

Middleware – Web Service Concat – GSOAP - MAVEN

1^{er} février 2016

1

Le service WSConcat

mAVEN vous permet de gérer intégralement votre projet sans vous soucier des problèmes de dépendances entre les différentes librairies importées. La particularité de **maven** est de s'appuyer sur un ensemble de conventions pour compiler du code, créer des fichiers **JAR**, déployer des **WAR**, etc. On se propose dans la suite d'étudier le fonctionnement de **maven**.

1. Veuillez télécharger **maven** 3.0.3 (<http://maven.apache.org/download.html>)
2. Désarchiver **maven** dans votre *home directory*, mettre à jour la variable **PATH** de votre *shell*.

```
# export M2_HOME=répertoire_maven  
# export PATH=$PATH:$M2_HOME/bin
```

3. Saisir la commande suivante afin de commencer un projet **maven** nommé **WSConcatService**

```
# mvn archetype:generate -DgroupId=net.tuto2.ws.service.concat  
-DartifactId=WSConcatService -DarchetypeArtifactId=maven-  
archetype-webapp
```

4. La structure de votre répertoire devrait avoir la forme suivante :

```

WSConcatService
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   ├── resources
│   │   ├── webapp
│   │   └── WEB-INF
│   └── target

```

- ▮ pom.xml, ce fichier décrit en détail votre projet Java
- ▮ src, contient les fichiers sources de votre projet
- ▮ main/java, contient les fichiers source Java qui seront inclus dans les packages JAR, WAR générés etc...
- ▮ main/webapp, contient les pages .jsp
- ▮ main/WEB-INF, contient les fichiers de configuration de la servlet
- ▮ target, ce répertoire rassemble tout le code généré à partir du code source, bytecode, etc...

Certains répertoires peuvent ne pas être créés comme par exemple src/main/resources. Dans ce cas, créer les à la main.

5. Editez le fichier pom.xml présent à la racine de WSConcatService. Ajouter les dépôts à partir duquel maven pourra télécharger les librairies nécessaires pour compiler votre projet, et en particulier le dépôt correspondant à toutes les librairies Java.

Listing 1– pom.xml

```

Line 1 <repositories>
-     <repository>
-         <id>maven2-repository.dev.java.net</id>
-         <name>Java.net Repository for Maven</name>
5         <url>https://maven.java.net/content/repositories/releases</url>
-     </repository>
- </repositories>

```

- ▮ Il ne peut y avoir qu'une seule balise xml <repositories> dans un fichier pom.xml

6. En cas de problèmes avec les dépôts distants, vous pouvez ajouter le dépôt suivant <http://www.labri.fr/perso/bromberg/repo/>.

Listing 2– pom.xml ➤

```

Line 1  <repositories>
-      ...
-      <repository>
-          <id>maven2-repository.dev.java.net</id>
5          <name>Java.net Repository for Maven 2</name>
-          <url>http://download.java.net/maven/2/</url>
-      </repository>
-      <repository>
-          <id>maven2-bromberg.dev.java.net</id>
10         <name>Java.net Repository for Maven 2</name>
-          <url>http://www.labri.fr/perso/bromberg/repo/</url>
-      </repository>
-      ...
-  </repositories>

```

7. Editez le fichier `pom.xml` présent à la racine de `WSConcatService`. Indiquer la version du compilateur que vous souhaitez utiliser pour compiler l'ensemble du projet. En particulier, nous choisissons la version 1.5 car nous allons utiliser des annotations dans le code source Java.

Listing 3– pom.xml ➤

```

Line 1  <project>
-      ...
-      <build>
-          <plugins>
5          <plugin>
-              <groupId>org.apache.maven.plugins</groupId>
-              <artifactId>maven-compiler-plugin</artifactId>
-              <version>2.0.2</version>
-              <configuration>
10                 <source>1.5</source>
-                 <target>1.5</target>
-             </configuration>
-          </plugin>
-        </plugins>
15      </build>
-      ...
-  </project>

```

- Il ne peut y avoir qu'une seule balise xml `<build>` dans un fichier `pom.xml`
- Il ne peut y avoir qu'une seule balise xml `<plugins></plugins>` incluse dans la balise `<build>` dans un fichier `pom.xml`

8. Spécifier les dépendances des librairies que nous allons utiliser au sein du projet. En particulier, nous allons manipuler le **framework** **Métro** qui implémente la spécification **jax-ws** fournit par **ORACLE/SUN**. Vous pouvez trouver davantage d'information sur le site web du **framework** : <https://metro.java.net/>

Listing 4– pom.xml [download](#)

```
Line 1  <project>
-      ...
-      <dependencies>
-          <dependency>
5          <groupId>org.glassfish.metro</groupId>
-          <artifactId>webservices-rt</artifactId>
-          <version>2.3</version>
-          </dependency>
-      </dependencies>
10     ...
- </project>
```

▮ Il ne peut y avoir qu'une seule balise xml `<dependencies></dependencies>` incluse dans la balise dans un fichier pom.xml

9. Vous allez créer un service Web selon la méthode last contract. Cette méthode vous évite d'avoir à écrire vous même l'interface WSDL de votre service. Cela vous permet dans un premier temps de vous familiariser avec les documents WSDL, et XML. L'interface du service, c'est à dire son document WSDL associé, est généré automatiquement à partir d'une interface Java annotée qui représente votre service Web WSConcat. Pour connaître la signification approfondie de toutes les annotations disponibles, vous pouvez consulter la documentation suivante. <http://jax-ws.java.net/nonav/2.1.3/docs/annotations.html>

10. Créer une interface Java avec une méthode permettant de concaténer deux chaînes de caractères. Cette interface doit se trouver dans le package `net.tuto2.ws.service.concat` dans le répertoire `src/main/java`

Listing 5– WSConcat.java - Interface du service WSConcat [download](#)

```

Line 1 package net.tuto2.ws.service.concat;
-
- import javax.jws.WebMethod;
- import javax.jws.WebParam;
5 import javax.jws.WebResult;
- import javax.jws.WebService;
- import javax.jws.soap.SOAPBinding;
-
- @WebService(targetNamespace = "http://david.bromberg.fr/service/concat",
-             name = "WSConcat")
10 @SOAPBinding(style = SOAPBinding.Style.RPC)
- public interface WSConcat {
-     @WebMethod
-     public String concat(
-         @WebParam(name = "str1")
15         String str1,
-         @WebParam(name = "str2")
-         String str2
-     );
- }

```

11. Indiquer à quoi servent les annotations.
12. Dorénavant, vous allez implémenter le service dans une classe appelée `WSConcatImpl` dans le package `net.tuto2.ws.service.concat`.

Listing 6– WSConcatImpl.java - Implémentation du service [download](#)

```

Line 1 package net.tuto2.ws.service.concat;
-
- import javax.jws.WebParam;
- import javax.jws.WebService;
5
- @WebService(endpointInterface = "net.tuto2.ws.service.concat.WSConcat",
-             serviceName = "MyConcatAgent")
- public class WSConcatImpl {
-
-     public String concat (String str1,String str2){
10     return str1+str2;
-     }
- }

```

13. L'étape suivante est de générer le document WSDL correspondant à l'interface Java WSConcat. Pour cela, on utilise l'outil wsgen disponible via le plugin jaxws-maven-plugin. Ajouter une cible de compilation dans le fichier `pom.xml` du projet service. L'outil wsgen prend en paramètre un Service End Interface (sei) : la classe Java à partir de laquelle est générée le document WSDL. On active la génération du document WSDL afin de vérifier sa correspondance avec la classe Java.

Listing 7– `pom.xml` [download](#)

```

Line 1  <project>
-      ...
-      <build>
-      ...
5      <plugins>
-      ...
-      <plugin>
-          <groupId>org.jvnet.jax-ws-commons</groupId>
-          <version>2.3</version>
10         <artifactId>jaxws-maven-plugin</artifactId>
-         <executions>
-             <execution>
-                 <goals>
-                     <goal>wsgen</goal>
15                 </goals>
-                 <configuration>
-                     <sei>net.tuto2.ws.service.concat.WSConcatImpl</sei>
-                     <genWsdL>true</genWsdL>
-                     <keep>true</keep>
20                 </configuration>
-             </execution>
-         </executions>
-     </plugin>
-     ...
25 </plugins>
-     ...
- </build>
- </project>

```

14. Saisir la commande `mvn clean install`. Vérifier/Analyser les documents WSDL générés dans le répertoire `target/generated-sources/wsdl/`. Le service est créé;-).

Afin d'éviter l'erreur suivante : **[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent !** indiquer dans votre fichier `pom.xml` que les fichiers utilisés sont basés sur l'encodage `UTF-8` grâce aux balises suivantes :

Listing 8– `pom.xml` [download](#)

```
Line 1  <project>
-      ...
-      <properties>
-          <project.build.sourceEncoding>
5          UTF-8
-          </project.build.sourceEncoding>
-      </properties>
-      ...
-  </project>
```

2

Déploiement du service Web

1. Avant de pouvoir déployer votre service, il faut le configurer afin qu'il puisse fonctionner dans un conteneur de servlet. Rendez vous dans le répertoire `WSConcatService/src/main/webapp/WEB-INF`. Et éditer le fichier `web.xml` comme suit :

Listing 9– web.xml

```
Line 1  <?xml version="1.0" encoding="UTF-8"?>
-      <web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
-            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
-            xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
5         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
-      <display-name>Sample web service provider</display-name>
-      <listener>
-        <listener-class>com.sun.xml.ws.transport.http.servlet.
          WSServletContextListener</listener-class>
-      </listener>
10     <servlet>
-       <servlet-name>WebServicePort</servlet-name>
-       <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-
         class>
-       <load-on-startup>1</load-on-startup>
-     </servlet>
15    <servlet-mapping>
-     <servlet-name>WebServicePort</servlet-name>
-     <url-pattern>/services/*</url-pattern>
-   </servlet-mapping>
- </web-app>
```


2. Le fichier `web.xml` est standardisé par la JSR-315. On retiendra les balises suivantes qui sont fondamentales pour le bon fonctionnement de votre *Web Service*.

- ▀ Les balises `<servlet>...</servlet>`, permettent d'indiquer quelle servlet sera chargée par le serveur d'applications. En l'occurrence, on utilise, dans le contexte de ce TD/TP, la pile Web Service de Sun (nom de code Metro). Par conséquent, on utilise la classe `com.sun.xml.ws.transport.http.servlet.WSServlet` qui va s'occuper de réceptionner toutes les requêtes HTTP.
- ▀ Les balises `<servlet-mapping>...</servlet-mapping>`, permettent de spécifier quelle servlet sera exécutée suivant l'URL demandée par le client HTTP.
- ▀ Les balises `<url-pattern>...</url-pattern>`, permettent de définir un motif qui déclenchera la redirection de la requête http vers la servlet adéquate. Par exemple, `<url-pattern>/services/*</url-pattern>` va mapper toutes les requêtes ayant dans l'URL demandée le motif `/services/` vers la servlet ayant le nom `WebServicePort`.
- ▀ Les balises `<listener-class>...</listener-class>`, spécifient quelle est la classe Java qui va configurer/gérer le cycle de vie de la servlet.

3. Enfin, la servlet de Sun, qui va exécuter votre service et potentiellement plusieurs autres services, utilise un fichier de configuration nommé `sun-jaxws.xml`. Créer ce fichier dans le répertoire `WSConcatService/src/main/webapp/WEB-INF` et peupler le comme suit :

Listing 10– `sun-jaxws.xml`

```
Line 1 <?xml version="1.0" encoding="UTF-8"?>
- <endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
-   version="2.0">
-   <endpoint
5       name="test1"
-       implementation="net.tuto2.ws.service.concat.WSConcatImpl"
-       url-pattern="/services/concat"/>
- </endpoints>
```

- ▀ La balise `<endpoint/>`, permet d'indiquer quel service web sera en mesure de répondre à l'invocation du client. Par conséquent, indiquer ici dans l'attribut `implementation` votre classe Java qui implémente le service.
- ▀ L'attribut `url-pattern`, permet de spécifier un motif qui déclenchera la redirection de la requête vers la servlet susmentionnée.

<http://www.oracle.com/technetwork/java/javaee/servlet/index.html>

4. Modifier le fichier `pom.xml` à la racine de votre répertoire `WSConcatService`, pour indiquer quel conteneur de servlet vous souhaitez utiliser. Dans un premier temps, nous allons utiliser le conteneur Jetty.

Listing 11– sun-jaxws.xml ➡

```

Line 1  <project>
-      ...
-      <build>
-        <plugins>
5         ...
-         <plugin>
-           ...
-         </plugin>
-         <plugin>
10        <groupId>org.eclipse.jetty</groupId>
-        <artifactId>jetty-maven-plugin</artifactId>
-        <version>9.0.5.v20130815</version>
-        </plugin>
-        ...
15       </plugins>
-     </build>
-     ...
- </project>

```

5. Une fois le plugin Jetty configuré, vous pouvez faire un `mvn clean install` pour télécharger le plugin. Enfin, vous pouvez lancer votre application web grâce à la commande `mvn jetty:run` dans votre *shell*. Pour plus d'informations sur le fonctionnement du plugin, se rendre à l'url suivante http://wiki.eclipse.org/Jetty/Feature/Jetty_Maven_Plugin. On retiendra en particulier les propriétés suivantes :

- ➡ Par défaut le serveur Jetty est accessible à l'url <http://localhost:8080>
- ➡ Jetty scanne périodiquement les fichiers du projet, et s'il y a eu des modifications sur les fichiers `.class`, l'application webapp est redéployée.
- ➡ Jetty fonctionne jusqu'à ce qu'il soit arrêté par un `<ctrl-c>`

6. Vous pouvez maintenant surfer sur la page du service Web, comme illustré dans la Figure ??, en allant à l'url suivante : <http://localhost:8080/WSConcatService>.

7. Pour obtenir la description WSDL de votre service Web fraîchement déployé, il suffit de saisir l'url suivante dans votre navigateur. <http://localhost:8080/WSConcatService/services/concat>

Par défaut, le chemin d'accès du **service web** correspond à la valeur du **artifactId** du projet. Attention toutefois, suivant la version de **jetty**, cela n'est pas toujours le cas (le contexte est alors la racine de l'url, c-a-d la /). Dans ces conditions, il est possible de le spécifier explicitement dans la configuration du plugin. Par exemple :

Listing 12– Configuration du plugin Jetty

```
Line 1 <configuration>
-   <webApp>
-       <contextPath>/${project.artifactId}</contextPath>
-   </webApp>
5   <webAppSourceDirectory>${basedir}/src/main/webapp</webAppSourceDirectory>
-   <webXml>${basedir}/src/main/webapp/WEB-INF/web.xml</webXml>
- </configuration>
```

8. Créer un nouveau service qui effectue une addition de nombres.

3

Création des clients

Il est possible de créer des clients aussi bien en Java qu'en C. est possible de créer des clients aussi bien en Java qu'en C. est possible de créer des clients aussi bien en Java qu'en C.

Conception d'un client Java

1. Afin de développer un client pour votre service `WSConcatService`, créez un nouveau projet Maven avec un `groupID` égal à `net.tuto2.ws.client`, un `artifactID` `WSConcatClient`, et enfin un `archetypeArtifactID` égal à `maven-archetype-quickstart`.
2. La structure de votre répertoire devrait avoir la forme suivante :

```

WSConcatClient
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   └── resources
└── target
  
```

- ▀ `pom.xml`, ce fichier décrit en détail votre projet Java
- ▀ `src`, contient les fichiers sources de votre projet
- ▀ `main/java`, contient les fichiers source Java qui seront inclus dans les packages JAR, WAR générés etc...
- ▀ `target`, ce répertoire rassemble tout le code généré à partir du code source, bytecode, etc...

Certains répertoires peuvent ne pas être créés comme par exemple `src/main/resources`. Dans ce cas, créer les à la main.

3. On se place dorénavant dans le répertoire du projet client. Pour coder le client nous allons bien évidemment réutiliser le document WSDL de votre service accessible à l'url <http://localhost:8080/WSConcatService/services/concat>. Vous allez utiliser l'utilitaire `wsimport` pour générer les stubs nécessaires pour invoquer votre service distant `WSConcat`. Configurer le fichier `pom.xml` de façon à ajouter l'exécution de l'action `wsimport`.

Listing 13– pom.xml


```

Line 1 <plugin>
-   <groupId>org.jvnet.jax-ws-commons</groupId>
-   <artifactId>jaxws-maven-plugin</artifactId>
-   <version>2.3</version>
5   <configuration>
-   <sourceDestDir>${basedir}/src/main/java</sourceDestDir>
-   <wsdlUrls>
-   <wsdlUrl>
-       http://localhost:8080/WSConcatService/services/concat?wsdl
10  </wsdlUrl>
-   </wsdlUrls>
-   <packageName>net.tuto2.ws.client</packageName>
-   </configuration>
-   <executions>
15  <execution>
-   <goals>
-   <goal>wsimport</goal>
-   </goals>
-   </execution>
20  </executions>
-   </plugin>

```

4. Vérifier que des fichiers Java ont bien été générés à partir du document WSDL du service distant *via* les commandes `mvn clean install`.
5. Suivant le code Java généré vous devriez être en mesure de coder le client en utilisant le fichier `App.java`.

6. Pour exécuter le client, rajouter une action d'exécution au `pom.xml` du client afin de modifier le cycle de vie du projet.

Listing 14– pom.xml 

```

Line 1 <plugin>
-   <groupId>org.codehaus.moj</groupId>
-   <artifactId>exec-maven-plugin</artifactId>
-   <executions>
5     <execution>
-       <goals>
-         <goal>exec</goal>
-       </goals>
-     </execution>
10  </executions>
-   <configuration>
-     <executable>java</executable>
-     <arguments>
-       <argument>-classpath</argument>
15    <classpath />
-     <argument>
-       -Dcom.sun.xml.ws.transport.http.client.HttpTransportPipe.dump=true
-     </argument>
-     <argument>
20      net.tuto2.ws.client.App
-     </argument>
-   </arguments>
- </configuration>
- </plugin>

```

7. Exécuter le client via la commande `mvn exec:exec;-)`.

Conception d'un client C

1. Créer un client Web Service concat en langage C à l'aide des outils `wsdl2h` et `soapcpp2`. Aidez vous de l'aide en ligne de commande (`man wsdl2h`, `man soapcpp2`, et/ou en consultant l'url suivante <http://www.cs.fsu.edu/~engelen/soap.html>).
2. Ecrire un `makefile` permettant de générer automatiquement les headers et les fichiers nécessaires pour compiler concat.
3. Tester le bon fonctionnement de votre programme concat à partir de votre web service WSConcat Java.

4

Création d'un client web

Grâce à la version 3.0 de l'API des servlets, le fichier `web.xml` n'est plus nécessaire. Le support d'exécution de JAX-WS est capable d'enregistrer dynamiquement une servlet pour chaque service définit dans le fichier `sun-jaxws.xml`. Ce dernier fichier reste toutefois nécessaire pour des conteneurs autre que Glassfish afin d'indiquer quels sont les `web service` à déployer. Toutefois, si l'on souhaite réaliser une configuration avancée des servlets à déployer, on peut toujours utiliser comme d'habitude le fichier `web.xml`.

1. Afin de développer un client pour votre service `WSConcatService`, créez un nouveau projet Maven avec un `groupId` égal à `net.tuto2.webclient`, un `artifactID` égal à `WSConcatWebClient`, et enfin un `archetypeArtifactID` égal à `maven-archetype-webapp`.

2. On se place dorénavant dans le répertoire du projet client. Pour coder le client nous allons bien évidemment réutiliser le document WSDL de votre service accessible à l'url <http://localhost:8080/WSConcatService/services/concat>. Vous allez utiliser l'utilitaire `wsimport` pour générer les stubs nécessaires pour invoquer votre service distant `WSConcat`. Configurer le fichier `pom.xml` de façon à ajouter l'exécution de l'action `wsimport`.

Listing 15– pom.xml

```
Line 1  <plugin>
-      <groupId>org.jvnet.jax-ws-commons</groupId>
-      <artifactId>jaxws-maven-plugin</artifactId>
-      <version>2.3</version>
5      <executions>
-      <execution>
-      <goals>
-      <goal>wsimport</goal>
-      </goals>
10     <phase>generate-sources</phase>
-     <configuration>
-     <sourceDestDir>${basedir}/src/main/java</sourceDestDir>
-     <wsdlUrls>
-     <wsdlUrl> http://localhost:8080/services/concat?wsdl </wsdlUrl>
15    </wsdlUrls>
-     <packageName>net.tuto2.ws.webclient</packageName>
-     </configuration>
-     </execution>
-     </executions>
20    </plugin>
```


A noter toutefois que dès lors que l'on supprime le fichier `web.xml`, le plugin maven qui crée l'archive `.war` provoque une erreur se plaignant de l'absence de ce fichier : **[ERROR] Failed to execute goal org.apache.maven.plugins:maven-war-plugin:2.1.1:war (default-war) on project WSConcatService : Error assembling WAR : webxml attribute is required (or pre-existing WEB-INF/web.xml if executing in update mode) -> [Help 1]**. Etant donné que ce fichier est devenu facultatif, on peut indiquer à maven d'ignorer ce message d'erreur en modifiant le fichier `pom.xml` de votre projet comme suit :

Listing 16- `pom.xml` ♥♥

```
Line 1  <project>
-      ...
-      <build>
-      ...
5      <plugin>
-        <groupId>org.apache.maven.plugins</groupId>
-        <artifactId>maven-war-plugin</artifactId>
-        <configuration>
-          <failOnMissingWebXml>false</failOnMissingWebXml>
10       </configuration>
-     </plugin>
-     ...
-     </build>
-     ...
15    </project>
```

3. Créer votre servlet cliente avec les méthodes traditionnelles `doGet` et `doPost`.

Listing 17– ConcatServlet.java

```
Line 1  /**
-      * Handles the HTTP <code>GET</code> method.
-      * @param request servlet request
-      * @param response servlet response
5      * @throws ServletException if a servlet-specific error occurs
-      * @throws IOException if an I/O error occurs
-      */
-      @Override
-      protected void doGet(HttpServletRequest request, HttpServletResponse
-                           response)
10     throws ServletException, IOException {
-         processRequest(request, response);
-     }
-
-      /**
15     * Handles the HTTP <code>POST</code> method.
-     * @param request servlet request
-     * @param response servlet response
-     * @throws ServletException if a servlet-specific error occurs
-     * @throws IOException if an I/O error occurs
20     */
-     @Override
-     protected void doPost(HttpServletRequest request, HttpServletResponse
-                           response)
-     throws ServletException, IOException {
-         processRequest(request, response);
25     }
}
```

4. Enfin, ajouter le code correspondant aux méthodes `processRequest()` et `concat()` manquantes. La méthode `concat()` effectue un appel au web service `WSConcat` grâce à la classe `MyAgentConcat` qui a été générée lors de la compilation, et qui correspond donc au `stub` du client.

Listing 18– ConcatServlet.java ➤

```

Line 1  /**
-      * Processes requests for both HTTP <code>GET</code>
-      * and <code>POST</code> methods.
-      * @param request servlet request
5       * @param response servlet response
-      * @throws ServletException if a servlet-specific error occurs
-      * @throws IOException if an I/O error occurs
-      */
-      protected void processRequest(HttpServletRequest request,
10         HttpServletResponse response)
-      throws ServletException, IOException {
-
-          response.setContentType("text/html;charset=UTF-8");
-
15         try (PrintWriter out = response.getWriter()) {
-
-             out.println("<html lang=\"en\">");
-             out.println("<head>");
-             out.println("<title>Servlet HelloServlet</title>");
20             out.println("</head>");
-             out.println("<body>");
-             out.println("<h1>Servlet HelloServlet at " +
-                 request.getContextPath () + "</h1>");
-             out.println("<p>" + concat("hello ", "david") + "</p>");
25             out.println("</body>");
-             out.println("</html>");
-
-         }
-     }

```

Listing 19– ConcatServlet.java ➤

```

Line 1  private String concat(String arg0, String arg1) {
-      service = new MyConcatAgent();
-      WSConcat port = service.getWSConcatImplPort();
-      return port.concat(arg0, arg1);
5      }

```

5. Le client doit s'exécuter dans un conteneur de servlet, par conséquent on va également utiliser **Jetty** comme support d'exécution. Dans notre exemple, étant donné que l'on a déjà lancé un container **Jetty** sur le port 8080, on va configurer le nouveau container sur un port différent, c-a-d le port 8085.

Listing 20– pom.xml ➤➤

```

Line 1  <plugin>
-       <groupId>org.eclipse.jetty</groupId>
-       <artifactId>jetty-maven-plugin</artifactId>
-       <version>9.0.5.v20130815</version>
5       <configuration>
-       <jettyXml>src/main/resources/jetty.xml</jettyXml>
-       <!-- <webApp> -->
-       <!-- <contextPath>/yourCtxPath</contextPath> -->
-       <!-- </webApp> -->
10      </configuration>
-      </plugin>

```

La configuration de Jetty se fait dans un fichier XML externe `jetty.xml` à créer dans le répertoire `src/main/resources`.

Listing 21– jetty.xml ➤➤

```

Line 1  <?xml version="1.0"?>
-       <!DOCTYPE Configure PUBLIC "-//Jetty//Configure//EN" "http://www.eclipse.org/
-           jetty/configure.dtd">
-
-       <Configure id="Server" class="org.eclipse.jetty.server.Server">
5       <Call id="httpConnector" name="addConnector">
-           <Arg>
-               <New class="org.eclipse.jetty.server.ServerConnector">
-                   <Arg name="server"><Ref refid="Server" /></Arg>
-                   <Set name="port"><Property name="jetty.port" default="8085" /></Set>
10                  <Set name="idleTimeout">30000</Set>
-               </New>
-           </Arg>
-       </Call>
-
15      </Configure>

```