ne prenons que ces mêmes tests. On déplace donc notre test dans le package `fr.atatorus.tutoselenium.selenium`. Nous ajoutons aussi quelques variables système que nous allons utiliser dans notre test.

VI-C - Adaptation de notre test

Nous devons modifier notre classe de test. En effet, même avec la balise `<contextPath />` pour configurer Jetty, je n'ai pas réussi à lui faire prendre en compte la racine « `/tutoselenium` » dans l'URL de base de notre appli. Mais nous avons besoin de cette racine si nous voulons lancer nos tests manuellement avec Tomcat. Pour des raisons similaires, nous devons ajouter « `faces/page1.xhtml` » à l'URL de base quand nous ouvrons notre page, sinon Jetty nous affiche simplement le contenu du répertoire...

Voici donc les adaptations nécessaires :

```

package fr.atatorus.tutoselenium.selenium;

public class SeleniumTest {
  ...

  @Before
  public void setUp() throws Exception {
    Properties properties = System.getProperties();
    baseUrl = properties.getProperty("base.url", "http://127.0.0.1:8080/tutoselenium");
  }

  ...
}

```

Ceci fait, nous pouvons lancer nos tests depuis une console avec :

```
mvn clean post-integration-test
```

Et vous aurez le plaisir de voir s'ouvrir et refermer les navigateurs, et d'admirer le défilement des pages.

VII - Liens

Voici quelques liens qui m'ont permis de trouver de l'aide pour la rédaction de cet article :

- le site officiel de Selenium : <http://seleniumhq.org/>;
- le javadoc de Selenium : <https://selenium.googlecode.com/git/docs/api/java/index.html>;
- le site officiel de FluentLenium : <https://github.com/FluentLenium/FluentLenium>, sur lequel on trouve une explication du pattern Page Object ;
- le wiki d'aide du driver pour Chrome : <https://code.google.com/p/selenium/wiki/ChromeDriver>;
- le wiki pour le driver d'Opera : <https://code.google.com/p/selenium/wiki/OperaDriver>;
- celui pour Firefox : <https://code.google.com/p/selenium/wiki/FirefoxDriver>
- et pour IE : <https://code.google.com/p/selenium/wiki/InternetExplorerDriver>;
- Xebia blog, où j'ai trouvé un article expliquant comment lancer les tests Selenium lors des tests d'intégration : <http://blog.xebia.fr/2011/02/18/automatiser-les-tests-selenium-avec-maven/>;
- Le projet complet du tutoriel pour Eclipse : <http://atatorus.developpez.com/tutoriels/java/test-application-web-avec-selenium/tuto-selenium-complet.zip>.

VIII - Conclusion

J'espère que cet article vous aidera à implémenter facilement les tests de vos applications Web avec Selenium. Ce n'est pas si compliqué que ça en a l'air. En fait, la principale difficulté provient des différences entre chaque navigateur. Nous avons vu que nous avons été obligé de réécrire certains bouts de code pour tenir compte de chaque navigateur. Même si ces différences se rencontrent principalement, du moins dans notre exemple, au niveau des styles CSS, il n'est pas dit que le problème ne surgira pas un jour ailleurs. Et je vous laisse le plaisir de jouer avec Internet Explorer. Il n'y a pas de raisons que Microsoft n'ait pas lui aussi ses petites différences ici ou là...

La mise en œuvre du pattern Page Object nous permet d'écrire et maintenir facilement nos tests. Quant aux tests paramétrés, ils nous simplifient la tâche d'écriture pour nos tests qui sont identiques (ou presque) d'un driver à l'autre.

Cependant, il est à noter que le découpage en de multiples petits tests a pour inconvénient la durée des tests. Dans le cas de notre petite application, le test complet dure plus de quatre minutes, alors qu'en écrivant un test pour dérouler une seule séquence, cette durée tombe à moins d'une minute. Dans le cas d'une application réelle, c'est tout à fait un argument qui justifie de passer les tests la nuit.

IX - Remerciements

Je tiens à remercier pour leur aide, que ce soit sous forme d'encouragement, de critiques ou autre, **zoom61**, **Lana.Bauer**, **mlny84**, **Nemek**, **keulkeul**, **alain.bernard**, **thierryler** qui m'a fait découvrir le pattern Page Object, et Mathilde Lemée pour ses conseils.

Je n'oublie pas non plus **Claude Leloup** pour ses corrections et remarques sur les subtilités de la langue française.