Apache OFBiz® Selenium-WebDriver

Version unspecified

Table of Contents

Ce projet est utilisé afin d’ecrire et d’exécuter des tests de l’interface utilisateur et des tests de cas d’usage métier.

1. Objectifs tests selenium

Il existe plusieurs types de test bien différent réalisable avec selenium-webdriver :

- les tests de l'interface utilisateur unitaires, appelé test unitaire;
- les tests de cas d'usage, appelé test de type scénario;
- les tests de performance, assez proche des tests de type scenario mais avec des objectifs différents.

Ils ont chacun leurs contraintes et leurs objectifs.

1.1. Test unitaire

1.1.1. Objectifs

Chaque test doit tester une fonctionnalité bien identifiée.
Chaque test doit être systématique dans le test, si possible tester tous les boutons, tous les messages d'erreur, mais dans un cadre de base.
Le passage des tests unitaires doit être assez rapide, car ils doivent permettre de repérer les erreurs les plus évidentes.
Les données utilisés pour ces tests sont le plus souvent des données dédiées, insérées dans la base via le load ext-test. Il n'y a pas besoin que les données soit significative.

1.1.2. Contraintes

La correction d'une erreur sur un test unitaire doit être assez rapide car le périmètre du test est précis.
Un test unitaire n'a pas pour objet d'être utilisé par un test de type scénario, donc s'il doit y avoir des parties de code commune, il faut mettre dans une méthode séparée cette partie de code commune (attendre le besoin pour le faire).
A l'opposé, les tests de type scénario font appel à des méthodes simples d'action qui doivent être utilisé dans un test unitaire.
Le nombre de message de log est limité, c'est plutôt les messages d'assert qui permettent de comprendre rapidement où le test a échoué. C'est plutôt des messages de log que des showInfoPanel.

1.2. Test scénario

1.2.1. Objectifs

Enchaîner un ensemble d'action correspondant à un cas d'usage utilisateur, que ce soit au sein d'une même application ou multi-application (back - front - portail).
Le test doit se limiter à un cas d'usage, il faut faire 3 tests plutôt qu'un seul test mélangeant des cas.
Un cas d'usage peut être simple, exemple : validation d'un article, ou correspondre à l'explication de l'usage d'une notion, exemple : caractéristique article et front
Passé au ralenti, un test scénario peux servir de tutoriel, il faut donc avoir des showInfoPanel

explicite fonctionnellement.
Les données doivent être explicites et dédiées, les données de base devant exister au par-avant peuvent être chargées via un scénario dédié qui passe juste avant le lancement du test.

1.2.2. Contraintes

Un test scénario doit être constitué d'appel de méthodes utilisées par les tests unitaires.
Comme un test correspond à un cas d'usage, seul les fonctionnalités nécessaires sont testées.
Les messages de log sont surtout des showInfoPanel pour expliciter où nous nous trouvons dans le déroulement du cas d'usage. Les messages sont explicites concernant l'action à réalisé ou réalisé et les données. Bien faire la différence entre les showInfoPanel à destination de "fonctionnel" et les log à destination des "fonctionnel & technique".

1.3. Test performance

1.3.1. Objectifs

L'objectif n'est pas de tester si cela fonctionne mais si ça fonctionne suffisamment rapidement avec de multiple utilisateur.
Pouvoir être lancé de très multiple fois sur un même environnement.
Durer un certain temps afin qu'il soit possible de lancer de multiple instance du test sans que la première instance se termine avant que la dernière instance ne soit démarré.
Le test doit se limiter à un cas d'usage, mais pour respecter la notion de durée, il est possible pour un test d'enchaîner plusieurs fois le même cas d'usage.

1.3.2. Contraintes

Un test performance doit être constitué d'appel de méthodes utilisées par les tests scénario.
Les contraintes se trouvent surtout au niveau de l'infrastructure permettant de lancer multiple instance en même temps.

2. Installation

Le projet OfbSwd est autonome, il a besoin que ofbiz soit opérationnel afin de pouvoir le tester mais il peut l'être en local ou à distance.

Souvent les développeurs (technique ou fonctionel) travaillent en parallèle sur le développement de l'application et sur les tests associés, donc classiquement OFBiz et OfbSwd sont installé côte à côte.
Ce chapitre installation concerne surtout cette configuration.

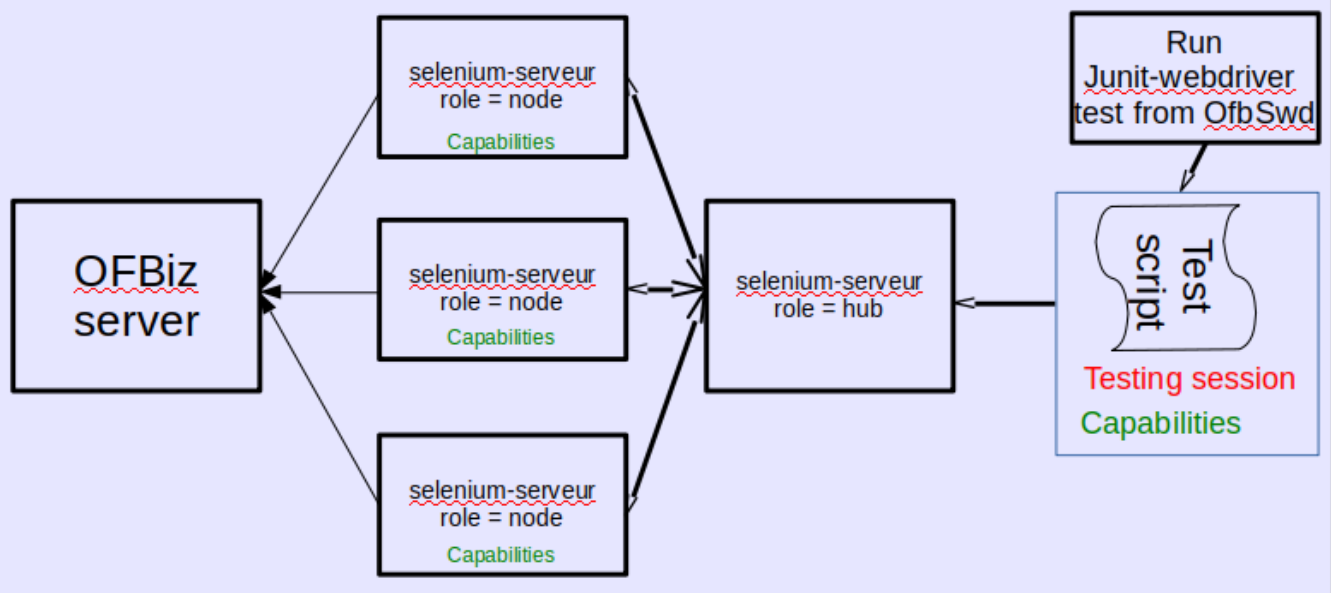
```
your developmnt directory/
 ├─── ofbiz-framework/       - Base directory for your Apache OFBiz installation
 ├─── OfbSwd/                - Base directory for OFBiz Selenium WebDriver
```

Normalement, il y a tout ce qu'il faut dans le projet OfbSwd, mais il est possible que, en fonction de la version ou du navigateur internet que vous utilisez, il soit necessaire de télécharger un driver adapté à votre environnement local (ou de test).

Dans ce projet nous utilisons les navigateurs Firefox et Chrome (ou chromium) dans leur dernière version.
Il est donc supposé que l'un des deux soit installé sur votre poste.

2.1. Lancer un test en local

Il y a plusieurs configuration possible, ce chapitre décrit la solution utilisé par défaut (grid).



Il est donc nécessaire d'utiliser 4 process
(donc 4 terminals quand on lance les process en interactif sur une seule machine) :

1. ofbiz afin de pouvoir le tester
2. hub : tools/hub.sh
3. node : (1 au minimum) tools/node.sh

4. OfbSwd pour lancer le test selenium : ./gradlew runSingleSelenium -PtestName=YOUR-TEST

Lors du lancement du test, le fichier selenium.properties est lu pour avoir les caractéristiques (Capabilities) de l'environnement de test : essentiellement Browser et sa version.

Avant de lancer le node, vérifiez les paramètres du fichier tools/node-conf-V3.json pour vérifier qu'ils correspondent à votre environnement (essentiellement les versions de navigateur). Quand le hub et le node ne sont pas lancé sur les même machines, il y a aussi les uri à corriger.

On linux debian environment, there is a package chrome-driver which is synchronised with package chrome to be sure to have the correct driver (on /usr/bin) for your chromium browser.

2.2. Driver pour les navigateurs

Les driver pour chrome et firefox se trouvent dans le répertoire tools/lib, si vous avez un soucis de compatibilité avec la version de votre navigateur, vous pouvez télécharger la bonne version (la dernière ;-))

- https://sites.google.com/a/chromium.org/chromedriver/downloads
- https://github.com/mozilla/geckodriver/releases for firefox

et pour tous les autres navigateurs, reportez vous à la page de référence selenium https://docs.seleniumhq.org/download/ in the chapter *Third Party Drivers, Bindings, and Plugins*

3. Bonnes pratiques

3.1. En synthèse

3.1.1. Les règles impératives sont, par ordre de priorité :

1. Un test doit être indépendant des autres tests
2. Tout développement d'une nouvelle fonction doit être accompagné de son/ses tests
   - s'il y a des nouveaux services, il doit y avoir au minimum un test junit par service
   - s'il y a une interface utilisateur (portlet, écran, menu) il doit y avoir un test selenium unitaire
   - l'usage de la fonction doit être utilisé dans un test selenium scénario
3. Aucune donnée ne doit être dans les tests, toutes les données sont lu dans les fichiers data
   - toutes méthodes qui a besoin d'un ensemble de donnés doit recevoir en paramètre 1 ou plusieurs data-obj pas une liste de variable unitaire.
     L'objectif est que un fonctionnel puisse raisonner au niveau objet métier et pas champ à champ.
     - le nom du paramètre ou d'une variable de type DataObj est, par défaut, le name du data-obj utilisé.
   - chaque fois qu'une donnée d'un data-obj doit être unique, il faut utiliser le type indexed, ainsi lors de son usage il sera automatiquement suffixé d'un index fournit dans le selenium.properties
4. Chaque méthode java, utilisable dans un selenium scénario, doit avoir sa javadoc, lisible par un "fonctionnel" et lui permettant de pouvoir utiliser cette méthode dans un enchaînement d'appel de méthode.
5. Quand une méthode utilise une donné d'un data-obj, le nom des champs utilisés doit être documenté dans la javadoc, ou alors il doit exister un data-obj correspondant dans le fichier template data correspondant à la methode.
6. Un test doit pouvoir être lancé plusieurs fois sur le même environnement.
7. il faut mettre un showInfoPanel en début de chaque **méthode d'action** avec l'explication de ce qu'il va se passer dans une perspective tutoriel),

il faut que la page soit chargé pour que le showInfoPanel fonctionne, donc attention pour la méthode de test.

3.1.2. Les règles java impérative :

1. règle de nommage package, class, méthode et variable
2. javadoc
3. pas de warning

4. pas de tabulation.

3.1.3. Environnement de test :

- Il est conseillé de tester au moins 2 navigateurs, par exemple Chrome et Firefox, par defaut les tests lancés par jenkins sur la platforme ofbizextra, le sont avec Chrome, il est donc souhaitable de tester en local avec firefox
- Pour l'instant, les tests et démonstrations sont réalisé sur PC, nous avons choisi le format classique de 1368x768 (un peu moins à cause des barres à droite ou en haut 1365x765). La définition souhaité est paramétré dans selenium.properties, lors du lancement des tests en local selenium essaye de redimensionne la fenêtre, si c'est possible matériellement.

3.2. Règles d'or

1. Un test est toujours améliorable, on ne teste jamais assez, donc il faut doser juste.
2. Un nouveau développement = Un test
3. Une modification = Un test à créer ou modifier.
4. Il faut mettre une entête Javadoc à chaque Méthode, et bien référencer les méthodes, ainsi il suffit de lire la Javadoc pour comprendre tout ce que fais la méthode, et ses appels.
5. Création →Recherche → Affichage → Modification → Désactivation → Activation → Suppression

Exemple :

```
  /**
   * Tester les moyens de paiements
   * <ul>
   *   <li>Saisir un mode de paiement. ({@link #CreateBankAccount
CreateBankAccount})</li>
   *   <li>Modifier mode de paiement. ({@link #ModifBankAccount
ModifBankAccount})</li>
   *   <li>//TODO : desactiver mode de paiement</li>
   *   <li>supprimer mode de paiement. ({@link #DeleteBankAccount
DeleteBankAccount})</li>
   * <ul>
   *
   * @param partyId
   *
   * @throws Exception
   */
```

3.3. Signaler un travail non terminé, ou à faire

Utiliser la syntaxe javadoc permettant de signaler quelque chose non finie via //**TODO** et si jamais cela concerne une partie de code qui est mis en commentaire en attendant une validation de sa suppression, il faut ajouter la date et au besoin quand il faudra supprimer ce code (par exemple un mois après la mise en commentaire

```
//TODO description du truc à faire après le machin
```

3.4. Découpage des tests

Dans le cadre des classe de test, il faut séparer les méthodes de test selenium unitaire, à destination des développeur des méthodes d'usage à destination des scénario.

Les premières ont un objectif de test de bon fonctionnement de l'iinterface utilisateur : chaque bouton, les ajouts / suppressions les critères de recherche. Les éléments à tester sont choisi par le développeur.

Les secondes ont un objectif pratique, enchainer des actions, il faut bien sur, un peu de test mais vraiment le minimum car le scénario aura sa propre logique de test. Exemple: si je crée un article pour l'utiliser dans une commande, si la création échoue j'aurais forcément un soucis lors de la création de commande.
Dans le cas des scénario, les logInfoPanel servent à repérer facilement l'action qui n'a pas fonctionné.

Toute entité devra être testée avec un minimum de 5 méthodes et avoir les 4 dernière méthode disponible en mode usage:

- testEntite() : permettant de faire un test de l'entité en lançant les 4 autre méthodes.
- createEntite(…) : permettant de faire la création, retourne l'id crée (incluant aussi la vérification de la création).
- modifEntite(…) : permettant de faire la modification (incluant aussi la vérification de la modification).
- deleteEntite(…) : permettant de faire la suppression (incluant aussi la vérification de la suppression).
- searchEntite(…) : permettant de faire la recherche de l'entité via l'écran supérieure gauche, et qui sélectionne l'entité trouvée. (incluant aussi la vérification de l'affichage des bons boutons et des bonnes informations).

Les variables doivent être initialisées dans la première méthode, puis passées en paramètre aux autres, cela afin d'avoir des méthodes indépendantes et utilisables dans les scénarios directement.

S'il est possible d'activer, désactiver un objet métier, alors 2 autres méthodes seront à ajouter :

- activateEntite(…) : permettant de faire une activation.
- deactivateEntite(…) : permettant de faire une désactivation.
- La gestion de l'historique étant alors gérée/testée dans ces 2 méthodes.

Actuellement il y a encore peu de test UI Selenium, donc il est préférable de surtout faire des tests scénario afin de tester beaucoup de fonctianalité de OFBiz.
Avec les tests unitaires, les tests sont plus complet mais uniquement sur une seule entité.
En écrivant des tests scénario, il est nécessaire d'ajouter des methodes d'usage ou d'action, qui seront utilisés par la suite pour les tests unitaires.

3.5. showInfoPanel() et log()

3.5.1. showInfoPanel()

Cette méthode permet de générer des messages à l'attention d'un "utilisateur" qui regarderait l'enregistrement vidéo du déroulement d'un test. Ces messages apparaissent aussi dans le fichier de log.

Il y a deux principaux cas d'usage :

1. l'usage de la vidéo en tant que tutoriel, l'utilisateur ne connaît pas l'application, il faut donc des messages explicites de l'actions à venir.
2. l'usage des messages pendant le déroulement du test en local (ou en regardant la vidéo) pour visualiser et comprendre pourquoi il y a un échec, les messages doivent donc être explicites par rapport à la javadoc pour facilement comprendre où nous nous trouvons dans le test.

Toute entrée dans une méthode commencera par un signalement clair (cible utilisateur lambda + information informatique), si cela aide un des deux utilisateurs potentiel (cf cas d'usage ci-dessus), par exemple, il n'est pas nécessaire de préciser le début des méthodes "techniques".

Chaque fois que c'est possible indiquer les données utilisées pour l'action à venir (cela aide à reproduire le test) > LogSelenium.showInfoPanel(module, source, "Affichage des relations (" + partyId + ")", 1);

3.5.2. log()

Cette méthode permet de générer des messages dans le fichier de log à l'attention des "réalisateurs / débuggers" de test, qu'ils soient technique ou fonctionnel (ou les deux ;-).

Afin que les logs soit lisibles, il est important de mettre des messages à valeur ajoutée, c'est à dire permettant de comprendre l'avancement du test et au besoin avec des données utilisées ou lues par le test.
Il ne faut pas mettre trop de message non plus.
Dans le fichier de log, le message s'affiche sur une ligne où est indiqué le nom de la méthode où il est appelé, il n'est donc pas nécessaire de le rajouter dans le message.

```
LogSelenium.log(module,"id=new_list-name_" + offerId);
```

Resultat

```
  2016-07-11 11:28:034 front.DroitMembreFamille.addList(DroitMembreFamille.java:310)
:: id=new_list-name_OB10803
```

Les showInfoPanel peuvent perturber le déroulement d'un test, par exemple il ferme un drop-down ouvert, dans ce cas, il peut être nécessaire d'utiliser le log à la place.

Comme le fichier log peut être lu par des "fonctionnels" qui ne regarderont pas le code, il faut être en cohérence avec ce qui est mis dans la javadoc. Des méthodes purement techniques n'ont pas forcément besoin de générer des messages de log

3.6. Indexer les Id uniques

Pour des raisons pratiques et de rapidité de traitement d'une erreur, nous nous imposons de pouvoir lancer chacun des tests selenium plusieurs fois sur la même instance SGBD, donc il est important que si dans un test un objet fonctionnel (une commande, un article, un fournisseur, ...) est crée, les données qui doivent être unique ne soit pas le même dans le second lancement que dans le premier.

Pour cela nous avons choisi "d'indexer" ces données uniques, c'est à dire :

1. les données uniques sont constituées d'une base plus d'un suffixe, par exemple pour la création d'un utilisateur sur le front, il s'appellera selenium325@otoit.com la première fois et selenium326@otoit.com la seconde fois. Le chiffre ajouté étant, par défaut un numéro fournit en paramètre du job jenkins (buildNumber dans selenium.properties).
2. au niveau de la définition des données, dans les fichiers data (scenario / test-suite / test-case / data-obj) au niveau de la donnée dans un data-obj, il faut indiquer comme type "indexed" et non "string".
3. lors du lancement du test, au moment de la lecture des fichiers de data, toutes les données indexed sont suffixées avec la valeur du champ buildNumber du fichier selenium.properties (lors d'un lancement jenkins, c'est 1 ou le paramètre saisi manuellement lors du lancemenr du job qui est utilisé). Il n'y a donc rien à faire de particulier au niveau de l'usage de cette donnée.

```
  <data-obj name="login">
      <indexed name="username" value="selenium"/> <!-- indexé via un suffix :
selenium1, selenium2, ... -->
      <string name="password" value="ofbiz"/>
  </data-obj>
```

Dans les messages de log, il faut penser à indiquer la donnée avec l'index car c'est important pour bien rechercher l'erreur sur la bonne instance du déroulement de test. De la même manière, il peut être intéressant d'indexer des données pas nécessairement uniques mais qui seront pratique pour différentier les données entre deux passages (par exemple un libellé article ou le prénom).

4. Selenium avec OFBiz

Les outils ou méthodes décritent dans ce chapitre sont disponible dans le projet OfbSwd.

4.1. Test et données

Les données utilisées pour les tests doivent être externalisé dans les fichiers data (répertoire tests-data).

Les données de test sont gérées via des fichiers xml, avec une notion de scénario, lors du lancement d'un test le nom du scénario est lu dans le fichier selenium.properties : dataScenario=default

A l'interieur du scénario, le nom de la test-suite et du test-case doivent reprendre exactement les même nom que les tests souhaités.

4.2. Enregistrement de la vidéo

Le fait d'utiliser le jar grid-service-provier et non pas le selenium-server-standalone permet de pouvoir demande l'enregistrement video du test en cours d'execution. Il enregistre l'écran sur lequel le navigateur est lancé.

A la fin du déroulement du test, la vidéo se trouvera dans build/outputs avec le nom className-testMethodName.avi

Par défaut SimpleTestSuite (méthode dont hérite toutes les méthodes de test) déclenche l'enregistrement de la vidéo en fonction du paramètre record.video=yes du fichier selenium.properties.

D'autre part la méthode LogSelenium.showInfoPanel permet d'afficher un message à l'écran en court de test, cela a pour objectif d'expliquer à la personne qui regarde le test se dérouler, ce qu'il va se passer.
Quand le paramètre infoPanelEnabled=yes du fichier selenium.properties est positionné ces messages apparaissent. Comme il génére des temps de pause à chaque message, c'est rarement activé pour des tests du process d'intégration classique.
Le paramètre logPanelMessage=yes permet d'avoir ces messages aussi dans le log output.

5. Analyser une erreur

L'execution d'un test selenium génère une erreur, que faire pour touver le plus rapidement possible la cause de cette erreur et en particulier de quel partie provient elle : du test lui-même, des données de test, du code OFBiz ou des données de OFBiz.
Quand le test selenium est correctement écrit et que le process décrit ci-dessous est suivi, la plupart du temps l'analyse de l'origine du probléme est rapide.

Préambule : Quand un selenium "plante" c'est sûrement qu'il y a une erreur dans l'application ou dans les données, il ne faut pas commencer par chercher l'erreur dans le selenium.

A l'opposé, si l'erreur semble vraiment surprenante (genre, aucune modification depuis le dernièr execution réussie), ne pas hésiter à relancer le test une seconde fois pour valider que l'erreur est bien reproductible de manière simple.

5.1. Premier niveau d'analyse

5.1.1. Être dans une attitude fonctionnelle :

1. noter le nom du test qui échoue
2. aller voir la copie d'écran au moment de l'échec
3. lire le message d'erreur, clair ou technique
4. lire la pile d'appel des méthodes au moment de l'erreur
5. lire la javadoc du test
6. lire la javadoc des méthodes de la pile en partant de la dernière en remontant, jusqu'au moment où vous arrivez à situer dans quel situation se produit l'erreur
7. aller voir dans le log pour avoir les données du lancement (en particulier les variables indexed)
8. ouvrir le fichier de données utilisé par ce test
9. aller dans le site utilisée par ce test et essayer de reproduire la situation de l'échec
10. s'il y a besoin de plus de détail de contexte, lire le "détail" du log (au cas où !)
11. si vous n'arrivez pas à reproduire l'erreur ou ne comprenez pas pourquoi elle apparaît, allez expliquer le résultat de votre analyse à une personne avec des compétences techniques.

Chacun de ces points sont explicités dans la suite

5.1.2. lire le message d'erreur

Dans le job jenkins, quand vous cliquer sur le lien du test qui a échoué, le message d'erreur c'est la 1,2 (ou 3) premières lignes qui apparaissent sous <<Message d'erreur>>

Le message est

- **Clair** Le message est généré par le test et dans ce cas il est censé être explicite une fois le contexte retrouvé, si ce n'est pas la cas, il faudra une fois le problème résolu le changer pour

qu’il le devienne.

- **Technique** Très souvent le message est technique, car c’est un élément de la page qui n’est pas trouvé, bien comprendre ce message, au besoin ce le faire expliquer s’il y a un doute.

5.1.3. lire la pile d’appel des méthodes

Dans le paragraphe Pile d’exécution, la pile ressemble à quelque chose comme

pile d'exécution

```
...
normal, platform: LINUX, platformName: LINUX, rotatable: false, setWindowRect: true,
takesHeapSnapshot: true, takesScreenshot: true, unexpectedAlertBehaviour: ,
unhandledPromptBehavior: , version: 69.0.3497.92, webStorageEnabled: true,
webdriver.remote.sessionid: f5bcdc89cad05106f743fbc2ea7...}
Session ID: f5bcdc89cad05106f743fbc2ea771fb0
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at
sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.ja
va:62)
    at
sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccesso
rImpl.java:45)
    at java.lang.reflect.Constructor.newInstance(Constructor.java:423)
    at org.openqa.selenium.remote.ErrorHandler.createThrowable(ErrorHandler.java:214)
    at
org.openqa.selenium.remote.ErrorHandler.throwIfResponseFailed(ErrorHandler.java:166)
    at
org.openqa.selenium.remote.http.JsonHttpResponseCodec.reconstructValue(JsonHttpRespons
eCodec.java:40)
    at
org.openqa.selenium.remote.http.AbstractHttpResponseCodec.decode(AbstractHttpResponseC
odec.java:80)
    at
org.openqa.selenium.remote.http.AbstractHttpResponseCodec.decode(AbstractHttpResponseC
odec.java:44)
    at
org.openqa.selenium.remote.HttpCommandExecutor.execute(HttpCommandExecutor.java:158)
    at org.openqa.selenium.remote.RemoteWebDriver.execute(RemoteWebDriver.java:548)
    at org.openqa.selenium.remote.RemoteWebElement.execute(RemoteWebElement.java:276)
    at org.openqa.selenium.remote.RemoteWebElement.click(RemoteWebElement.java:83)
    at org.ofbizextra.ofbswd.test.SimpleTestSuite.click(SimpleTestSuite.java:540) ①
    at
org.ofbizextra.ofbswd.test.ExampleVuejsTestSuite.simpleTest(ExampleVuejsTestSuite.java
:34)  ①
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    ...
```

avec en haut la méthode qui a provoqué l'erreur et au fur et à mesure que l'on descend les méthodes qui ont appelé celle de la ligne précédente.

① Nous de nous intéresseront que aux méthodes de org.ofbizextra.xxx (xxx en fonction du composant testé)

Dans cet exemple on peut dire que simpleTest a appellé click qui a provoqué l'erreur (au plan fonctionnel)
sur le plan technique la méthode simpleTest est dans le fichier ExampleVuejsTestSuite.java et l'appelle à click est à la ligne 34 et la méthode click se trouve dans le fichier SimpleTestSuite.java et c'est l'exécution de la ligne 540 qui a provoqué l'erreur via l'appelle à ….

5.1.4. lire la javadoc

la javadoc se trouve là
Pour aller directement sur la javadoc de la méthode désirée utilisez index (dans le menu en haut), puis vous pouvez suivre les liens.

Dans un test, quand des données sont utilisées (un offerId, un loggin, …), c'est normalement indiqué dans la javadoc qui précise le data-obj (cf paragraphe suivant) qui est utilisé (avec une notation de type xxxxx.yyyy xxxx étant le test-case et yyyy le data-obj).

5.1.5. voir le log du test

Dans l'écran où vous avez regardé, le message d'erreur (section Message d'erreur) et la pile des appels des méthodes (section Pile d'exécution), il faut descendre jusqu'à la section **Sortie standard** puis trouver le log concernant votre test, car la sortie concerne tous les tests du job.
Si vous avec utilisez "Gradle selenium Result", il y a un bouton "Standard Output" pour aller directement dedans.

Pour trouver le début du log du test qui vous intéresse, cherchez avec le nom du test

```
: **** Start test : NomDuFichier.NomDuTest
```

Ce log est censé être compréhensible y compris dans une optique fonctionnelle, au fur et à mesure de son usage il est important d'améliorer son contenu en ayant des messages précis et avec l'affichage de données en cours permettant de se retrouver facilement dans le déroulement du test.

5.1.6. fichier de donnée utilisé

Toutes les données saisies ou à vérifier, dans les tests sont dans des fichiers externes aux tests, il y a normalement/classiquement un fichier par package (c'est une convention de nommage pas une obligation technique) au format xml et dans chaque fichier on va trouver au minimum 4 niveaux hiérarchiques :

1. scenario ⇐ c'est le paramètre fournie pour les data à utiliser quand on lance un selenium
2. test-suite ⇐ correspond à la suite de test
3. test-case ⇐ correspond à un cas de test
4. data-obj (qui peut aussi contenir des data-obj) ⇐ correspond à un objet fonctionnel
5. donnée (string, date, integer, double)

```
  <?xml version="1.0" encoding="UTF-8"?>
  <testdata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="http://ofbiz.apache.org/dtds/testcasedata.xsd">
      <version>1.0</version>
      <scenario name="job-jenkin"> <!-- c'est le paramètre fournie pour les data à
utiliser quand on lance  un selenium -->
          <test-suite name="suite de test">
              <test-case name="méthode de test">
                  <data-obj name="objet"> <!-- un objet fonctionnel : une offre, un
utilisateur, ...  -->
                      <string name="maChaine" value="la valeur"/>
                  </data-obj>
              </test-case>
          <test-suite>
      </scenario>
  </testdata>
```

Les fichiers data se trouve dans le répertoire du projet OfbSwd dans le sous-répertoire test-data.

Beaucoup d'Id sont générés par le système (partyId, N° de commande, ...), donc pour retrouver les id générés lors du test afin de pouvoir les visualiser dans l'application (s'il n'y a pas de critère de recherche simple) il est nécessaire de lire les log, normalement si un id est utilisé suite à sa création il doit être affiché dans un des message de log.

5.1.7. aller sur le site testé

Il existe plusieurs environnements de test, 1 par version d'ofbiz de test :

- 13.07 : https://ofbiz13-07-selenium.ofbizextra.org/example/control/main
- trunk : https://ofbiz-selenium.ofbizextra.org/example/control/main

5.2. Trucs & Astuces

5.2.1. Lancement plusieurs fois du job

Par défaut, comme un test est dépendant des données en base et de son jeu de données associé, il ne peut pas être rejoué une seconde fois dans le même environnement avec les mêmes données. Cette contrainte nous semble inacceptable, aussi nous nous imposons une bonne pratique pour la dépasser.
Dans les bonnes pratiques, nous avons implémenté le fait d'indexer les données essentiel de manière à pouvoir lancer un test plusieurs fois sur le même environnement.
Pour cela, il faut que le job est un paramètre buildNumber qui permette de le modifier en relancant le job?

En pratique cela veux dire que pour certaine données, il ne faut pas seulement lire le fichier de données, il faut aussi lire le log pour connaître l'index utilisé pour ce test.
En effet, si par exemple, le "nom" doit être unique, dans le fichier data "nom" sera de type indexed

et aura comme valeur (par exemple) Martin, et lors de l'execution si le buildNumber est 2 c'est Martin2 qui sera utilisé pour le test.

6. Tests Selenium (ex-webhelp)

6.1. Ecrire des tests fonctionnels (seleniums-webdriver)

Il y a de multiple manières d'écrire des tests de non régression, ce chapitre décrit comment écrire des tests pour les outils sélénium avec Webdriver de manière à pouvoir les exécuter via un serveur de non régression (de type jenkins) au travers de la solution grid de selenium.

Afin d'écrire, d'exécuter en local, et de mettre au point un test, il est nécessaire d'installer l'addon org.ofbizextra.framework.test.webdriver-tools. L'aide de cet addon contient l'ensemble des éléments à installer sur son poste.

Le navigateur par défaut avec selenium est firefox 28.0, pour lequel il existe des outils d'enregistrement de scénario. Nous en conseillons deux, qui sont similaire :

- selenium ide, outil le plus ancien et qui est le plus convivial dans un premier temps, il permet de mieux tester les expressions xpath (Téléchargement sur http://docs.seleniumhq.org/download/)
- selenium builder, outil adapté à la rédaction de test en java. (Téléchargement sur https://saucelabs.com/builder)

Avec ces deux outils, après l'enregistrement d'un test, il faut l'exporter en java pour le compléter. Avec selenium ide exportez en ofbiz / webdriver et avec selenium builder exportez en java

Il est possible d'utiliser les deux outils en même temps.

Cet aide n'est pas un tutoriel mais plutôt un pense bête des principales commandes utilisés.

La règle majeure à avoir à l'esprit lors de la réalisation d'un test, c'est qu'un script se déroule à vitesse machine, et il faut donc penser à gérer l'attente du résultat de toute action. C'est à dire après un « click », il faut dire comment repérer que le traitement demandé est fini, avant d'enchaîner sur la suite.

Pour un test Selenium, il faut donc :

- le serveur Selenium (après avoir récupéré le jar), et le lancer ainsi : java -jar /chemin_vers_le_jar/selenium-server-standalone-2.44.0.jar
- Aller dans le répertoire ofbiz, et initialiser les données (si pas déjà fait) : ./ant load-exttest
- Lancer ofbiz via ./ant start

Pensez aussi à configurer le fichier framework/testtools/selenium/resources/selenium.properties

```
#WebDriver type : local, htmlUnit or grid
testType=grid
#targetBrowser type : firefox, chrome
targetBrowser=firefox

#Address to call from selenium grid node to contact ofbiz default to
https://localhost:8443/
ofbizBaseUrl=https://localhost:8443/

#Address of the grid hub in most cases it will be http://localhost:4444/wd/hub/
gridHubUrl=http://localhost:4444/wd/hub/

#This address is used by ant to see if ofbiz is started or not
#only change the host name don't change protocol or port number or controller uri
ofbiz.started.test.url=http://localhost:8080/ordermgr/control/view

#the driver user for chrome browser ajsut depending or the server architecture 32 or
64 bit (both drivers under same directory)
#don't forget to start the node with with the following parameters if you are using
the grid mode
#java -Dwebdriver.chrome.driver=<full-path>chromedriver-64 -jar selenium-server-
standalone-2.42.2.jar -role node

webdriver.chrome.driver=resources/chromedriver-64

#two tag for helping lisibility of webdriver job output
# first one to show (or not) user message during the test. These message are done with
showInfoPanel method
# second one to show message in log (first one should be to yes if this one is yes)
# be careful, with yes, the tests will be more longer because showInfoPanel included a
wait for human being able to read the message
infoPanelEnabled=yes
logPanelMessage=yes
#define the scenario to be loaded and provide data for test cases
#data can be assigned for test-cases with scenario.
#scenario can be splitted in many files
dataScenario=test_tnr
record.video=yes
```

6.2. Lancer un test Selenium

Le test Selenium se lance en ligne de commande en se plaçant dans le répertoire ofbiz

exemple : ./ant run-one-webdriver-test -DtestName=PartyMgmt

Cet exemple lance la classe PartyMgmt,java qui contient un test Selenium.

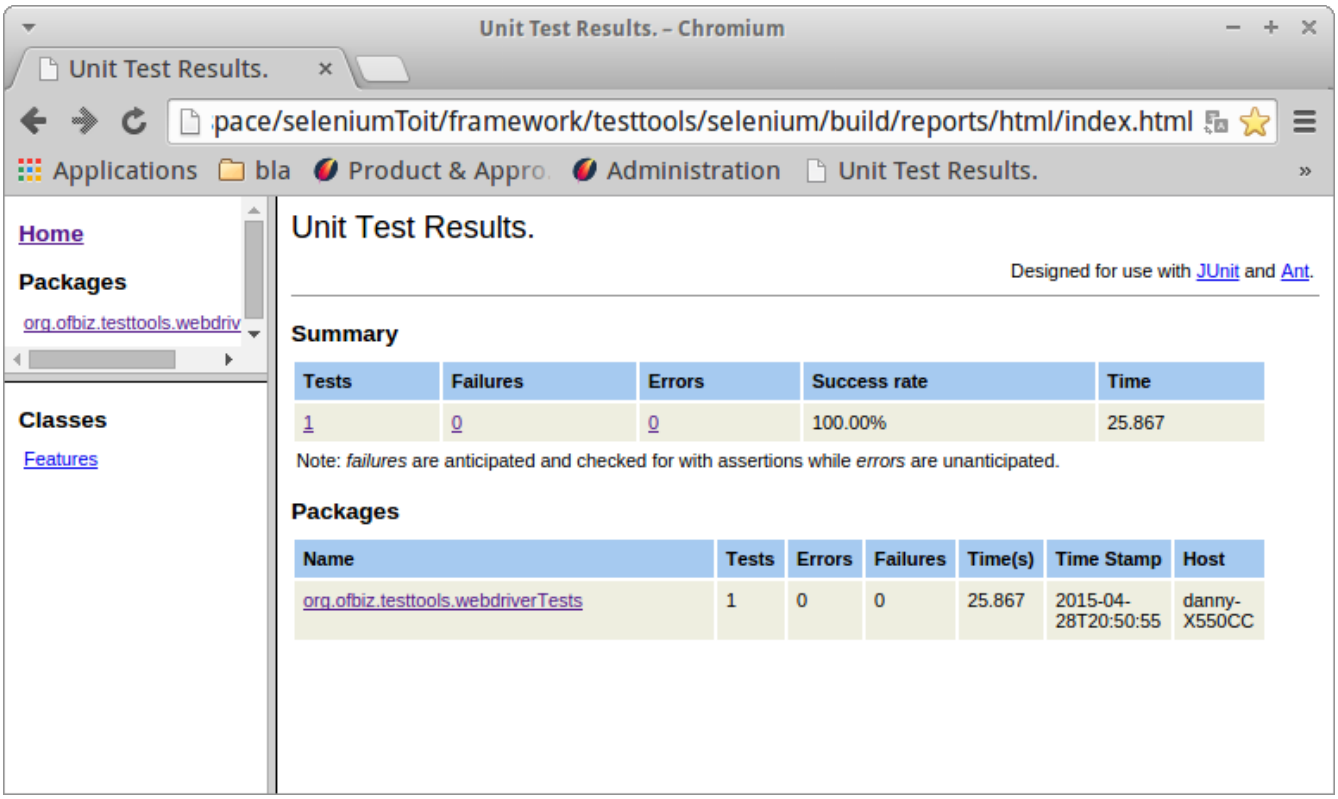
Ou pour une classe specifique

exemple : ./ant Selenium

Cet exemple lance les classes demandées via votre fichier SpecificTestRunner.java

6.3. Contrôler le résultat

On peut voir le résultat du test, et ainsi voir les éventuelles erreurs via le fichier index dans votre ofbiz à l'emplacement framework/testtools/selenium/build/reports/html/index.htm



6.4. Tips and Tricks

- Les principaux soucis que vous pouvez avoir, seront des soucis avec la synchronisation du navigateur et ajax, qu'il sera facile de contourner. Par exemple dans un écran de modification, il peut y avoir un délai «ajax» entre le clic sur «Modifier» et l'affichage suivant à contrôler, aussi, il faut penser à attendre l'écran suivant avant de poursuivre.

  Exemple 1

```
                click(By.id("ListFacilities_addFacility_a"));
                showInfoPanel("On va saisir l'emplacement " + facilityValue, 2);
                sendKeys(By.id("EditFacility_facilityId"), facilityValue);
                click(By.id("0_lookupId_button"));

driverWait.until(ExpectedConditions.presenceOfElementLocated(By.id("ListFacilities_
row0")));
                assertTrue(...
```

  Ici on peux voir de l'information utilisateur avec un ShowInfoPanel, mais aussi un driverWait

qui attend l'affichage suivant pour suivre les tests. *

- Un exemple de test classique : faire la création, ensuite faire la modification, en commentant la partie création, et ainsi de suite ; et quand tout est fini, vous n'avez qu'à dé-commenter.

  L'avantage de cette méthode est d'éviter de relancer tous les tests quand vous êtes entrain de développer.

- L'utilisation de méthodes, permet d'éviter de reproduire du code déjà fait, et ainsi d'optimiser la compréhension.
- Quand on prend un xpath de chromium, il met des doubles quotes («) , donc attention à les remplacer par des simple quotes ('), sachant qu'on peut aussi intégrer des « \ » pour que certaines ne soient pas interprétées.

```
assertTrue("Test1",driver.findElements(By.cssSelector("img[title=\"Test1\"]")).size
()>0);
```

- Sous Eclipse, il ne faut pas hésiter à utiliser l'intellisense, car documentation en anglais de toutes les fonctions, c'est pratique quand on ne connaît pas d'avance ce qu'on veut utiliser et que l'on doit chercher.

6.5. Tests normaux

- En pratique, voici le déroulement pour tester via 3 terminaux:

```
Tester via 3 terminaux

# lancement du selenium server standalone (Terminal 1)
        java -jar selenium-server-standalone-2.44.0.jar
# Régler Selenium
        framework/testtools/selenium/resources/selenium.properties
# Régler les classes à exécuter

framework/testtools/selenium/tests/org/ofbiz/testtools/projetSelenium/ProjetTestRun
ner.java
# Installer la base
        Exemple : psql -U ofbiz -h localhost -f ofbiz-prod.sql monofbiz2
# Lancer load pour test
        ./ant load-exttest
# Lancer Ofbiz (Terminal 2)
        ./ant start
# Lancer le test Selenium (Terminal 3)
        ./ant Selenium
# Les terminaux 1 & 2 ne bougeront plus, seul le Terminal 3 servira à relancer les
tests.
```

6.5.1. Tests Vidéos

- En pratique, voici le déroulement pour tester en ayant la vidéo

```
Pour tester et obtenir une vidéo (4 terminaux)

1) charger le jar
http://sourceforge.net/projects/ofbizextra/?source=typ_redirect
2) Dans le fichier properties, rajouter la vidéo
                record.video=yes

3) Lancer le hub (Terminal 1)
        java -jar grid-service-provider-20150704.jar -role hub
4) Dans le nœud (Terminal 2)
        java -jar grid-service-provider-20150704.jar -role node -hub
http://localhost:4444/grid/register

5) Lancer le serveur ofbiz (Terminal 3)
        ./ant start
6) Lancer le test Selenium (Terminal 4)
        ./ant Selenium

La vidéo se trouvera alors dans /framework/testtools/selenium/build/reports/outputs
```

6.6. Gestion des données scénarisées

- Gestion des données scénarisées

```
# Définir le scénario à utiliser via
framework/testtools/selenium/resources/selenium.properties ; ajouter le scénario à
utiliser
#define the scenario to be loaded and provide data for test cases
#data can be assigned for test-cases with scenario.
#scenario can be splitted in many files
dataScenario=test_tnr

# Ajouter un fichier scénario (peux importe le nom), avec ce scénario
par exemple : /framework/testtools/selenium/resources/tests-data/Projet-TNR.xml

# Dans le fichier scénario, mettre les données
<?xml version="1.0" encoding="UTF-8"?>
<testdata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://ofbiz.apache.org/dtds/testcasedata.xsd">
    <version>1.0</version>
    <scenario name="test_tnr">
        <test-case name="TestProjet">
            <data-obj name="objet">
                <string name="maChaine" value="la valeur"/>
            </data-obj>
        </test-case>
    </scenario>
</testdata>

pour récupérer la donnée :
     String user =
scenario.getTestCase("TestProjet").getDataObj("objet").getString("maChaine");

On sait que c'est test_tnr, grâce à ce qui a été précédemment déclaré dans le
fichier selenium_properties.
```

- Exemple pratique :

```
<?xml version="1.0" encoding="UTF-8"?>
<testdata xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://ofbiz.apache.org/dtds/testcasedata.xsd">
    <version>1.0</version>
    <scenario name="test_tnr">
        <test-case name="maClasse">
            <data-obj name="login">
                <string name="username" value="ser"/>
                <string name="password" value="ofbiz"/>
            </data-obj>
        </test-case>
        <test-case name="Features">
            <data-obj name="liste">
                <string name="desc" value="liste des chats"/>
            </data-obj>
        </test-case>
    </scenario>
</testdata>

# Dans vos classes de test, vous pouvez ainsi récupérer la donnée :

     String user =
scenario.getTestCase("maClasse").getDataObj("login").getString("username");
     String password =
scenario.getTestCase("maClasse").getDataObj("login").getString("password");

On récupère ici donc les valeurs qui sont présentent dans le scénario test_tnr,
pour la classe maClasse, et la méthode/regroupement login,

    String description =
scenario.getTestCase("Features").getDataObj("liste").getString("desc");
```