

Utilisation de GWT avec spring et hibernate

par hugo (<http://hugo.developpez.com>)

Date de publication : 01 décembre 2009

Dernière mise à jour :

Vous avez testé GWT et l'avez apprécié. Vous avez suivi avec attention l'ensemble des tutoriaux hello world sur le net mais désormais vous aimeriez démarrer un vrai projet avec des services de persistance, réutiliser vos frameworks préférés et éviter de réinventer la roue. Ca tombe bien, nous allons essayer à travers cet article d'aborder des concepts un peu plus avancés qu'un simple hello world.


Commentez cet article :









I - Introduction.....	3
II - Création du projet.....	3
II-A - Plugin eclipse.....	3
II-B - Plugin maven.....	4
III - Séparation des projets.....	6
III-A - Inclusion de source externe dans le module GWT.....	6
III-B - Le mode noServer.....	7
III-C - Le lancement de la partie serveur dans cargo.....	8
IV - Partie cliente.....	11
IV-A - Wrapper or not wrapper.....	11
IV-B - Comparatif.....	11
IV-C - SmartGWT.....	13
V - Spring.....	16
V-A - Installation des dépendances.....	16
V-B - Chargement de l'application context.....	16
V-C - Stratégies d'intégration.....	17
VI - GWT-SL.....	19
VI-A - Présentation.....	19
VI-B - Modification du pom.....	19
VI-C - Ajout de la servlet de dispatch.....	19
VI-D - Création d'un service RPC.....	20
VI-E - Exportation du service.....	22
VII - Spring-security.....	23
VII-A - Introduction.....	23
VII-B - Mise en oeuvre.....	24
VII-C - Le service d'authentification.....	24
VII-D - Le contexte Spring.....	26
VII-E - La partie cliente.....	27
VII-F - Sécuriser vos appels RPC.....	31
VIII - Hibernate.....	32
VIII-A - Problématique.....	32
VIII-B - Modèle.....	33
VIII-C - Configuration spring.....	36
VIII-D - Résultat.....	37
IX - Conclusion.....	39
IX-A - Remerciements.....	39
IX-B - Références.....	39

I - Introduction

J'ai beaucoup cherché sur internet des références, des exemples, des tutoriaux mais j'ai été régulièrement frustré par la simplicité des exemples qui reste relativement éloigné du monde réel. Le simple hello world est très vite frustrant lorsque vous voulez aborder la sécurisation de votre application, le build automatisé, le packaging, le debugging etc... L'exemple que nous allons développer au cours de cet article se veut plus complet afin d'aborder un grand nombre de pièges communs. Il est loin d'être parfait mais comporte quelques éléments intéressants qui, je l'espère, vous permettront d'en apprendre davantage et de partir du bon pied sur un projet plus complexe.


L'objectif de cet exemple est de construire un site avec une bannière horizontale, un bouton de login/logout et un tabpanel comportant une liste d'éléments éditables. Nous ajouterons un service permettant de consulter des éléments stockés côté serveur. Le tout sera protégé en lecture par un système de rôle.

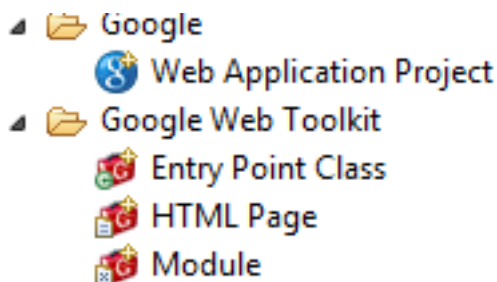
 *Librairies et versions utilisées pour les besoins de l'article :*

-  **GWT** : 1.7
-  **gwt-maven** : 2.0-RC1
-  **smartgwt** : 1.1
-  **Spring-security** : 2.0.4
-  **Spring core** : 2.5.6
-  **GWT eclipse plugin** : 1.0.0
-  **GWT-SL** : 0.1.5b
-  **Gilead** : 1.2.3

II - Création du projet

II-A - Plugin eclipse

J'aborderai ici l'utilisation d'eclipse puisque c'est l'IDE que j'utilise. Google a sorti un  **plugin pour eclipse** afin d'être capable de faire les actions de base :

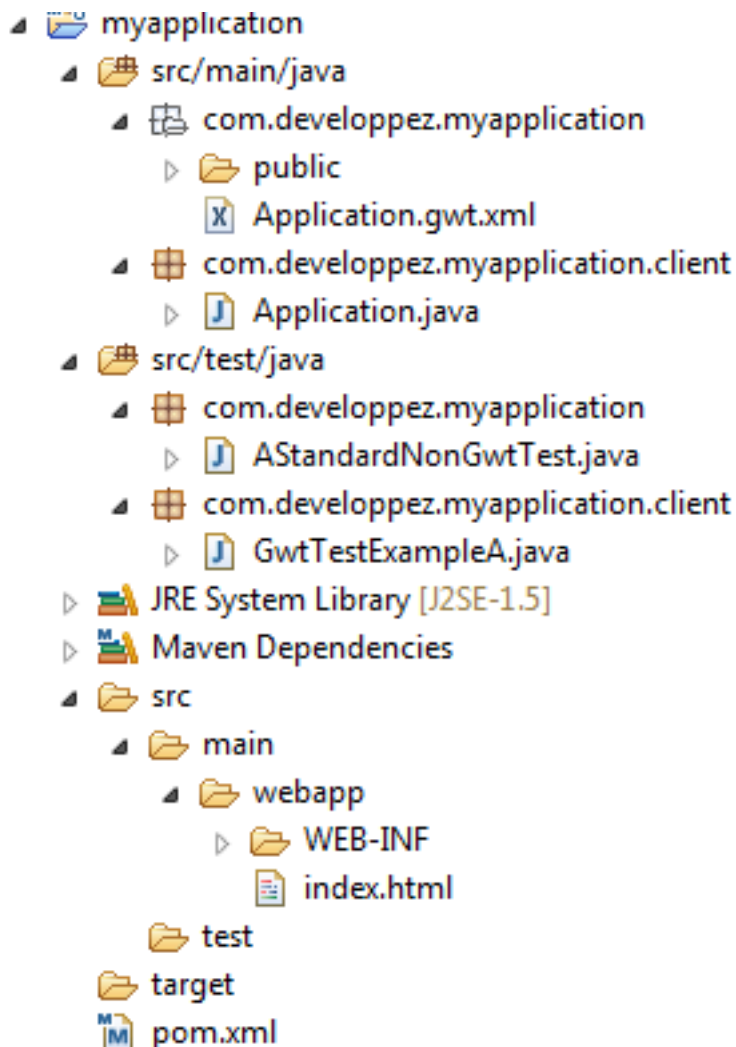


fonctionnalités du plugin eclipse pour GWT

Si le plugin est facile à prendre en main et plutôt efficace dans un premier temps, j'ai rapidement été ennuyé lorsque j'ai voulu modifier la structure du projet créé. Je n'ai pas trouvé de paramétrage permettant de changer l'emplacement du répertoire war par exemple et j'ai eu toutes les peines du monde à adapter mon projet lorsque j'ai voulu utiliser une arborescence "à la Maven". Je vous invite à essayer ce plugin mais je ne l'utiliserai pas par la suite, celui-ci n'étant pas compatible avec mes besoins. Gageons cependant que ce plugin s'améliorera dans le futur.

II-B - Plugin maven

Le plugin gwt-maven permet de démarrer assez simplement un projet GWT avec une arborescence Maven. Je le conseille aussi bien pour son build automatisé que pour le mode debug et la capacité à jouer les tests. Je m'éviterai la rédaction de son utilisation car un très bon tutorial existe déjà : [création d'un projet smartGwt avec maven](#). Le pom complet utilisé pour cet article se trouve cependant dans les sources de l'article. Pour indication, j'ai créé un projet avec groupId = com.developpez et artifact id = myapplication.



Arborescence créée par l'archétype gwt-maven

Je vais cependant m'attarder sur le pom produit par l'utilisation de l'archétype GWT. Prenons par exemple la configuration du plugin maven :

Configuration du plugin GWT

```
<plugin>
<groupId>com.totsp.gwt</groupId>
<artifactId>maven-googlewebtoolkit2-plugin</artifactId>
<version>2.0-RC1</version>
<configuration>
<compileTargets>
<value>com.developpez.myapplication.Application</value>
</compileTargets>
<runTarget>com.developpez.myapplication.Application/Application.html</runTarget>
<LogLevel>INFO</LogLevel>
</configuration>
</plugin>
```

Configuration du plugin GWT

```
<style>DETAILED</style>
<noServer>false</noServer>
<extraJvmArgs>-Xmx512m</extraJvmArgs>
<gwtVersion>${gwtVersion}</gwtVersion>
</configuration>
<executions>
  <execution>
    <goals>
      <!-- <goal>mergewebxml</goal>-->
      <!-- <goal>i18n</goal>-->
      <goal>compile</goal>
      <goal>test</goal>
    </goals>
  </execution>
</executions>
</plugin>
```

On note que la liste des goals est définie dans le plugin. Ce sont les goals lancés sur les phases post gwt:debug et gwt:gwt. Le goal de test est donc lancé lors de la phase d'installation, que vous ayez spécifié -Dmaven.test.skip ou non.

Autre chose à noter, il s'agit de la configuration du plugin dependency qui permet de décompresser une archive, ici les bibliothèques GWT, dans votre repo local.

Configuration du plugin maven-dependency

```
<!-- Use the dependency plugin to unpack gwt-dev-PLATFORM-libs.zip -->
<!--
(this is a replacement for the old "automatic" mode - useful if you
don't have GWT installed already, or you just want a maven way to
handle gwt deps)
-->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <executions>
    <execution>
      <id>unpack</id>
      <phase>compile</phase>
      <goals>
        <goal>unpack</goal>
      </goals>
      <configuration>
        <artifactItems>
          <artifactItem>
            <groupId>com.google.gwt</groupId>
            <artifactId>gwt-dev</artifactId>
            <version>${gwtVersion}</version>
            <classifier>${platform}-libs</classifier>
            <type>zip</type>
            <overwrite>false</overwrite>
            <outputDirectory>${settings.localRepository}/com/google/gwt/gwt-dev/
            ${gwtVersion}</outputDirectory>
          </artifactItem>
        </artifactItems>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Afin de coller au mieux à l'arborescence standard maven, j'ai déplacé le projet public et le module Application.gwt.xml dans src/main/resources. De même j'utilise bien un répertoire src/main/webapp pour stocker les fichiers de ma webapp. On a donc désormais une arborescence maven2 claire et une méthode simple pour lancer GWT en hosted mode. On peut passer à la suite.

III - Séparation des projets


Après génération nous avons l'ensemble des sources de notre projet dans 2 packages : `com.developpez.myapplication.client` et `com.developpez.myapplication.server`. Partons du bon pied dès le départ, je vous encourage à séparer vos projets par couche :

- un projet de modèle : `com.developpez.myapplication.model =>.myapplication-model`
- un projet de service : `com.developpez.myapplication.services =>.myapplication-services`
- un war contenant la partie cliente : `com.developpez.myapplication.client =>.myapplication-client`
- un war final d'intégration des composants : `myapplication`

Il n'y a pas de grande difficulté à créer des projets vide mais je vais cependant détailler quelques étapes qui peuvent s'avérer complexe.

III-A - Inclusion de source externe dans le module GWT


En séparant les sources du modèle, des services et de la partie cliente, la compilation du projet `myapplication-client` ne fonctionne plus.

 *l'exemple de compilation ci-dessous fait état de source que vous n'avez pas encore créé, il n'est là que pour illustrer. Nous verrons par la suite les sources des fichiers abordés ici.*

```
Compiling module com.developpez.myapplication.Application
  Refreshing module from source
    Validating newly compiled units
      Removing units with errors
        [ERROR] Errors in
'file:/E:/Developpement/checkout/developpez/myapplication-client/src/main/java/com/developpez/
myapplication/client/model/datasources/ProjectDS.java'
          [ERROR] Line 65: No source code is available for type
com.developpez.services.ProjectServiceAsync;
            did you forget to inherit a required module?
          [ERROR] Line 65: No source code is available for type
com.developpez.services.ProjectService;
            did you forget to inherit a required module?
.....
```

En effet nous n'avons pas inclus ces packages dans la liste des sources traduisibles en javascript, ce que nous allons faire immédiatement dans le fichier de module : `Application.gwt.xml`

```
<source path="model"/>
<source path="client"/>
<source path="services">
  <exclude name="**/impl/**" />
</source>
```

 *Nous noterons que nous avons exclu le répertoire d'implémentation des services. En effet, si nous souhaitons connaître la définition (les interfaces) des services ainsi que les exceptions côté clients, nous ne devons pas avoir connaissance des implémentations des services. Celles-ci ne seront de toute façon pas translatables dans la majorité des cas.*

Cependant pour que cela soit possible, il nous faut les sources de ces artefacts dans le classpath pour que GWT les translate en javascript, ce qui n'est pas le cas puisque n'avons pour dépendances que les binaires seuls sous forme de jars. Nous allons donc modifier les poms de nos projets `model` et `services` afin de générer les jars de source.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-source-plugin</artifactId>
      <executions>
        <execution>
          <id>attach-sources</id>
          <phase>verify</phase>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
    
```


Ceux-ci sont ensuite ajoutés en dépendance du projet client en utilisant le type java-source :


```

<dependency>
  <groupId>${groupId}</groupId>
  <artifactId>myapplication-model</artifactId>
  <version>${version}</version>
  <type>java-source</type>
</dependency>
<dependency>
  <groupId>${groupId}</groupId>
  <artifactId>myapplication-persistence</artifactId>
  <version>${version}</version>
  <type>java-source</type>
</dependency>
    
```

III-B - Le mode noServer

Tout d'abord quelques définitions utiles :

 *GWT peut être utilisé en "hosted mode" lors de la phase de développement. GWT utilise alors une librairie java pour le rendu de l'IHM. Il ne s'agit pas de Web mais bien d'un mode Java ("à la Swing") ! Ceci permet notamment d'utiliser votre débogueur favori pour analyser le comportement de votre application.*

 *En mode **hosted** on peut préciser si on souhaite utiliser la partie serveur courante ou un autre serveur déjà démarré. Cette fonctionnalité est très pratique car nous verrons par la suite que nos servlets ne pourront pas cohabiter avec le mode **hosted** ce qui rendra obligatoire l'utilisation du mode **noServer**.*

Pour pouvoir utiliser la partie cliente en mode **hosted** sans la partie serveur, nous allons rendre paramétrable l'activation du mode **noServer**. Le mode **noServer** permet de ne lancer que la partie cliente en hosted mode, la partie serveur utilisée étant accessible via un port défini dans la configuration du plugin gwt-maven. Voici donc le pom modifié :

```

<configuration>
  <compileTargets>
    
```

```

<value>com.developpez.myapplication.Application</value>
</compileTargets>
<runTarget>com.developpez.myapplication.Application/Application.html</runTarget>
<logLevel>INFO</logLevel>
<style>DETAILED</style>
<noServer>${gwt.noserver}</noServer>
<port>8082</port>
<extraJvmArgs>-Xmx512m</extraJvmArgs>
<gwtVersion>${gwtVersion}</gwtVersion>
</configuration>
    
```


La variable `gwt.noserver` étant définie dans les propriétés plus haut dans le fichier :

```

<properties>
....
<gwt.noserver>true</gwt.noserver>
....
</properties>
    
```

Il devient alors possible d'appeler au besoin le goal `gwt-maven:gwt` en mode `noserver` ou non. Par exemple : `gwt-maven:gwt -Dgwt.noserver=false`

III-C - Le lancement de la partie serveur dans cargo

L'utilisation du plugin maven  **Cargo** vous permet de déployer votre web application dans un container Tomcat lors du build. Nous allons pour cela paramétrer le `pom.xml` et ajouter la configuration du plugin dans la partie `build` :

```

<build>
<plugins>
<plugin>
<groupId>org.codehaus.cargo</groupId>
<artifactId>cargo-maven2-plugin</artifactId>
<version>0.3.1</version>
<configuration>
<wait>${cargo.wait}</wait>
<container>
<containerId>${tomcat.containerId}</containerId>
<log>${project.build.directory}/${tomcat.containerId}/cargo.log</log>
<zipUrlInstaller>
<url>${tomcat.downloadUrl}</url>
<installDir>${tomcat.installDir}</installDir>
</zipUrlInstaller>
<timeout>360000</timeout>
</container>
<configuration>
<home>${project.build.directory}/${tomcat.containerId}/container</home>
<properties>
<cargo.servlet.port>${tomcat.port}</cargo.servlet.port>
<cargo.logging>high</cargo.logging>
</properties>
</configuration>
</configuration>
</plugin>
</plugins>
</build>
    
```

Dans la section **profiles**, nous décrivons les deux profiles que nous allons utiliser pour lancer cargo :

```

<!-- profiles (with activation per platform) -->
    
```

```

<profiles>
<profile>
<id>debug</id>
<!--
    profil permettant de debugger tomcat
    les logs sont dans myapplication\target\tomcat5x\cargo.log
-->
<activation>
<property>
<name>debug</name>
<value>>true</value>
</property>
</activation>

<build>
<plugins>
<plugin>
<groupId>org.codehaus.cargo</groupId>
<artifactId>cargo-maven2-plugin</artifactId>
<configuration>
<wait>>true</wait>
<container>
<containerId>${tomcat.containerId}</containerId>
</container>
<configuration>
<properties>
<cargo.jvmargs>
-Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdp:transport=dt_socket,server=y,suspend=y,address=7456
</cargo.jvmargs>
</properties>
</configuration>
</configuration>
</plugin>
</plugins>
</build>
</profile>
<profile>
<!-- Launch cargo on pre-integration-test phase -->
<id>cargo</id>
<build>
<plugins>
<plugin>
<groupId>org.codehaus.cargo</groupId>
<artifactId>cargo-maven2-plugin</artifactId>
<executions>
<execution>
<id>start-container</id>
<phase>pre-integration-test</phase>
<goals>
<goal>start</goal>
</goals>
</execution>
<execution>
<id>stop-container</id>
<phase>post-integration-test</phase>
<goals>
<goal>stop</goal>
</goals>
</execution>
</executions>
</plugin>
</plugins>
</build>
</profile>
</profiles>
    
```

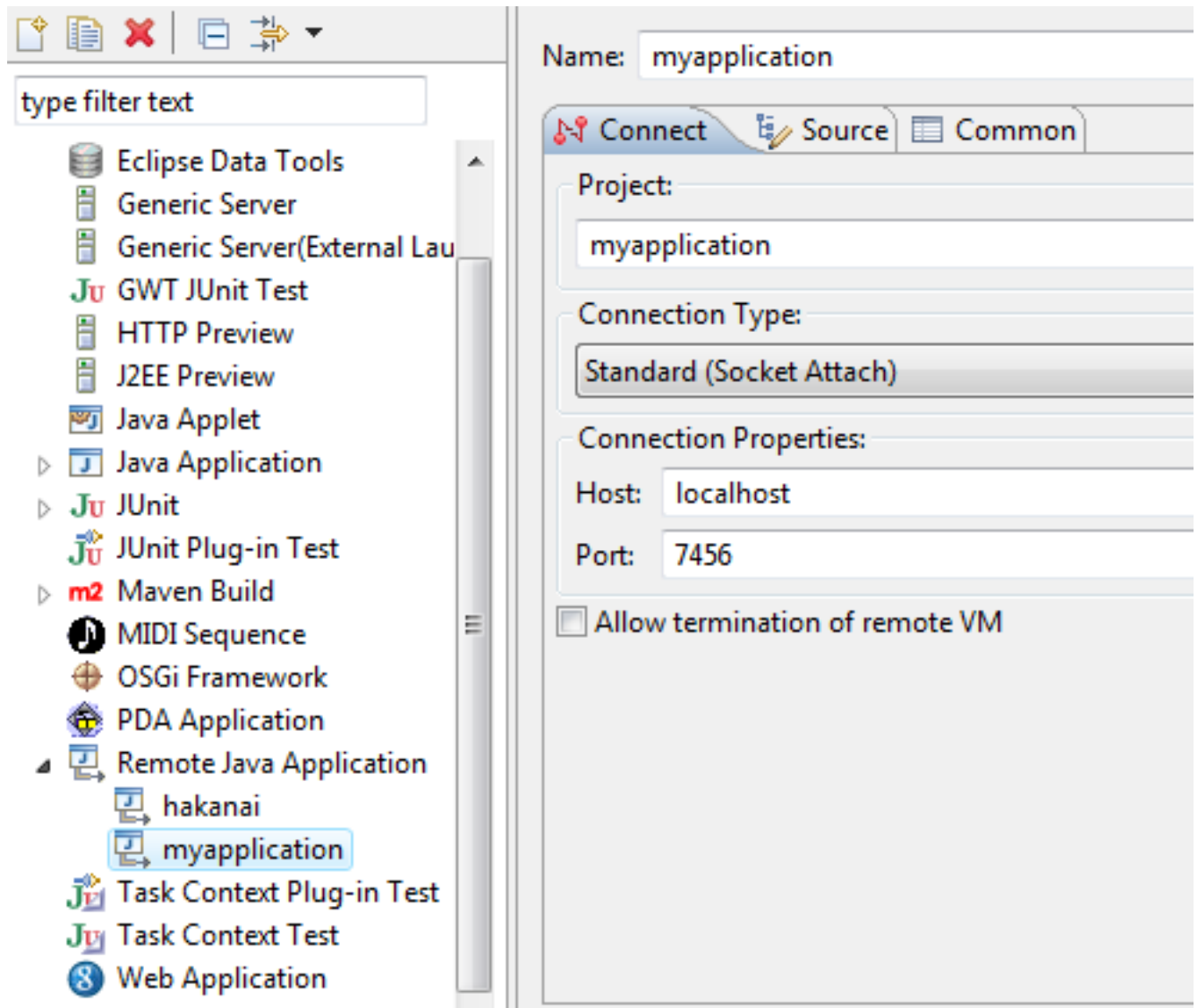
Avec ce paramétrage, vous êtes désormais capable de lancer cargo lors de la phase d'install avec la commande suivante :

```
mvn install -Dmaven.test.skip -Dcargo.wait -Pcargo
```


La variable cargo.wait permettra de suspendre Cargo après le start pour que vous puissiez utiliser votre application et qu'elle ne s'arrête pas immédiatement. Un autre profile a été défini pour pouvoir lancer le serveur en mode debug avec la commande

```
mvn install -Dmaven.test.skip -Ddebug -Pcargo
```

Pour ce faire, une fois Cargo démarré et bloqué sur la phase de démarrage du container, lancez un debug distant via eclipse :



Lancement en mode debug distant


 Vous noterez que vous êtes désormais capable de lancer le mode **hosted** de GWT en mode debug avec l'option **noServer** configurée pour attaquer votre serveur web déployé par cargo, lui-même en mode debug. Vous avez donc la possibilité de développer/débugger sans recompiler (sauf en cas d'ajouts de méthode ou de classe).

IV - Partie cliente

De base GWT est déjà assez intéressant mais il manque tout de même un grand nombre de widgets évolués que nous ne souhaitons pas créer nous même. Il existe plusieurs librairies additionnelles pour réaliser des interfaces plus poussées. Je vous propose donc un petit comparatif.

IV-A - Wrapper or not wrapper

Tout d'abord abordons un petit point de vocabulaire et revoyons ensemble le concept de librairies wrappées ou pures GWT.

Lorsqu'on parle de wrapper GWT, il s'agit d'enrobage en  **Jsni** d'une librairie Javascript déjà existante. Cette technique présente l'avantage de nécessiter un effort "minimal" (mais néanmoins conséquent) pour porter une librairie javascript existante en GWT. L'autre avantage pour une librairie wrappée est de pouvoir maintenir une seule version de librairie Javascript, une librairie pure GWT nécessiterait de toute réécrire en GWT.

En contrepartie, un wrapper GWT comporte plusieurs inconvénients :


- fuites mémoires potentielles (les fuites dues au Javascript non généré par GWT)
- la compatibilité interbrowser repose sur le Javascript et plus sur GWT lui-même.
- les performances du Javascript inséré ne sont pas optimisées par le compilateur GWT.
- difficulté de débogage puisque le mode debug en Hosted Mode va pointer sur des méthodes Jsni non débuggables.
- difficulté pour étendre les composants en GWT qui wrappent du Jsni.

Une librairie pure GWT à l'inverse est écrite uniquement en Java avec traduction en Javascript par le compilateur GWT. Elle a l'avantage d'être déboguable et extensible en Java plus aisément et surtout de profiter des optimisations du compilateur GWT ainsi que de la cross compatibilité du code généré.















Finalement, choisir un wrapper GWT c'est accorder une grande confiance à la librairie sous-jacente sur ses performances et sa richesse de composants.

Sachant cela, le choix reste assez difficile à faire car de très bonnes librairies Javascript ont désormais un wrapper et faire l'impasse sur celles-ci paraît difficile. Pour ma part, ayant utilisé des librairies comme Dojo, Ext-Js, Yahoo etc... je sais n'avoir jamais eu besoin de les étendre ou les déboguer ou bien je m'y suis adapté puisque je n'avais aucune compétence en Javascript (par déboguer j'entend "plonger au coeur du code de la librairie"). Sur un projet en freelance ou pour du court terme, je suis prêt à faire confiance à un wrapper. Sur un projet en entreprise je serais plus circonspect et je pencherais sans doute pour une librairie pure GWT sachant que celles de qualité ne sont pas légion (pour l'instant).



IV-B - Comparatif


Pour effectuer le comparatif ci-dessous je n'ai pas eu la possibilité de tester toutes les librairies qui sont exposées ici :  **librairies GWT**. J'ai cependant souhaité apporter un comparatif sur des critères objectifs. Pour cela j'ai donc pris des critères simples :

- license
- importance de la communauté
- type (wrapper ou non)
- gestionnaire de bugs ouverts sur internet
- mode de direction du projet (despotisme, conseil restreint, aucun etc...)
- activité
- qualité du showcase

Nom et site	Showcase	Type	License	Gestionnaire de bugs	Mode de direction	Activité	Qualité du showcase
 GXT	 Showcase	Pur GWT	Open source (GPLv3) et Commercial	oui sous forme de forum pour la license open source	Directoire (même si la gestion du projet par Jack Slocum a fait des vagues par le passé)	Importante	5/5
 GWT Mosaic	 Showcase	Pur GWT	Open source (Apache 2.0)	oui	Directoire	Importante	4/5
 SmartClient	 Showcase	Wrapper	Open source (LGPL)	oui	Développeur unique mais associé à la société SmartClient pour la partie Javascript	Importante	5/5
 IT Mill Toolkit	 Showcase	Pur GWT	Open source (Apache 2.0)	oui	Directoire	Bonne	5/5
 GwtPHP	 Showcase ?	?	Dual license (open source et commercial)	?	Directoire	Aucune, projet en gestation	5/5 (il déchire vraiment mais il faut prendre le thème aéro)
 Tatami	 Showcase	wrapper dojo	Open source (LGPL, BSD, Apache 2.0)	oui	Directoire	Moyenne	3/5
 Rialto	 Showcase	wrapper rialto	Open source (Apache 2.0)	oui (mailing list)	Petite équipe	Moyenne	2/5

L'étude de ces différentes librairies m'a permis de mettre en évidence que la technologie GWT était encore assez jeune. Les librairies ne sont pas encore toutes matures même si certaines sont prometteuses. Les wrappers créés autour de très bonnes librairies ne sont malheureusement que des wrappers (Cf. le chapitre précédent) et j'aurais tendance à m'en éloigner pour un développement professionnel.

Parmi les librairies pures GWT, GXT semble intéressante mais bénéficie d'une mauvaise presse sur la toile quand à sa  **politique d'upgrade** mal documentée et mal conçue, sa  **qualité de codage** ou encore la récente

 **polémique** sur le changement de license un peu sauvage. ITMill (Vaadin) ou Mosaic dans une moindre mesure sont prometteurs mais j'ai trouvé peu d'échos d'eux sur le net.

En bref, si je devais faire un choix pour un projet professionnel, je m'orienterais sur ITMill ou GXT mais j'essaierais de me laisser une porte ouverte pour changer par la suite en créant des adaptateurs par exemple.

Gardez cependant à l'esprit un élément important lorsque vous ferez un choix : au final même si votre GUI côté client est du plus bel effet visuellement (bling bling, avouons-le ^^), l'essentiel c'est tout de même ce qui se passe côté serveur. Et si le travail effectué sur l'IHM est important, il doit bien s'intégrer avec le reste. Vos contrôles de formulaire seront côté serveur, votre modèle doit être cohérent et non dupliqué, la sécurité doit être une de vos préoccupations les plus importantes etc...

Et toute cette petite mécanique très coûteuse est encore bien souvent zappée par les librairies GWT. On trouve cependant des notions intéressantes chez smartGWT qui propose un système de datasource évolué lié aux composants et capable de faire du RPC, du REST ou du Web service. Quelques projets sur google incubator semblent réfléchir à une implémentation standard pour la sécurité (ce que nous verrons plus bas dans cet article). GXT propose lui des aides sur le databinding. Gageons qu'avec le temps les librairies s'enrichiront sur ces points qui restent encore assez légers.


Pour cet article, j'avoue m'être laissé tenter par SmartGWT justement pour sa notion de Datasource. Effectivement c'est un wrapper et j'ai donné plusieurs critiques contre les wrappers mais cet article a surtout pour but de démontrer l'intégration d'un projet complet maven, spring, GWT etc... Le changement de librairie ne devrait pas modifier la philosophie de la démonstration.

IV-C - SmartGWT

Parmi les concepts très importants de SmartGWT, l'un des principaux concerne les Datasources. Les datasources sont non seulement des conteneurs de données mais ils contiennent en sus du metadata : de la description de données.

Les datasources SmartGWT s'interfacent avec l'ensemble des composants pour permettre leur affichage sans se soucier des données sous jacentes. La même datasource peut donc être utilisée pour un tree, un tableau, une liste de choix etc... L'instance étant commune, un changement d'une donnée sera reflété sur les autres contrôles.

Mieux encore, si votre datasource est connecté au serveur, les changements seront effectués de façon transparente pour vous automatiquement. Autrement dit c'est SmartGWT qui s'occupera d'effectuer les appels vers le serveur, de mettre à jour les données etc...

A ce titre, l'un des meilleurs exemples pour illustrer les datasources est présent sur le  **showcase**. Dans l'exemple, une seule datasource est utilisée pour un TreeGrid, une ListGrid et un PickTreeItem. La notion de Datasource est aussi utilisé pour faire du databinding.

Il existe par défaut deux modes de datasource dans SmartGWT, les datasources REST et les datasources WebServices. Ces deux méthodes sont très intéressantes, notamment si vous possédez déjà une architecture à base de webservices ou si vous ne faites pas de JAVA côté serveur. Ces deux modes de communication sont des alternatives très convaincantes au RPC traditionnellement utilisé avec GWT.

Si on souhaite faire du RPC, on trouve sur le forum GWT une datasource utilisant RPC pour la communication avec le serveur, la GwtRpcDataSource. C'est celle ci que nous allons ajouter à notre projet :

GwtRpcDataSource.java

```
package com.developpez.client.model.datasources;
```

GwtRpcDataSource.java

```

import com.smartgwt.client.data.DSRequest;
import com.smartgwt.client.data.DSResponse;
import com.smartgwt.client.data.DataSource;
import com.smartgwt.client.types.DSDataFormat;
import com.smartgwt.client.types.DSProtocol;

/**
 * Data source with ability to communicate with server by GWT RPC.
 * <p/>
 * SmartClient natively supports data protocol "clientCustom". This protocol
 * means that communication with server should be implemented in
 * <code>transformRequest (DSRequest request)</code> method. Here is a few
 * things to note on <code>transformRequest</code> implementation:
 * <ul>
 * <li><code>DSResponse</code> object has to be created and
 * <code>processResponse (requestId, response)</code> must be called to finish
 * data request. <code>requestId</code> should be taken from original
 * <code>DSRequest.getRequestId ()</code>.</li>
 * <li>"clientContext" attribute from <code>DSRequest</code> should be copied to
 * <code>DSResponse</code>.</li>
 * <li>In case of failure <code>DSResponse</code> should contain at least
 * "status" attribute with error code (<0>).</li>
 * <li>In case of success <code>DSResponse</code> should contain at least "data"
 * attribute with operation type specific data:
 * <ul>
 * <li>FETCH - <code>ListGridRecord[]</code> retrieved records.</li>
 * <li>ADD - <code>ListGridRecord[]</code> with single added record. Operation
 * is called on every newly added record.</li>
 * <li>UPDATE - <code>ListGridRecord[]</code> with single updated record.
 * Operation is called on every updated record.</li>
 * <li>REMOVE - <code>ListGridRecord[]</code> with single removed record.
 * Operation is called on every removed record.</li>
 * </ul>
 * </li>
 * </ul>
 *
 * @author Aleksandras Novikovas
 * @author System Tier
 * @version 1.0
 */
public abstract class GwtRpcDataSource extends DataSource
{
    /**
     * Creates new data source which communicates with server by GWT RPC. It is
     * normal server side SmartClient data source with data protocol set to
     * <code>DSProtocol.CLIENTCUSTOM</code> ("clientCustom" - natively supported
     * by SmartClient but should be added to smartGWT) and with data format
     * <code>DSDataFormat.CUSTOM</code>.
     */
    public GwtRpcDataSource()
    {
        setDataProtocol(DSProtocol.CLIENTCUSTOM);
        setDataFormat(DSDataFormat.CUSTOM);
        setClientOnly(false);
    }

    /**
     * Executes request to server.
     *
     * @param request
     *         <code>DSRequest</code> being processed.
     * @return <code>Object</code> data from original request.
     */
    @Override
    protected Object transformRequest(DSRequest request)
    {
        String requestId = request.getRequestId();
        DSResponse response = new DSResponse();
        response.setAttribute("clientContext", request.getAttributeAsObject("clientContext"));
        // Assume success
    }
}

```

GwtRpcDatasource.java

```

response.setStatus(0);
switch (request.getOperationType())
{
    case FETCH:
        executeFetch(requestId, request, response);
        break;
    case ADD:
        executeAdd(requestId, request, response);
        break;
    case UPDATE:
        executeUpdate(requestId, request, response);
        break;
    case REMOVE:
        executeRemove(requestId, request, response);
        break;
    default:
        // Operation not implemented.
        break;
}
return request.getData();
}

/**
 * Executed on <code>FETCH</code> operation.
 * <code>processResponse (requestId, response)</code> should be called when
 * operation completes (either successful or failure).
 *
 * @param requestId
 *         <code>String</code> extracted from
 *         <code>DSRequest.getRequestId ()</code>.
 * @param request
 *         <code>DSRequest</code> being processed.
 * @param response
 *         <code>DSResponse</code>. <code>setData (list)</code> should be
 *         called on successful execution of this method.
 *         <code>setStatus (<0></code> should be called on failure.
 */
protected abstract void executeFetch(String requestId, DSRequest request, DSResponse response);

/**
 * Executed on <code>ADD</code> operation.
 * <code>processResponse (requestId, response)</code> should be called when
 * operation completes (either successful or failure).
 *
 * @param requestId
 *         <code>String</code> extracted from
 *         <code>DSRequest.getRequestId ()</code>.
 * @param request
 *         <code>DSRequest</code> being processed.
 *         <code>request.getData ()</code> contains record should be
 *         added.
 * @param response
 *         <code>DSResponse</code>. <code>setData (list)</code> should be
 *         called on successful execution of this method. Array should
 *         contain single element representing added row.
 *         <code>setStatus (<0></code> should be called on failure.
 */
protected abstract void executeAdd(String requestId, DSRequest request, DSResponse response);

/**
 * Executed on <code>UPDATE</code> operation.
 * <code>processResponse (requestId, response)</code> should be called when
 * operation completes (either successful or failure).
 *
 * @param requestId
 *         <code>String</code> extracted from
 *         <code>DSRequest.getRequestId ()</code>.
 * @param request
 *         <code>DSRequest</code> being processed.
 *         <code>request.getData ()</code> contains record should be

```

GwtRpcDatasource.java

```

*          updated.
* @param response
*          <code>DSResponse</code>. <code>setData (list)</code> should be
*          called on successful execution of this method. Array should
*          contain single element representing updated row.
*          <code>setStatus (<0></code> should be called on failure.
*/
protected abstract void executeUpdate(String requestId, DSRequest request, DSResponse response);

/**
* Executed on <code>REMOVE</code> operation.
* <code>processResponse (requestId, response)</code> should be called when
* operation completes (either successful or failure).
*
* @param requestId
*          <code>String</code> extracted from
*          <code>DSRequest.getRequestId ()</code>.
* @param request
*          <code>DSRequest</code> being processed.
*          <code>request.getData ()</code> contains record should be
*          removed.
* @param response
*          <code>DSResponse</code>. <code>setData (list)</code> should be
*          called on successful execution of this method. Array should
*          contain single element representing removed row.
*          <code>setStatus (<0></code> should be called on failure.
*/
protected abstract void executeRemove(String requestId, DSRequest request, DSResponse response);
}
    
```

Nos datasources RPC seront donc toutes des classes filles de cette classe. Elles implémenteront les méthodes fetch, update, remove, add qui correspondent aux méthodes principales des Datasources SmartGWT.

V - Spring

L'objet de ce chapitre n'est pas de présenter Spring, nous allons partir du principe que vous connaissez ce framework. Notre objectif sera de l'utiliser côté serveur pour gérer notre application.


V-A - Installation des dépendances

Nous intégrons les dépendances maven dans le pom.

```

<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring</artifactId>
<version>2.5.6</version>
</dependency>
    
```


V-B - Chargement de l'application context


Pour charger l'application context nous utiliserons le mode standard de Spring dans une application web : le  **ContextLoaderListener**. Celui-ci est un listener J2EE positionné dans le fichier web.xml.

```
<context-param>
```

```







<param-name>contextConfigLocation</param-name>
<param-value>
  classpath:applicationContext_GWT.xml
</param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
    
```


 Le fichier `applicationContext_GWT.xml` dont il est question dans le paramètre `contextConfigLocation` sera défini plus loin. Nous en rajouterons encore d'autres par la suite.

Désormais votre  **WebApplicationContext** est chargé et accessible côté serveur. Nous allons l'utiliser dans les chapitres suivants.

V-C - Stratégies d'intégration

Autant dire tout de suite, l'intégration de spring n'est pas chose aisée à mon sens si on veut le faire bien. Il existe une multitude d'approches et d'articles sur le net ayant chacun des avantages et inconvénients. Voici une liste des articles les plus représentatifs que j'ai pu lire :

Nom	Description
 GWT-SL	GWT-SL utilise principalement une DispatcherServlet de spring pour mapper les appels rpc vers des services définis en spring. La librairie propose aussi des helpers pour l'intégration avec hibernate.
 How to Integrate GWT with Spring 2.0 (gwt incubator)	De même que le précédent, cet article recourt à une DispatcherServlet et un handler spécialisé - le GWController - pour rediriger les requêtes vers les bons services.
 Luca Masini	Très proche de la précédente, l'approche de Luca Massini remplace le GWController par un ServletForwardingController ou un ServletWrappingController.
 Chris Lee	L'un des articles que j'ai le plus eu de mal à appréhender puisqu'il utilise l'AOP intensivement et ne présente pas d'exemple. Des annotations sont utilisées pour éviter de mêler la couche services aux librairies GWT.
 Spring4gwt	La méthode ressemble à celles impliquant une DispatcherServlet sauf qu'une surcouche semble avoir été rajoutée : la SpringGwtRemoteServiceServlet. La jeunesse de la librairie (à peine 1 mois lors de la rédaction de l'article), le manque d'exemple et la profusion d'AOP qui m'a un peu perdu n'a pas retenu mon attention.
 Seewah	Cette fois, plus de DispatcherServlet. Seewah s'oriente surtout sur la méthode pour initialiser les servlets avec l'applicationContext de spring en utilisant des servlet "auto injective". C'est surtout son


	article sur l'intégration spring security qui m'intéressera par la suite.
 Pgt	On retrouve la même logique que le précédent, l'objectif étant d'avoir des servlets utilisant le contexte spring pour initialiser ses services.

C'est au final GWT-SL que je vais tenter d'utiliser puisque la librairie m'a paru plus aboutie que les autres et propose des addons intéressants, notamment son intégration hibernate. Au moment de la rédaction de l'article je n'ai cependant pas senti qu'une approche se démarquait singulièrement des autres.

Une des limitations principales de toutes les solutions à base de DispatcherServlet est qu'elles ne permettent plus l'utilisation du mode hosted pour la partie serveur (Cf.  [explication de GWT-SL](#)). En effet, observons un instant le web.xml retouché par gwt-maven au moment de la compilation :

```

<!--inserted by gwt-maven-->
<servlet>
<servlet-name>shell</servlet-name>
<servlet-class>com.google.gwt.dev.shell.GWTShellServlet</servlet-class>
</servlet>
<!--inserted by gwt-maven-->
<servlet-mapping>
<servlet-name>shell</servlet-name>
<url-pattern>/*</url-pattern>
</servlet-mapping>
    
```

Pour fonctionner en mode hosted, une servlet GWT est ajoutée au web.xml afin de capter tout les appels. Il y a alors conflit entre les deux servlets. Une  [solution](#) pas très propre permet de rediriger de la DispatcherServlet vers la servlet GWT mais je ne la conseille pas. Je l'indique cependant pour que vous la connaissiez.

```

<bean id="gwtShellController"
class="org.springframework.web.servlet.mvc.ServletForwardingController">
  <property name="servletName">
    <value>shell</value>
  </property>
</bean>

<bean id="urlMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/*<user.rpc">userController</prop>
      <prop key="/*>gwtShellController</prop>
    </props>
  </property>
</bean>
    
```

Cette limitation dans le cas où vous n'auriez qu'un projet pour la partie client et serveur ne nous dérange pas. En effet nous avons créé un artefacts pour chaque couche de notre application. La partie cliente sera toujours jouée par le plugin gwt-maven mais en mode noServer, tandis que notre partie serveur sera lancée par Tomcat avec le plugin Cargo.

VI - GWT-SL

VI-A - Présentation

GWT-SL est une librairie simplifiant l'intégration de spring côté serveur. Son principal objectif étant de proposer des outils pour exporter vos services, transformer vos exceptions ou faire transiter vos beans hibernate vers la couche cliente. Nous utiliserons par la suite Spring pour gérer la sécurité.


VI-B - Modification du pom


Nous intégrons les dépendances maven dans le pom du projet myapplication (le war final) :

```

<!-- GWT SL for spring integration on server side -->
<dependency>
  <groupId>net.sf.gwt-widget</groupId>
  <artifactId>gwt-sl</artifactId>
  <version>0.1.5b</version>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring</artifactId>
  <version>2.5.6</version>
</dependency>
    
```

 *Nous utiliserons GWT-SL en version 0.1.5b mais en redéfinissant la version de spring-core. A noter que gwt-servlet est en version 1.7. GWT-SL ramène spring et gwt-servlet dans ses dépendances transitives mais n'oubliez pas que Maven, pour un même artefact, prend toujours la version déclaré au niveau le plus haut, ici notre pom.*

 *Je n'ai pas trouvé la version 0.1.5b sur les repos officiels maven, j'ai donc installé cette librairie à partir du site officiel dans mon repository local. N'hésitez pas à faire de même si la librairie n'est toujours pas sur les repository officiels au moment où vous lirez cet article.*

VI-C - Ajout de la servlet de dispatch

La servlet de Dispatch est propre à Spring et non à GWT-SL. Elle est utilisée dans un grand nombre de solutions visant à intégrer Spring et GWT. Cette servlet s'initialise dans le fichier web.xml :

web.xml

```

<servlet>
  <servlet-name>handler</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>handler</servlet-name>
  <url-pattern>*.rpc</url-pattern>
</servlet-mapping>
    
```

Nous verrons par la suite que le nom de la servlet est important puisqu'il détermine aussi le nom du fichier de contexte spring qui sera cherché par la DispatcherServlet.

Le pattern de filtrage concerne uniquement les appels RPC que nous suffixerons par convention par **.rpc**. Nous ne souhaitons en effet pas avoir de conflit avec les fichiers statiques de l'application.

VI-D - Création d'un service RPC

L'un des services que nous allons écrire va nous permettre de gérer des projets (fallait bien un exemple non ?). Nous le créons dans le projet myapplication-service, créé à cet effet. L'interface reflète les opérations principales d'un service CRUD : Create, Retrieve, Update, Delete nécessaires pour SmartGWT (voir plus haut la partie sur les DataSources SmartGWT) :

```
package com.developpez.myapplication.services;

import java.util.List;

import org.springframework.security.annotation.Secured;

import com.developpez.myapplication.model.Project;
import com.developpez.myapplication.services.exception.ServiceSecurityException;
import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

/**
 * Services related to projects operations
 *
 * @version 1.0
 */
@RemoteServiceRelativePath("ProjectService.rpc")
public interface ProjectService extends RemoteService
{

    /**
     * @return the list of projects available
     * @throws ServiceSecurityException
     */
    @Secured("ROLE_ADMIN")
    List<Project> fetch() throws ServiceSecurityException;

    /**
     * @param record the project to add
     * @return the record added and potentially modified on server side
     * @throws ServiceSecurityException
     */
    @Secured("ROLE_ADMIN")
    Project add(Project record) throws ServiceSecurityException;

    /**
     * @param record the project to update
     * @return the record updated and potentially modified on server side
     * @throws ServiceSecurityException
     */
    @Secured("ROLE_ADMIN")
    Project update(Project record) throws ServiceSecurityException;

    /**
     * @param record the record to remove
     * @throws ServiceSecurityException
     */
    @Secured("ROLE_ADMIN")
    void remove(Project record) throws ServiceSecurityException;
}
```

On remarque tout de suite quelque chose d'assez désagréable, nous utilisons une annotation GWT dans notre service, ce qui le rend dépendant de GWT même si nous souhaitons l'utiliser dans un autre contexte. Notez cependant que nous aurions pu nous en passer en déclarant les interfaces de nos services au moment de l'export via l'exporteur GWT-SL.

L'implémentation de ce service reste pour l'instant basique pour nos besoins, nous l'enrichirons par la suite :

```
package com.developpez.services.impl;

import java.util.ArrayList;
import java.util.List;

import com.developpez.myapplication.model.Project;
import com.developpez.myapplication.services.ProjectService;

/**
 * Mock implementation
 */
public class ProjectServiceImpl implements ProjectService
{
    /**
     * serial id
     */
    private static final long serialVersionUID = 9177398707013302517L;

    /**
     * list of project (in memory)
     */
    private List<Project> liste;

    /**
     * empty constructor
     */
    public ProjectServiceImpl() {

        liste = new ArrayList<Project>();
        liste.add(new Project(0, "nom1", "prenom1"));
        liste.add(new Project(1, "nom3", "prenom3"));
    }
    /** (non-Javadoc)
     * @see com.developpez.client.services.ProjectService#add(com.developpez.model.Project)
     */
    public Project add(Project record)
    {
        liste.add(record);
        return record;
    }

    /** (non-Javadoc)
     * @see com.developpez.client.services.ProjectService#fetch()
     */
    public List<Project> fetch()
    {
        return liste;
    }

    /** (non-Javadoc)
     * @see com.developpez.client.services.ProjectService#remove(com.developpez.model.Project)
     */
    public void remove(Project record)
    {
        liste.remove(record);
    }

    /** (non-Javadoc)
     * @see com.developpez.client.services.ProjectService#update(com.developpez.model.Project)
     */
}
```

```

    */
    public Project update(Project record)
    {
        liste.remove(record);
        liste.add(record);
        return record;
    }
}
    
```

VI-E - Exportation du service

Notre service étant créé, nous allons maintenant le déclarer en spring et l'associer à une url grace au GWTHandler de GWT-SL :

applicationContext_GWT.xml


```

<bean id="projectService"
    class="com.developpez.myapplication.services.impl.ProjectServiceImpl">
</bean>
    
```

handler-servlet.xml

```

<bean id="urlProjectMapping" class="org.gwtwidgets.server.spring.GWTHandler">
<!-- Supply here mappings between URLs and services. Services must implement the RemoteService interface but
are not otherwise restricted.-->
<property name="mappings">
    <map>
        <!-- Other mappings could follow -->
        <entry key="/com.developpez.myapplication.Application/ProjectService.rpc" value-
ref="projectService" />
    </map>
</property>
</bean>
    
```

 *D'autres méthodes d'exportation du service sont utilisables, se référer à la [documentation officielle](#).*

Détail important à noter, les services et autres beans qui ne sont pas dépendants de la DispatcherServlet sont créés dans le fichier applicationContext_GWT.xml. Mais la DispatcherServlet effectue elle-même la recherche du fichier spring qui définit les mappings dans le répertoire WEB-INF. Ce fichier doit être nommé du nom de la servlet + "-servlet.xml".

Exemple pour nous : handler-servlet.xml

Il est cependant possible de passer outre cette convention de nommage et de redéfinir le nom du fichier via le paramétrage suivant :

```

<init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/handler-servlet.xml
    </param-value>
</init-param>
    
```

Voilà, notre service est désormais appelable en RPC. Pour cela il reste à déclarer l'interface asynchrone.

Le callback du service asynchrone doit suivre une convention de nommage bien précise :

- être dans le même package que le service qu'il décorrelle
- avoir le même nom collé avec "Async"

Ca peut vous sembler familier si vous utilisez déjà ce type de convention des nommages pour les interfaces des MBeans en JMX.

ProjectServiceAsync.java

```
package com.developpez.myapplication.services;

import java.util.List;

import com.developpez.myapplication.model.Project;
import com.google.gwt.user.client.rpc.AsyncCallback;

public interface ProjectServiceAsync {

    public abstract void fetch (AsyncCallback<List<Project>> asyncCallback);

    public abstract void add (Project record, AsyncCallback<Project> asyncCallback);

    public abstract void update (Project record, AsyncCallback<Project> asyncCallback);

    public abstract void remove (Project record, AsyncCallback<Object> asyncCallback);

}
```

VII - Spring-security

VII-A - Introduction

L'objectif de ce chapitre sera désormais de gérer la sécurité par Spring security. La problématique consistant essentiellement à introduire notre contexte de sécurité dans les requêtes GWT et les réconcilier côté serveur.

Cependant il n'est pas possible de parler de spring-security sans introduire auparavant la sécurité des sites GWT. En recherchant sur le web, j'ai séparé les approches que j'ai pu trouver en deux :


- login inclus dans la page globale
- page de login externe aux autres modules

Côté client, la première méthode présuppose que vous ayez un affichage conditionnel de votre site selon une variable. Naïvement vous pourriez donc écrire ceci en GWT :

```
if (!isAuthenticated)
{
    mainWidget = createLoginWidget();
} else
{
    mainWidget = createMainFrame();
}
```

Vous pouvez aussi ne donner accès à certaines widgets sous les mêmes conditions ou cacher des données selon le rôle de l'utilisateur. Cependant il faut garder à l'esprit que tout votre package client va être traduit en javascript puis affiché sur une page html téléchargée sur le poste client. Le client a donc toute possibilité de modifier les variables javascript de son côté. Il est toujours nécessaire de sécuriser les appels aux données qui peuvent être déclenchés par vos widgets ou un utilisateur malveillant. Ne faites pas confiance à ce qui provient de l'utilisateur.

Cette approche est tout à fait utilisable mais gardez à l'esprit que tout ce qui est côté client peut être visible, même si vous avez mis une condition quelconque dans votre code GWT. Ne tentez pas de cacher des données sur votre page web avec un simple "if (!isAuthenticated)" !!!

La seconde approche consiste à séparer la page de login du reste de l'application. Cette méthode est celle encouragée par  [cet article de la FAQ de GWT](#). L'accès aux modules de l'application pourra donc être protégé par une première sécurité basée sur un filtre par exemple.




Les deux méthodes me paraissent valables, tout dépend ensuite si vous considérez qu'accéder aux modules sans les données associées est acceptable ou non.

Dans les deux cas cités ci dessus, les appels aux services RPC pourront être protégés de deux façons :

- un filtre vérifiant l'authentification de l'utilisateur
- un filtre vérifiant les autorisations de l'utilisateur

VII-B - Mise en oeuvre

Pour cette section je me suis basé sur différents articles du web :

-  [Implémentation par Seewah](#)
-  [GWT incubator : spring security](#)
-  [GWT incubator : filtrage d'url](#)

La mise en oeuvre va suivre le modèle suivant :

- le positionnement d'un contexte Spring Security dans un service d'authentification
- l'utilisation d'un DelegatingFilterProxy dont le rôle sera de synchroniser les requêtes HTTP avec le contexte de sécurité
- l'utilisation d'un contexte de sécurité pour une gestion des rôles (par AOP ou non)
- l'interdiction de l'accès à certaines url en fonction du contexte de sécurité

VII-C - Le service d'authentification

Tout d'abord créons son interface :

```
package com.developpez.myapplication.services;

import com.developpez.myapplication.model.ReturnMemento;
import com.google.gwt.user.client.rpc.RemoteService;
import com.google.gwt.user.client.rpc.RemoteServiceRelativePath;

/**
 * Authentication service.
 *
 */
@RemoteServiceRelativePath("AuthenticationService.rpc")
public interface AuthenticationService extends RemoteService {

    /**
     * Authenticates user.
     *
     * @param username
     * @param password
     * @return whether authentication is successful
     */
}
```

```
ReturnMemento authenticate(String username, String password);

/**
 * Terminates a user's security session.
 */
void logout();
}
```

L'implémentation s'appuie sur Spring Security. Nous allons positionner le security context dans le contexte spring si l'utilisateur est reconnu.

```
package com.developpez.myapplication.services.impl;

import org.springframework.security.Authentication;
import org.springframework.security.GrantedAuthority;
import org.springframework.security.GrantedAuthorityImpl;
import org.springframework.security.context.SecurityContext;
import org.springframework.security.context.SecurityContextHolder;
import org.springframework.security.context.SecurityContextImpl;
import org.springframework.security.providers.UsernamePasswordAuthenticationToken;
import org.springframework.security.userdetails.User;

import com.developpez.myapplication.model.ReturnMemento;
import com.developpez.myapplication.services.AuthenticationService;

/**
 * {@link AuthenticationService} implementation.
 */
public class AuthenticationServiceImpl implements AuthenticationService {

    /* (non-Javadoc)
     * @see com.developpez.myapplication.services.AuthenticationService#authenticate(java.lang.String, java.lang.String)
     */
    public ReturnMemento authenticate(String username, String password)
    {
        // create a test case where admin have ROLE_ADMIN and ROLE_USER
        if (username.equals("admin"))
        {
            // creating an authenticated user token for demo
            // regardless of username and password values
            GrantedAuthority[] authorities = new GrantedAuthority[] { new
            GrantedAuthorityImpl("ROLE_ADMIN"), new GrantedAuthorityImpl("ROLE_USER") };
            User user = new User("xxx", "yyy", true, true, true, true, authorities);
            Authentication auth = new UsernamePasswordAuthenticationToken(user, password, authorities);
            SecurityContext sc = new SecurityContextImpl();
            sc.setAuthentication(auth);
            SecurityContextHolder.setContext(sc);

            return new ReturnMemento(ReturnMemento.CODE_SUCCESS, ReturnMemento.CODE_SUCCESS);
        }
        // user only have ROLE_USER
        else if (username.equals("user"))
        {
            // creating an authenticated user token for demo
            // regardless of username and password values
            GrantedAuthority[] authorities = new GrantedAuthority[] { new
            GrantedAuthorityImpl("ROLE_USER") };
            User user = new User("xxx", "yyy", true, true, true, true, authorities);
            Authentication auth = new UsernamePasswordAuthenticationToken(user, password, authorities);
            SecurityContext sc = new SecurityContextImpl();
            sc.setAuthentication(auth);
            SecurityContextHolder.setContext(sc);

            return new ReturnMemento(ReturnMemento.CODE_SUCCESS, ReturnMemento.CODE_SUCCESS);
        }
        // other people can't log in
        else
        {
```

```

ReturnMemento rm = new ReturnMemento(ReturnMemento.CODE_ERROR, ReturnMemento.CODE_ERROR);
rm.getErrors().put("login", "login or password incorrect");
rm.getErrors().put("password", "login or password incorrect");
return rm;
    }
}

/* (non-Javadoc)
 * @see com.developpez.myapplication.services.AuthenticationService#logout()
 */
public void logout()
{
    SecurityContextHolder.clearContext();
}
}
    
```


Analysons un peu le bout de code ci-dessus. L'implémentation est très simple et autorise uniquement les utilisateurs *admin* et *user* à se connecter. Lors d'une authentification réussie, nous positionnons un ensemble de droits pour l'utilisateur, ces GrantedAuthority sont positionnés dans le contexte de sécurité pour cet utilisateur en particulier. Ici notre utilisateur *admin* a le ROLE_ADMIN et le ROLE_USER.


La fonction de logout se contente de réinitialiser le contexte de sécurité.

Nous noterons le bean ReturnMemento (non détaillé ici mais présent dans les sources de l'exemple) que nous réutiliserons régulièrement pour les validations de formulaires. Celui-ci permet de renvoyer une map des erreurs de la forme : nom du champ / erreur. Vous en comprendrez l'utilité par la suite côté client.

VII-D - Le contexte Spring

Nous écrivons le contexte Spring Security à part du premier fichier de contexte applicationContext_GWT.xml. Nous nommerons celui-ci : applicationContext-security_GWT.xml.

 *Veillez à bien l'inscrire dans les fichiers à lire par le ContextLoaderListener dans le fichier web.xml !*

Détaillons les différents beans positionnés dans ce fichier. (Cf.  [documentation officielle](#) pour plus d'explications). Tout d'abord le contexte principal :

```

<!--
Note: forcing eager session creation in
HttpSessionContextIntegrationFilter to ensure that session cookie is
set at the completion of the authenticate call.
-->
<security:http entry-point-ref="authenticationProcessingFilterEntryPoint"
create-session="always" access-denied-page="/index.html" session-fixation-protection="none">
<security:intercept-url pattern="/com.developpez.MyApplication/
MyApplication.html" access="ROLE_USER"/>
<security:form-login login-page="/index.html" />
</security:http>
    
```

Nous précisons ici les fonctionnalités que nous souhaitons utiliser, cela aura pour effet la création de filtres qui seront utilisés par la suite dans une chaîne de filtres web de Spring Security. Nous précisons ainsi que nous souhaitons intercepter les appels au module principal afin de vérifier que les autorisations (GrantedAuthority) du user incluent bien le role ROLE_USER. Nous affectons un filtre standard pour la vérification de la sécurité :

```

<!--
Standard spring filter used to redirect unauthorized request to the login form
Unauthorized request are intercepted by the pattern specified in the security namespace
above : eg the application itself.
Except for the main filter which prevent the access to the applicaton,
remember that here we are dealing with GWT
RPC requests and NOT normal web requests. The only sensible thing to
do is to, in GWT server-side code, explicitly handle exceptions thrown
by the AbstractSecurityInterceptor (Method Security interceptor in
this case) to either "redirect" the user to the login page or display
"access denied" messages.
-->
<bean id="authenticationProcessingFilterEntryPoint"
      class="org.springframework.security.ui.webapp.AuthenticationProcessingFilterEntryPoint">
  <property name="loginFormUrl" value="/index.html"/>
  <property name="forceHttps" value="false"/>
</bean>
    
```

Comme l'indiquent les commentaires, n'oubliez pas que nous allons gérer deux types de sécurité de façon distincte, la sécurité générale de la page de l'application et les autorisations nécessaires pour accéder aux données. Dans le premier cas, nous souhaitons alors utiliser Spring Security pour s'occuper de filtrer les requêtes. Dans le second cas nous utiliserons les annotations `@Secured` dans notre code pour autoriser ou non les appels à certaines méthodes. L'activation des annotations se faisant comme suit :

```

<security:global-method-security
  secured-annotations="enabled" jsr250-annotations="disabled" />
    
```

A noter que nous n'utiliserons pas d'"authentication provider" fourni par Spring Security. En effet nous souhaitons utiliser un formulaire GWT et gérer nous mêmes la vérification de l'authentification via le service écrit plus haut. Cependant celui-ci doit être déclaré dans le namespace du fichier xml, nous allons déclarer une classe pipeau pour cela en veillant à préciser : `custom-authentication-provider`.

```

<bean id="dummyAuthenticationProvider"
      class="com.developpez.security.core.DummyAuthenticationProvider">
  <security:custom-authentication-provider />
</bean>
    
```

En parlant de notre service d'authentification, il ne faut évidemment pas oublier de l'inscrire dans les urls gérées par notre GWTHandler dans le fichier `handler-servlet.xml` :

```

.....
  <entry key="/com.developpez.myapplication.Login/AuthenticationService.rpc" value-
ref="authenticationService" />
  <!-- because logout can also be called from the application module -->
  <entry key="/com.developpez.myapplication.Application/AuthenticationService.rpc" value-
ref="authenticationService" />
  .....
    
```

VII-E - La partie cliente

Pour gérer notre formulaire sur une page à part nous allons créer un second module : `Login.gwt.xml` :

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit 1.6.4//EN" "http://google-web-toolkit.googlecode.com/svn/tags/1.6.4/distro-source/core/src/gwt-module.dtd">
<module >

  <!-- Inherit the core Web Toolkit stuff. -->
  <inherits name='com.google.gwt.user.User' />

  <!-- Inherits smartGWT -->
  <inherits name="com.smartgwt.SmartGwt"/>

  <source path="model"/>
  <source path="client"/>
  <source path="services">
    <include name="**/*" />
    <exclude name="**/impl/**" />
  </source>

  <!-- Specify the app entry point class. -->
  <entry-point class='com.developpez.myapplication.client.LoginEntry' />
</module>
    
```

Pour le formulaire nous allons utiliser les formulaires dynamiques de SmartGWT ainsi que la notion de datasource :

```

DataSource dataSource = new DataSource();

DataSourceTextField login = new DataSourceTextField("login", "Username", 50, true);
DataSourcePasswordField password = new
DataSourcePasswordField("password", "Password", 50, true);
DataSourceBooleanField rememberField = new
DataSourceBooleanField("remember", "Remember me on this computer.", 50, false);

dataSource.setFields(login, password, rememberField);

final DynamicForm form = new DynamicForm();
form.setDataSource(dataSource);
form.setUseAllDataSourceFields(true);

IButton validateItem = new IButton("Log in");

....

VLayout formLayout = new VLayout(10);
formLayout.addMember(form);
formLayout.addMember(validateItem);
    
```

Ce bout de code permet de créer notre formulaire de login et d'y associer des règles basiques sur la présence des champs. L'API de SmartGWT permet aussi de rajouter des contrôles plus élaborés avec l'api `com.smartgwt.client.widgets.form.validator.Validator`. Graphiquement cela donne ceci :

Sign in
Sign up

Username :

Password :

Remember me on this computer.

Rajoutons maintenant le code du bouton :

```

validateItem.addClickHandler(new ClickHandler()
{
    public void onClick(ClickEvent event)
    {
        if (form.validate(false))
        {
            AuthenticationServiceAsync service = GWT.create(AuthenticationService.class);
            service.authenticate(form.getValueAsString("login"),
            form.getValueAsString("password"), new AsyncCallback<ReturnMemento>()
            {
                public void onSuccess(ReturnMemento result)
                {
                    if (result.getCode() == ReturnMemento.CODE_SUCCESS)
                    {
                        String path = Window.Location.getPath();
                        String modulePath = "/com.developpez.myapplication.Login/Login.html";
                        int index = path.indexOf(modulePath);
                        String contextPath = path.substring(0, index);

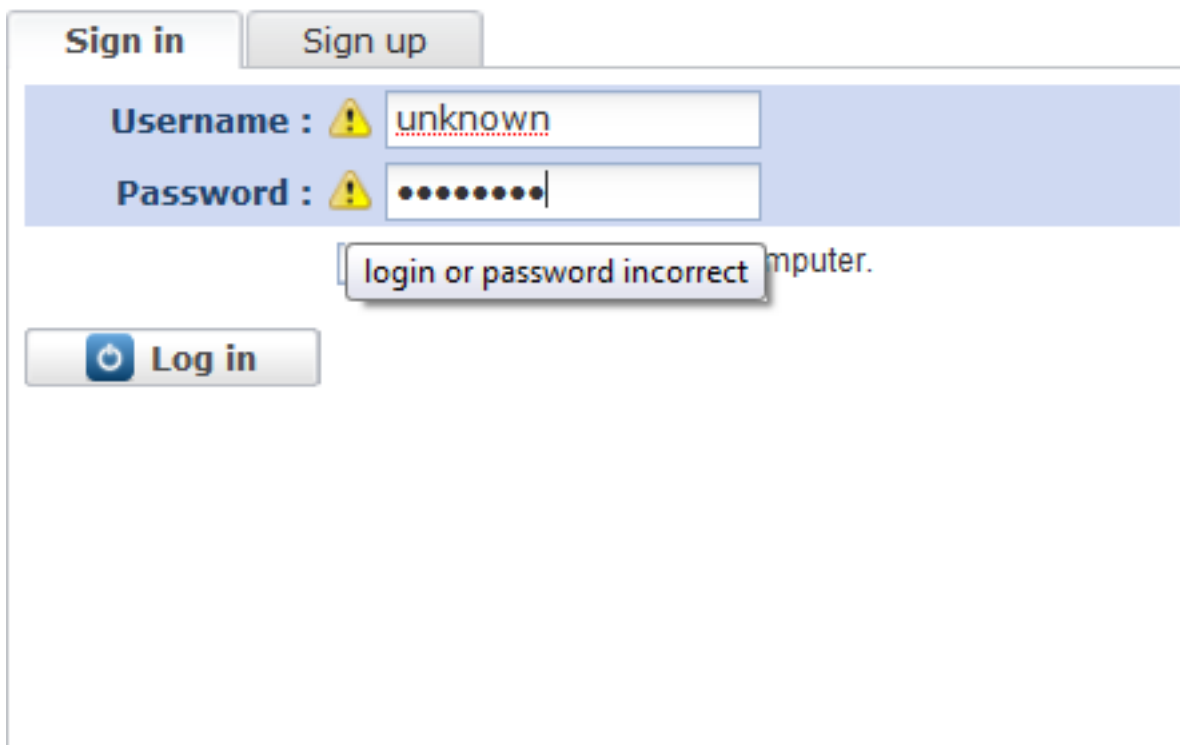
                        Window.open(contextPath+"/com.developpez.myapplication.Application/
Application.html", "_self", "");
                    }
                    else
                    {
                        form.setErrors(result.getErrors(), true);
                    }
                }

                public void onFailure(Throwable arg0)
                {
                    SC.say("error : " + arg0);
                }
            });
        }
    }
});
    
```

```
});
```

Dans le code ci-dessus, nous appelons tout d'abord `form.validate` qui permet de valider côté client avant d'appeler le service RPC. Dans vos futurs formulaires, n'oubliez pas de doubler vos validations côté serveur, la validation côté client n'est pas suffisante. Le retour est ensuite analysé via l'objet `ReturnMemento`, dont nous avons déjà parlé, afin de déterminer si nous pouvons rediriger vers le module principal ou non.

Voici d'ailleurs un résultat obtenu en ayant entré un mauvais nom d'utilisateur, la validation s'étant effectué côté serveur :



i A noter que la redirection après validation d'un formulaire est très rare en GWT. Ici c'est adapté car nous souhaitons vraiment rediriger notre utilisateur vers le module principal.

Vous aurez sans doute noté le petit bout de code permettant de récupérer le chemin de contexte :

```
String path = Window.Location.getPath();
String modulePath = "/com.developpez.myapplication.Login/Login.html";
int index = path.indexOf(modulePath);
String contextPath = path.substring(0, index);
```


Je n'ai pas trouvé mieux dans l'API GWT pour déterminer le contexte de l'application. C'est un peu bricolage mais faute de mieux, nous nous en contenterons, surtout que les redirections devraient être très rares.

VII-F - Sécuriser vos appels RPC

Nous avons désormais toute l'infrastructure nécessaire pour sécuriser nos méthodes. Nous allons donc modifier notre service abordé dans la première partie pour le sécuriser en fonction du rôle de l'utilisateur. Tout d'abord dans l'interface du service, nous allons utiliser l'annotation `@Secured` pour préciser le rôle nécessaire pour jouer une fonction :

```
/**
 *
 * @param record the project to add
 * @return the record added and potentially modified on server side
 * @throws ServiceSecurityException
 */
@Secured("ROLE_ADMIN")
Project add(Project record) throws ServiceSecurityException;
```


Nous précisons donc que l'ajout d'un projet ne peut se faire qu'avec le rôle `ROLE_ADMIN` (que nous aurons positionné dans le service d'authentification). Le service est désormais susceptible d'envoyer une exception de sécurité que nous définissons dans les exceptions translatables du package `services`.

Mais alors me direz vous, comment faire pour envoyer ce type d'exception alors que par défaut nos services vont envoyer des `SpringSecurityException` ? Pour cela je me suis inspiré du travail effectué sur  [GwtIncubatorSecurity](#) pour n'en tirer que l'essentiel. J'ai donc créé mon propre `GWTRPCServiceExporter` afin de surcharger la méthode permettant de renvoyer des exceptions vers l'appelant :

```
/* (non-Javadoc)
 * @see org.gwtwidgets.server.spring.GWTRPCServiceExporter#handleInvocationTargetException(
 *      java.lang.reflect.InvocationTargetException,
 *      java.lang.Object, java.lang.reflect.Method,
 *      com.google.gwt.user.server.rpc.RPCRequest)
 */
@Override
protected String handleInvocationTargetException(InvocationTargetException e, Object service,
Method targetMethod, RPCRequest rpcRequest) throws Exception
{
    Throwable cause = e.getCause();
    if (cause instanceof SpringSecurityException)
    {
        String failurePayload = RPC.encodeResponseForFailure(rpcRequest.getMethod(),
        new
        ServiceSecurityException(cause.getMessage()),
        rpcRequest.getSerializationPolicy());

        return failurePayload;
    }
    else
    {
        return super.handleInvocationTargetException(e, service, targetMethod, rpcRequest);
    }
}
```

Ceci me permet donc de convertir une `SpringSecurityException` vers une `ServiceSecurityException` pour la renvoyer côté client.

 *Vous aurez bien compris que renvoyer directement une `SpringSecurityException` aurait posé problème puisque celle-ci ne fait pas partie des exceptions connues côté client (non traduite en javascript).*

Pour injecter ce service exporter, il faut créer une factory à injecter dans le `GWTHandler` :

```
....  
<property name="serviceExporterFactory" ref="serviceExporterFactory"></property>  
....
```

Voilà, la partie sécurité est terminée pour l'instant. Nous avons utilisé Spring-security et mis en place les filtres nécessaires. Je vous laisse enrichir en fonction de vos besoins. Passons désormais à des services un peu plus représentatifs du monde réel.

VIII - Hibernate

VIII-A - Problématique

Nous allons aborder ici la dernière partie de cet article, l'intégration entre Hibernate et GWT. Pour cela nous allons tout de suite aborder la problématique principale, les collections. Si vous remplaciez l'implémentation de notre service ci-dessus par une implémentation simple sur une seule table (la table PROJECT) vous ne rencontriez en fait aucun problème et cet article pourrait vous sembler superflu au premier abord. C'est pourquoi nous allons dès le début ajouter une collection à notre exemple.

Partons de l'hypothèse que vous ayez déjà une implémentation de votre service ProjectService avec Hibernate et ajoutons à notre bean Project une collection d'auteur (le détail complet de l'implémentation Hibernate sera vu plus loin, ici il s'agit uniquement de mettre en évidence la problématique des collections) :

```
/**  
 * list of associated authors  
 */  
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)  
@JoinTable(name="project_authors", joinColumns= @JoinColumn( name="PROJECT_ID"),  
inverseJoinColumns=@JoinColumn( name="AUTHOR_ID"))  
private List<Authors> authors = new ArrayList<Authors>();
```

Le bean Authors ne comporte que deux propriétés, un ID et un NAME. Nous chargeons l'association sans utiliser le mode tardif (lazy).

Nous relançons notre application et nous observons que l'affichage de la liste des projets ne fonctionne plus et nous pouvons lire l'erreur suivante dans les logs :

```
[20:49:49.576][info][talledLocalContainer] java.lang.RuntimeException:  
com.google.gwt.user.client.rpc.SerializationException:  
java.lang.reflect.InvocationTargetException  
...  
[20:49:49.841][info][talledLocalContainer] Caused by:  
com.google.gwt.user.client.rpc.SerializationException:  
Type 'org.hibernate.collection.PersistentBag' was not included in the set of types which can be  
serialized by this SerializationPolicy  
or its Class object could not be loaded. For security purposes, this type will not be serialized.
```

Effectivement, notre bean Project a été instrumenté par Hibernate et il comporte désormais un PersistentBag qui ne peut être sérialisé par GWT.

En réalité le problème de la sérialisation n'est que la partie visible de l'iceberg, le problème est plus complexe que cela. En effet nous pourrions tenter de remplacer la liste avant l'export par une implémentation translatable en Javascript, comme une ArrayList. Cependant pour le cas d'une association tardive, cette liste serait non initialisée et nous enverrions une liste vide. Ce serait problématique dans le cas d'une mise à jour de GWT vers le serveur.

Il existe donc toute une problématique, qui n'est pas propre à GWT d'ailleurs, de la manipulation d'objets Hibernate détachés de leur session. Une partie de la grappe d'objet associé à mon bean est proxifié par une implémentation de collections propres à Hibernate. Elle peut être chargée à la demande en rattachant l'objet à la session. Cependant si nous envoyons cet objet en dehors du contexte de notre JVM (via WebServices, REST ou RPC) en remplaçant l'implémentation hibernate de la collection par une collection Java, alors l'objet perd toute connaissance des objets liés non chargés. Plus dangereux encore, si nous tentons de mettre à jour l'objet, nous allons demander à Hibernate de supprimer ces objets.

C'est toute cette problématique qu'adresse la librairie Gilead (ex Hibernate4Gwt). Cette librairie propose ainsi d'exporter les beans avec les collections d'objets chargées et de réconcilier ensuite ces collections au retour en ayant gardé les informations sur les proxy Hibernate.

GWT-SL propose de plus une intégration avec Gilead, nous allons donc la mettre en place.

VIII-B - Modèle

Tout d'abord voici le bean Project annoté pour la persistance :

```
package com.developpez.myapplication.model;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.OneToOne;
import javax.persistence.Table;

/**
 *
 */
@Entity
@Table(name="PROJECT")
public class Project implements Serializable
{
    /**
     * serial id
     */
    private static final long serialVersionUID = -5922364316829478146L;

    /**
     * name of the project
     */
    @Column(name="NAME")
    private String name;

    /**
     * description of the project
     */
    @Column(name="DESCRIPTION")
    private String desc;

    /**
     * url of the project
     */
}
```

```
@Column(name="URL")
private String url;

/**
 * list of associated authors
 */
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinTable(name="project_authors", joinColumns= @JoinColumn( name="PROJECT_ID"),
inverseJoinColumns=@JoinColumn( name="AUTHOR_ID"))
private List<Authors> authors = new ArrayList<Authors>();

/**
 * unique identifier
 */
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
private Integer id;

/**
 * Empty constructor
 */
public Project() {
}

/**
 * Full constructor
 * @param id
 * @param name
 * @param desc
 */
public Project(Integer id,String name,String desc)
{
    this.id = id;
    this.name = name;
    this.desc=desc;
}

/**
 * @return the list of authors
 */
public List<Authors> getAuthors() {
    return authors;
}

/**
 * @param the list of authors
 */
public void setAuthors(List<Authors> authors) {
    this.authors = authors;
}

/**
 * @return
 */
public String getName()
{
    return name;
}

/**
 * @param name
 */
public void setName(String name)
{
    this.name = name;
}

/**
 * @return
 */
public String getDesc()
```

```
{
    return desc;
}

/**
 * @param desc
 */
public void setDesc(String desc)
{
    this.desc = desc;
}

/**
 * @return
 */
public String getUrl()
{
    return url;
}

/**
 * @param url
 */
public void setUrl(String url)
{
    this.url = url;
}

/**
 * @param id
 */
public void setId(Integer id)
{
    this.id = id;
}

/**
 * @return
 */
public Integer getId()
{
    return id;
}

/* (non-Javadoc)
 * @see java.lang.Object#hashCode()
 */
@Override
public int hashCode()
{
    final int prime = 31;
    int result = 1;
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    return result;
}

/* (non-Javadoc)
 * @see java.lang.Object#equals(java.lang.Object)
 */
@Override
public boolean equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Project other = (Project) obj;
    if (id == null)
    {
```

```

        if (other.id != null)
            return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }
}

```

La seule chose à faire sera de modifier notre modèle afin que celui-ci étende la classe LightEntity :



```
public class Project extends LightEntity implements Serializable
```

Evidemment cela a un impact sur la compilation GWT puisque LightEntity ne fait pas partie des packages translatables en Javascript. Il est donc nécessaire de rajouter un héritage dans nos deux modules GWT :


```


<!-- Gilead light entity -->
<inherits name='net.sf.gilead.Adapter4Gwt15' />

```

On remarquera que cette première étape est assez intrusive dans notre modèle. Cependant ceci est dû au fait que j'ai choisi une stratégie  **stateless** (StatelessProxyStore), j'aurais pu l'éviter en utilisant une stratégie  **stateful** (HttpSessionProxyStore). J'ai cependant estimé naïvement qu'une stratégie stateful serait plus coûteuse côté serveur. Je n'ai cependant pas fait de tests pour déterminer l'overhead rajouté par le mode stateful.

VIII-C - Configuration spring

Je passe sur le détail de l'implémentation Hibernate en Spring, je vous laisse voir directement le code source rattaché à l'exemple. A savoir simplement que j'utilise une  **AnnotationSessionFactoryBean** afin d'utiliser les annotations et m'éviter d'utiliser des fichiers de mappings en xml. La partie persistence est configurée dans le fichier myapp-persistence-tech.xml.

Ce qui nous intéresse d'avantage concerne l'implémentation de Gilead. La documentation de GWT-SL est obsolète au moment de la rédaction de cet article puisqu'elle traite de Hibernate4Gwt. Nous suivrons donc en partie cette documentation ainsi que la  **documentation officielle**.

Dans le fichier myapp-persistence-tech.xml nous rajoutons quelques beans nécessaires à Gilead :

```

<!--
The Gilead POJO stateless proxy store.
-->
<bean id="proxyStore" class="net.sf.gilead.core.store.stateless.StatelessProxyStore" />

<bean id="persistenceUtil" class="net.sf.gilead.core.hibernate.HibernateUtil">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
<!--
Gilead persistentBeanManager binds the POJO store with a Hibernate session factory (through the hibernate util).
-->
<bean id="persistentBeanManager" class="net.sf.gilead.core.PersistentBeanManager">
  <property name="proxyStore" ref="proxyStore" />
  <property name="persistenceUtil" ref="persistenceUtil" />
</bean>

```

Ensuite il faudrait en théorie utiliser un HB4GWTRPCServiceExporter cependant nous avons déjà customisé notre exporteur pour la gestion de la sécurité. Nous allons donc modifier notre class custom pour hériter de cette classe :

```
public class GWTRPCSecuredServiceExporter extends HB4GWTRPCServiceExporter
```

Cela oblige aussi à modifier la factory qui doit créer ces exporters car ceux-ci ont besoin du `persistentBeanManager` que nous avons créé plus haut :

```
/**
 * Factory to use with {@link GWTRPCSecuredServiceExporter}
 */
public class GWTRPCSecuredServiceExporterFactory implements RPCServiceExporterFactory
{
    private PersistentBeanManager manager = new PersistentBeanManager();

    /* (non-Javadoc)
     * @see org.gwtwidgets.server.spring.RPCServiceExporterFactory#create()
     */
    public RPCServiceExporter create()
    {
        GWTRPCSecuredServiceExporter exporter = new GWTRPCSecuredServiceExporter();
        exporter.setBeanManager(manager);
        return exporter;
    }

    /**
     * @return the manager
     */
    public PersistentBeanManager getManager()
    {
        return manager;
    }


    /**
     * @param manager the manager to set
     */
    public void setManager(PersistentBeanManager manager)
    {
        this.manager = manager;
    }
}
```

Le `persistentBeanManager` est injecté dans la factory :

```
<bean id="serviceExporterFactory" class="com.developpez.server.spring.GWTRPCSecuredServiceExporterFactory" >
  <property name="manager" ref="persistentBeanManager"></property>
</bean>
```

VIII-D - Résultat

Et voilà, vous pouvez désormais relancer votre application et contempler le résultat.

 Profile Projects

Project	Description	Url
developpez		http://
google	google	google

Remove First

Remove First Selected

Remove All Selected

Add New

Terminé

IX - Conclusion

Lors de mon premier article sur GWT j'avoue être un peu passé à côté de la logique GWT. Ici, bien que débutant, j'ai pu plonger un peu plus au coeur de ce framework et j'espère vous avoir donné les billes pour travailler réellement avec. Il manque une partie importante cependant à cet article car je n'ai pas voulu alourdir cet article déjà assez conséquent, les tests unitaires. Je vous invite grandement à ne pas les oublier lors de vos développements.

D'un point de vue personnel, je reste assez critique sur GWT même si je reconnais un certain potentiel. Il me semble que GWT et les outils autour de GWT manque encore un peu de maturité. Par exemple :

- Plugin maven codehaus encore en incubator
- Processus de build assez lent
- GWT-SL semble avoir peu d'activité
- Toute la littérature autour de la sécurité est encore expérimentale et peu de standards se dégagent
- Bibliothèques de Widget encore assez jeunes

















Cependant un développeur Java dispose désormais d'un bon panel de choix pour faire une application ergonomiquement riche entre JavaFX, GWT, JSF Icefaces ou l'utilisation directe de bibliothèques Javascript. Gageons que des bonnes pratiques émergeront rapidement de tout cela.

IX-A - Remerciements

Je tiens à remercier Ricky81, Ridekick et Benwit pour leurs corrections et leur aide à la publication de cet article.

IX-B - Références

Tout d'abord les **sources complètes** de l'exemple présenté ici. Et les différents articles ayant permis la rédaction de celui-ci.

-  **GWT**
-  **gwt-maven**
-  **SmartGWT**
-  **Spring-security**
-  **Spring core**
-  **GWT eclipse plugin**
-  **GWT-SL**
-  **Gilead**
-  **plugin pour eclipse**
-  **création d'un projet SmartGWT avec Maven.**
-  **How to Integrate GWT with Spring 2.0 (gwt incubator)**
-  **Luca Masini**
-  **Chris Lee**
-  **Spring4gwt**
-  **Seewah**
-  **Pgt**

Et n'oubliez pas le  **forum**.