COURS DE JAVA – Java Web Java/JEE - Les Servlets
El Hadji Gaye - AlizNet

Avec Netbeans.

Auteur	El Hadji Gaye
Pour	Ecole
Date	12/03/18
Objet	Java/Web Java/J2ee - Les Servlets.

I) Introduction ... 3
II) Vocabulaire ... 4
III) Rappels sur le protocole HTTP. ... 5
IV) Architecture Java Web et Java EE. ... 7
V) Fonctionnement d'un serveur d'application. ... 11
VI) Introduction aux Servlets. ... 23
VII) Notre première Servlet. ... 24
VIII) Notion de requête HTTP avec HttpServletRequest. ... 55
IX) Notion de réponse HTTP avec ServletResponse et HttpServletResponse. ... 58
X) L'API JSTL (Tags JSTL). ... 60
XI) Notion de portée de variable JSTL. ... 65
XII) Manipuler un JavaBean. ... 66
XIII) Notion de Filtre de Servlet. ... 70
XIV) Notion de Listener de Servlet. ... 79
XV) Application 3 couches en Java Web. ... 88
XVI) Exercice d'Application sur l'implémentation 3 couches. ... 95

I) Introduction

J2EE est l'acronyme de Java 2 Entreprise Edition. Cette édition est dédiée à la réalisation d'applications pour entreprises. J2EE est basée sur J2SE (Java 2 Standard Edition) qui contient les API de base de Java. J2EE désigne aussi les technologies Java utilisées pour créer des applications d'entreprise avec le JDK 1.4 ou antérieur.
Depuis la version du JDK 1.5, J2EE est renommée Java EE (Enterprise Edition).
Les termes Java EE 5, Java EE 6 et Java EE 7 ont alors été utilisés pour désigner l'ensemble des technologies qui concourent à créer une application d'entreprise avec la plate-forme Java.

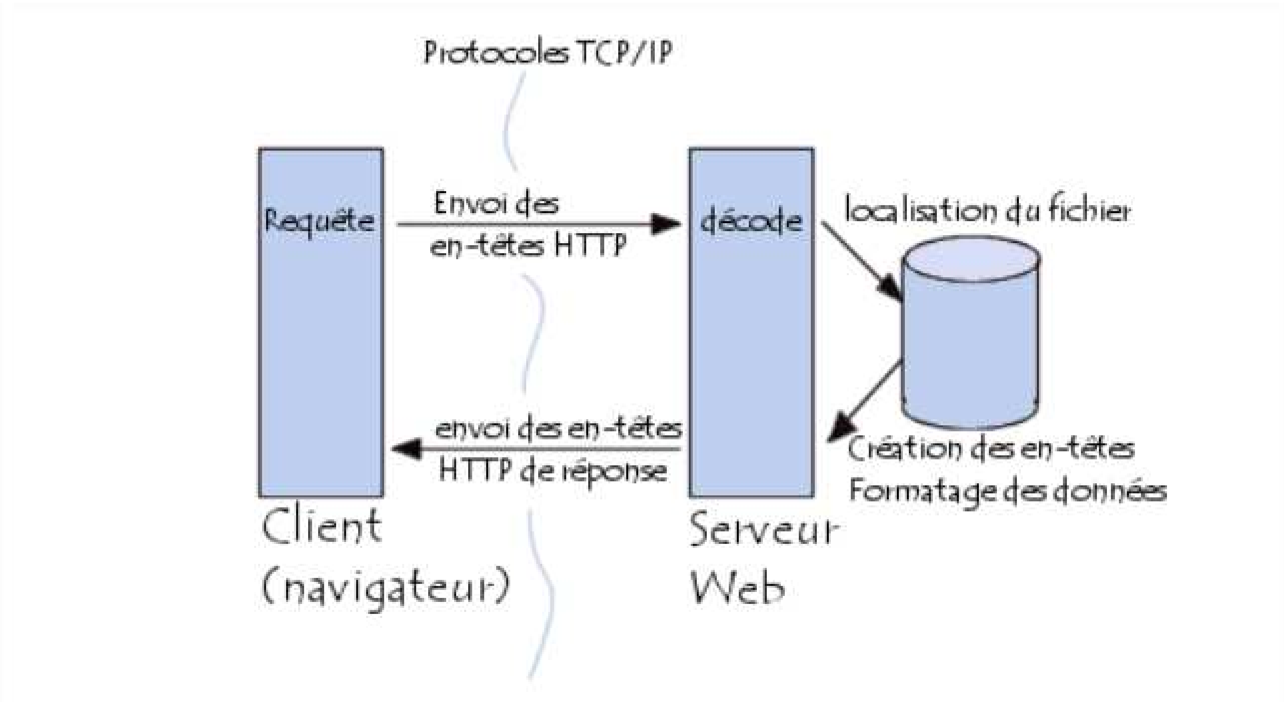
II) Vocabulaire

- **API** : Signifie Application Programming Interface. Ce qui veut dire que c'est un ensemble de bibliothèques et librairies dédié pour implémenter une fonctionnalité donnée.
- **Servlet** : Une servlet est une classe Java qui permet de créer dynamiquement des données au sein d'un serveur HTTP. Ces données sont le plus généralement présentées au format HTML, mais elles peuvent également l'être au format XML ou tout autre format destiné aux navigateurs web. Les servlets utilisent l'API Java Servlet (package **javax.servlet**). Une servlet s'exécute dynamiquement sur le serveur web et permet l'extension des fonctions de ce dernier, typiquement : accès à des bases de données, transactions d'e-commerce, etc. Une servlet peut être chargée automatiquement lors du démarrage du serveur web ou lors de la première requête du client ; une fois chargées, les servlets restent actives dans l'attente d'autres requêtes du client.
- **Bean** : le « **Bean** » (ou haricot en français) est une technologie de composants logiciels écrits en langage Java. Les **Beans** sont utilisés pour encapsuler plusieurs objets dans un seul objet. Le « **Bean** » regroupe alors tous les attributs des objets encapsulés. Ainsi, il représente une entité plus globale que les objets encapsulés de manière à répondre à un besoin métier.
- **JSP** : Java Server Pages.
- **JSF** : Java Server Faces.
- **EJB** : Entreprise Java Bean.
- **EAR** : Un EAR (pour Enterprise Application Archive) est un format de fichier utilisé par Java EE pour empaqueter (en) un ou plusieurs modules dans une seule archive, de façon à pouvoir déployer ces modules sur un serveur d'applications en une seule opération, et de façon cohérente.
- **War** : WAR (Web Application Archive) est un fichier JAR utilisé pour contenir un ensemble de Java Server Pages, servlets, classes Java, fichiers XML, et des pages web statiques (HTML, JavaScript…), le tout constituant une application web. Cette archive est utilisée pour déployer une application web sur un serveur d'applications.
- **JSTL** : JSTL (Java Server page Standard Tag Library). C'est un ensemble de tags personnalisés et développés qui propose des fonctionnalités souvent rencontrées dans les JSP.

III) Rappels sur le protocole HTTP.

Le protocole HTTP (HyperText Transfer Protocol) est le protocole le plus utilisé sur Internet depuis 1990.
Le but ce protocole est de permettre un transfert de fichiers (essentiellement au format HTML) localisés grâce à une chaîne de caractères appelée URL entre un navigateur (le client) et un serveur Web.

La communication entre le navigateur et le serveur se fait en deux temps :



- Le navigateur effectue une **requête HTTP**
- Le serveur traite la requête puis envoie une **réponse HTTP**

Une requête HTTP est traitée à travers plusieurs méthodes :

GET : C'est la méthode la plus courante pour demander une ressource. Une requête GET est sans effet sur la ressource, il est possible de répéter la requête sans effet.

POST : Cette méthode est utilisée pour transmettre des données en vue d'un traitement de ressource (le plus souvent depuis un formulaire HTML). L'URI fourni est l'URI d'une ressource à laquelle s'appliqueront les données envoyées. Le résultat peut être la création de nouvelles ressources ou la modification de ressources existantes.

HEAD : Cette méthode ne demande que des informations sur la ressource, sans demander la ressource elle-même.

OPTIONS : Cette méthode permet d'obtenir les options de communication d'une ressource ou du serveur en général.

CONNECT : Cette méthode permet d'utiliser un proxy comme un tunnel de communication.

TRACE : Cette méthode demande au serveur de retourner ce qu'il a reçu, dans le but de tester et effectuer un diagnostic sur la connexion.

PUT: Cette méthode permet de remplacer ou d'ajouter une ressource sur le serveur. L'URI fourni est celui de la ressource en question.

PATCH: Cette méthode permet, contrairement à PUT, de faire une modification **partielle** d'une ressource.

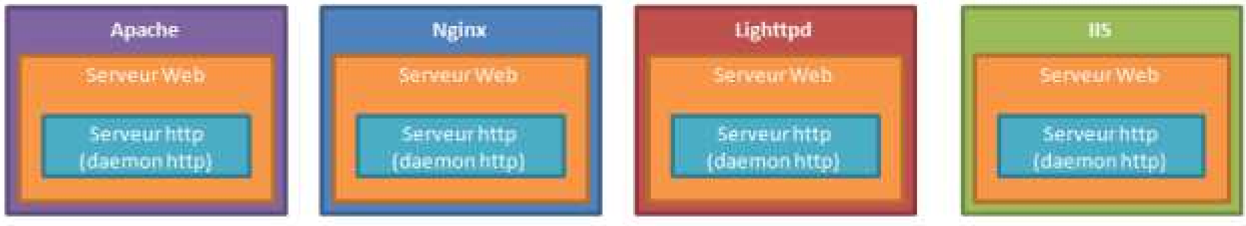
DELETE: Cette méthode permet de supprimer une ressource du serveur.

IV) Architecture Java Web et Java EE.

Nous allons essayer dans cette partie du cours de vous expliquer l'architecture d'une application JEE.

1) Serveur Web http

Pour commencer, il nous faut tout d'abord expliquer un certain nombre de choses :
Qu'est-ce c'est un serveur HTTP ?
Un serveur HTTP, c'est un serveur qui gère exclusivement des requêtes HTTP. Il a pour rôle d'intercepter les requêtes HTTP, sur un port qui est par défaut 80, pour les traiter et générer ensuite des réponses HTTP. Tous les serveurs web embarquent un daemon HTTP (httpd) ou équivalent qui s'occupe de cette fonctionnalité.



Exemple de serveurs web : Apache, Nginx, Lighttpd, IIS…

Quelles sont les fonctionnalités d'un serveur Web ?
A part sa fonction basique qui est d'intercepter et de répondre en Http, un serveur web peut avoir d'autres fonctionnalités telles que :

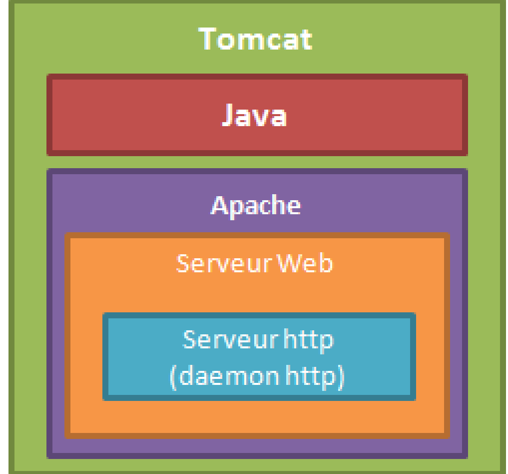
- La gestion de la sécurité, comme les fonctionnalités de restriction des accès par domaine, par utilisateur, par groupe ou par adresse IP,
- La gestion du contenu, comme la redirection des requêtes http, la personnalisation des messages d'erreurs, ou la gestion des timeout ...

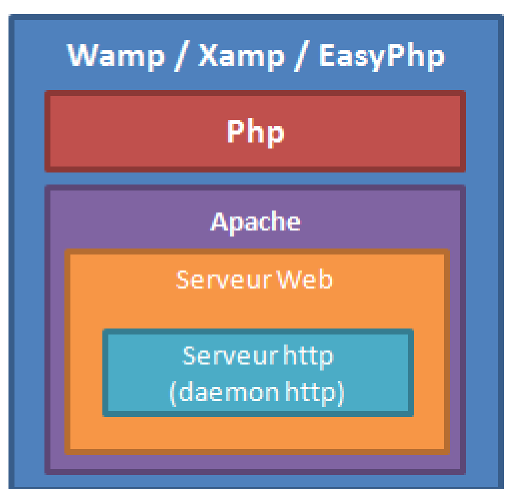
Dans le serveur apache par exemple ces fonctionnalités sont implémentées sous forme de modules (mod_alias, mod_authn_core, mod_proxy…).

2) Conteneur web ou conteneur à Servlets

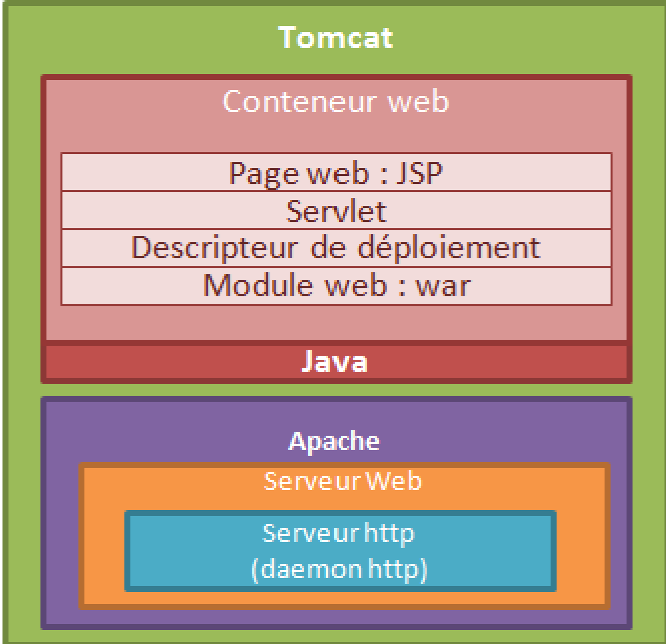
Lorsqu'on étend notre serveur web il devient un conteneur Web. Cette extension va permettre d'avoir la possibilité d'exécuter des programmes écrits avec des langages de programmation (Java, Php, C# ou autres) dans le serveur web.

Par exemple le serveur Tomcat n'est autre qu'un serveur Apache couplé avec un moteur web java et, les serveurs tel-qu'EasyPhp, Wamp ou Xamp ne sont que des serveurs Apaches couplés avec un moteur web Php.





Maintenant, on va se concentrer sur le serveur Tomcat qui est un conteneur web Java (et pas un serveur d'application JEE car il n'a pas de conteneur d'EJB), pour analyser son architecture.



Le conteneur Web Tomcat est composé d'un moteur JSP, un moteur servlet et d'un descripteur de déploiement pour les modules web de type **war**. Ces moteurs sont en réalité des API qui sont implémentés dans le serveur Tomcat, et qui permettent de faire déployer seulement des applications web Java de type **war**.

Les applications Java de type EAR, ne peuvent pas être déployées dans Tomcat, parce que tout simplement le serveur manque les API nécessaires et conformes aux spécifications pour l'implémentation des serveurs d'application Java JEE.

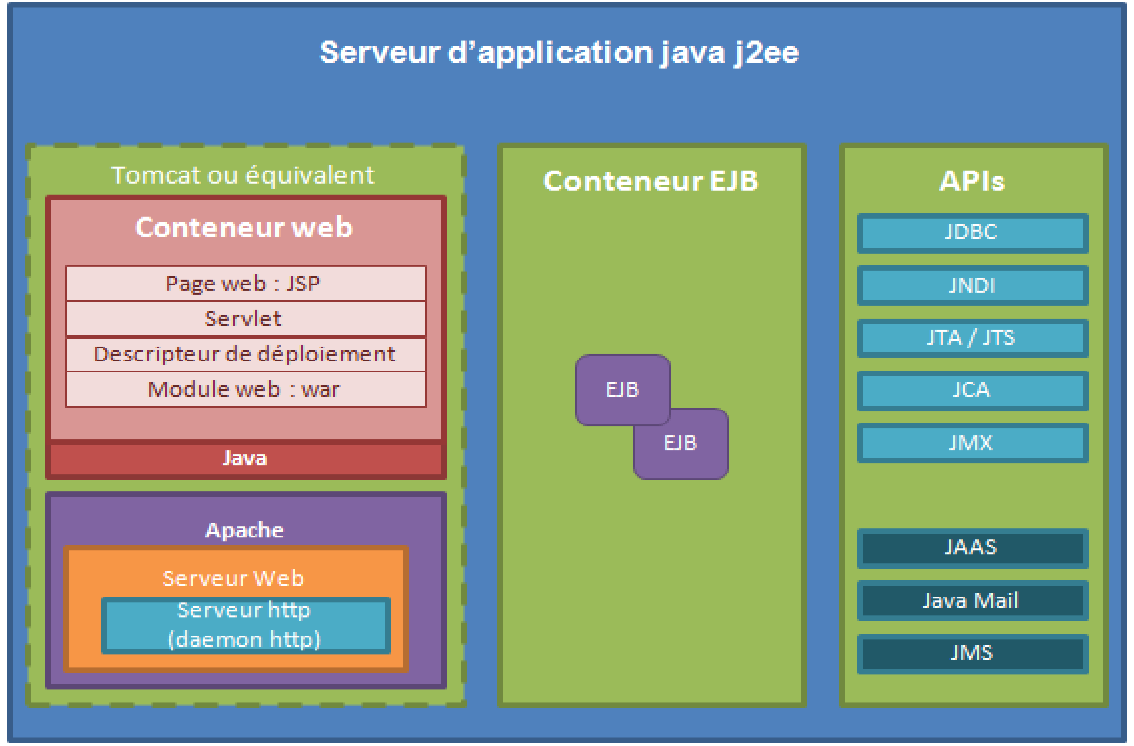
Par exemple, si vous utilisez la bibliothèque JPA dans votre application web et vous le déployez sur un serveur Tomcat, vous serez obligés d'embarquer les jars JPA dans le répertoire lib de l'application, alors que si vous déployez sur un serveur d'application comme JBOSS, vous n'aurez besoin d'aucun **jar** additionnel.

3) Serveur d'application JEE

SI vous avez bien suivi, il faut étendre encore plus le serveur Tomcat pour qu'il devienne un vrai serveur d'application Java JEE.

L'extension nécessaire est composée de deux parties essentielles :

a) **Un conteneur EJB** qui encapsule les traitements des Entreprise JavaBeans.
b) **Un conteneur à Servlet.**
c) Un ensemble de services répartis en :
   A. Des services d'infrastructures
      i. **JDBC** (Java DataBase Connectivity) API d'accès aux bases de données relationnelles.
      ii. **JNDI** (Java Naming and Directory Interface) API d'accès aux services de nommage et aux annuaires d'entreprises.
      iii. **JTA/JTS** (Java Transaction API/Java Transaction Services) API pour la gestion de transactions.
      iv. **JCA** (JEE Connector Architecture) API de connexion au système d'information de l'entreprise comme les ERP.
      v. **JMX** (Java Management Extension) API permettant de développer des applications web de supervision d'applications
   B. Des services de communication:
      i. **JAAS** (Java Authentication and Authorization Service) API de gestion de l'authentification.
      ii. **JavaMail API** pour la gestion de courrier électronique.
      iii. **JMS** (Java Message Service) API de communication asynchrone entre application.



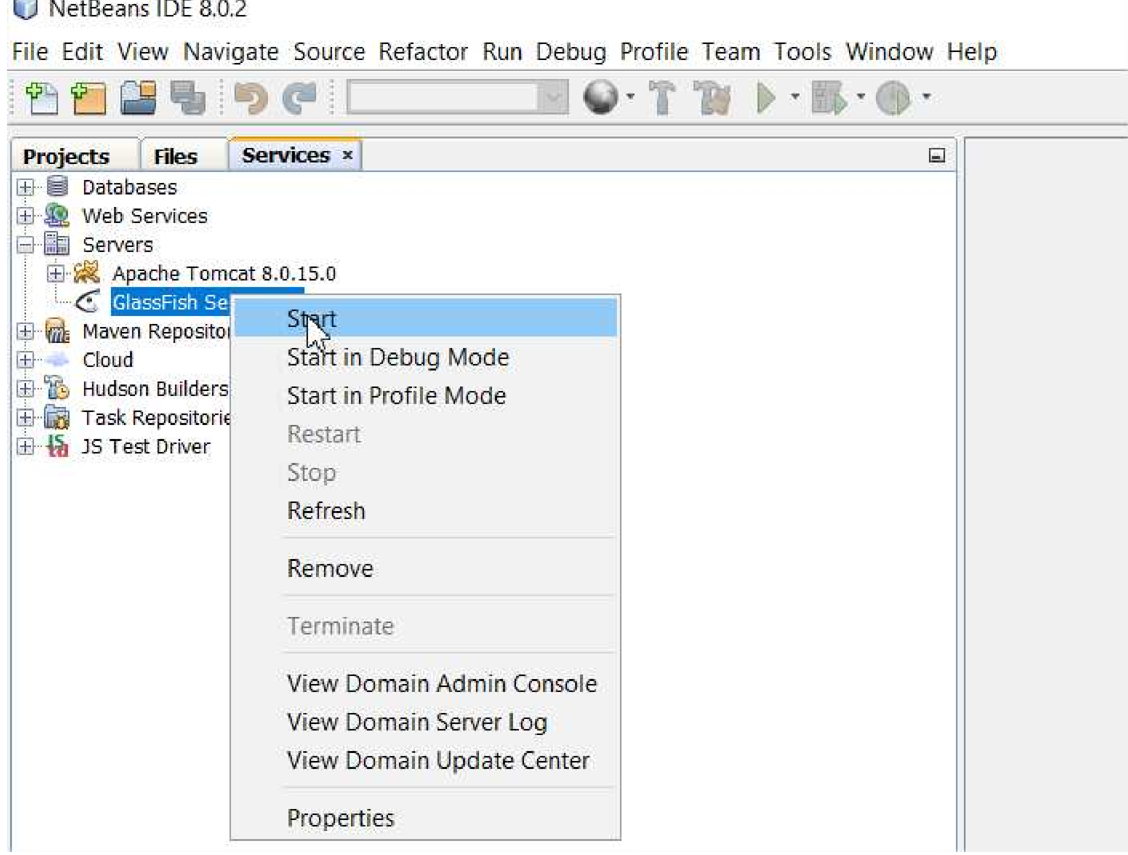
V) Fonctionnement d’un serveur d’application.

Nous allons utiliser le serveur d’application **GlassFish** qui est le serveur d’application par défaut d’Oracle (anciennement Sun). Il contient un conteneur **Web** (conteneur de Servlets) avec un serveur Web **Http**, un conteneur d’**EJB** et un ensemble de services.

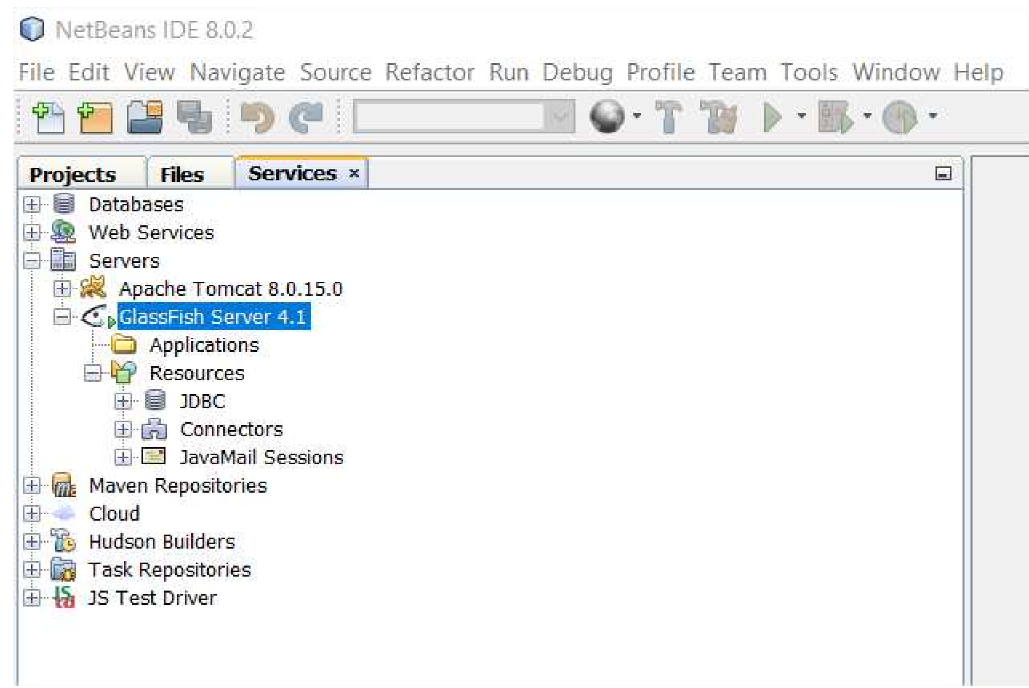
Avec l’IDE NetBeans vous trouverez le serveur d’application **GlassFish** dans l’onglet **Service**.

Pour afficher l’onglet **Services** il est parfois nécessaire de faire **Window → Services**.

Vous pouvez ensuite lancer le serveur GlassFish avec : Clic droit →Start



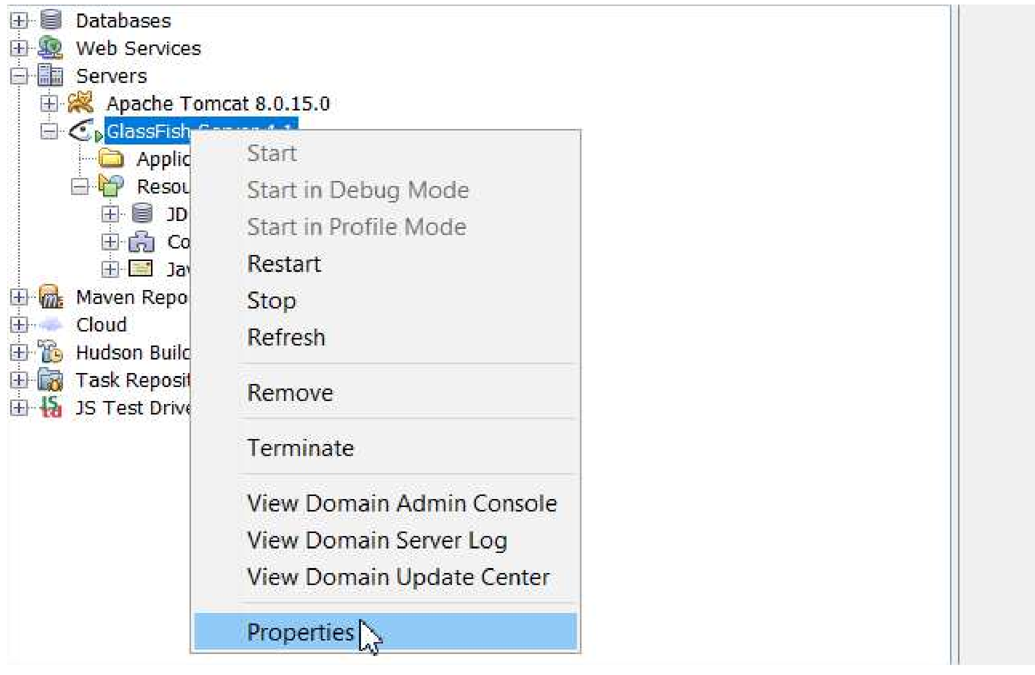
Une fois que vous avez démaré **GlassFish** vous aurez :

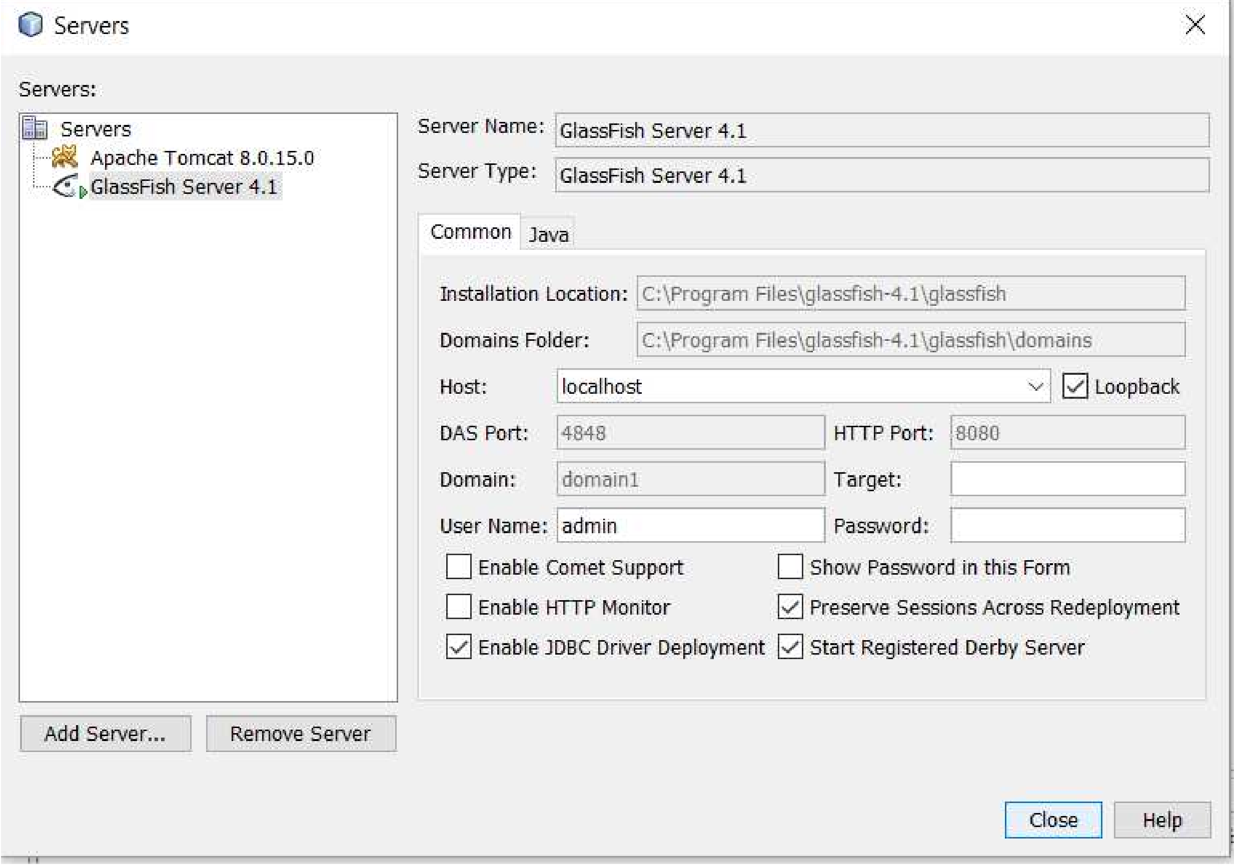


En dépliant le serveur d'application on obtient :

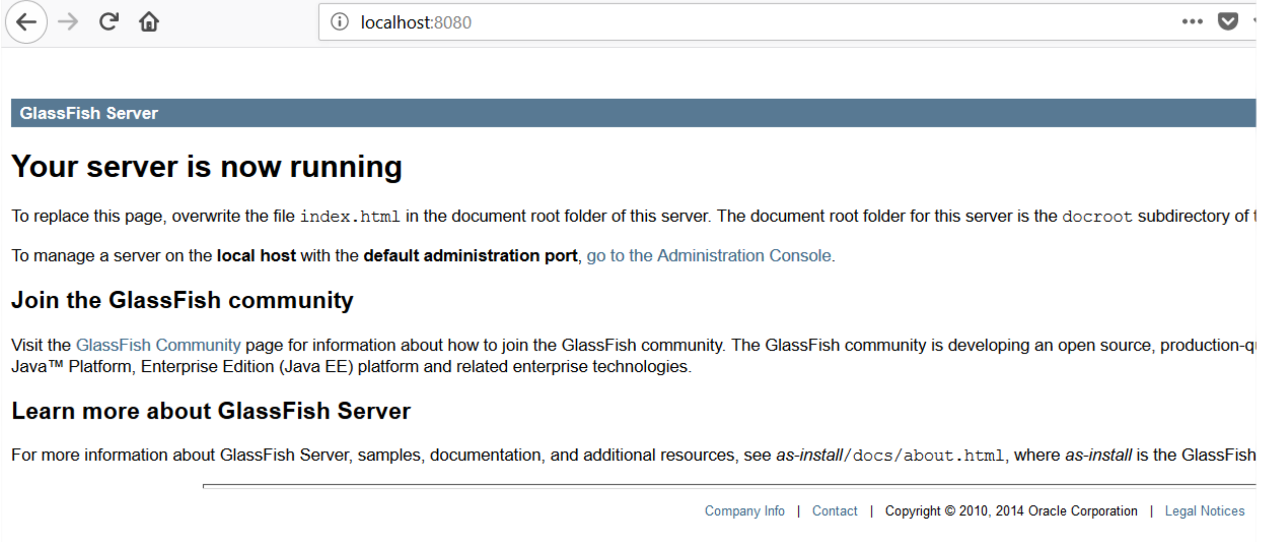
- **Applications** dans lequel vous aurez toutes les applications (application Web et Archive) que vous avez déployé dans votre serveur.
- **Resources** dans lequel vous aurez les ressources de type JDBC, les Connecteurs et les sessions mails de Java.

Pour voir les propriétes du serveur d'application (comme le port d'écoute par exemple) il suffit de faire :

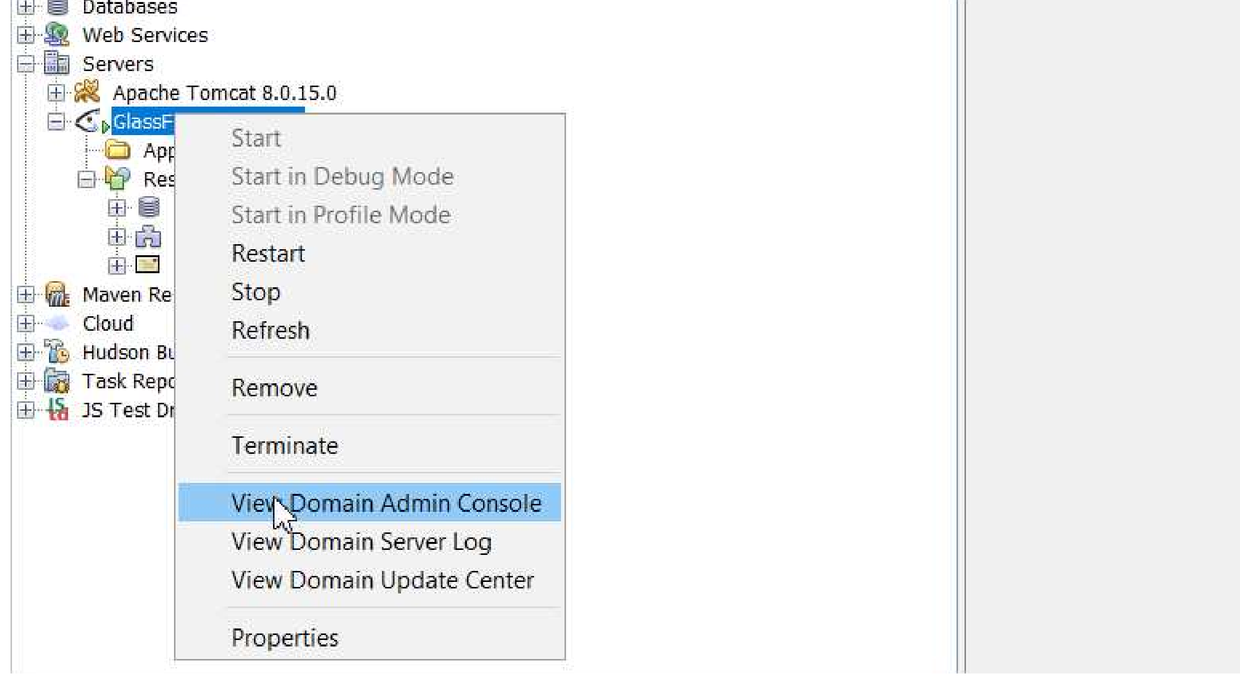




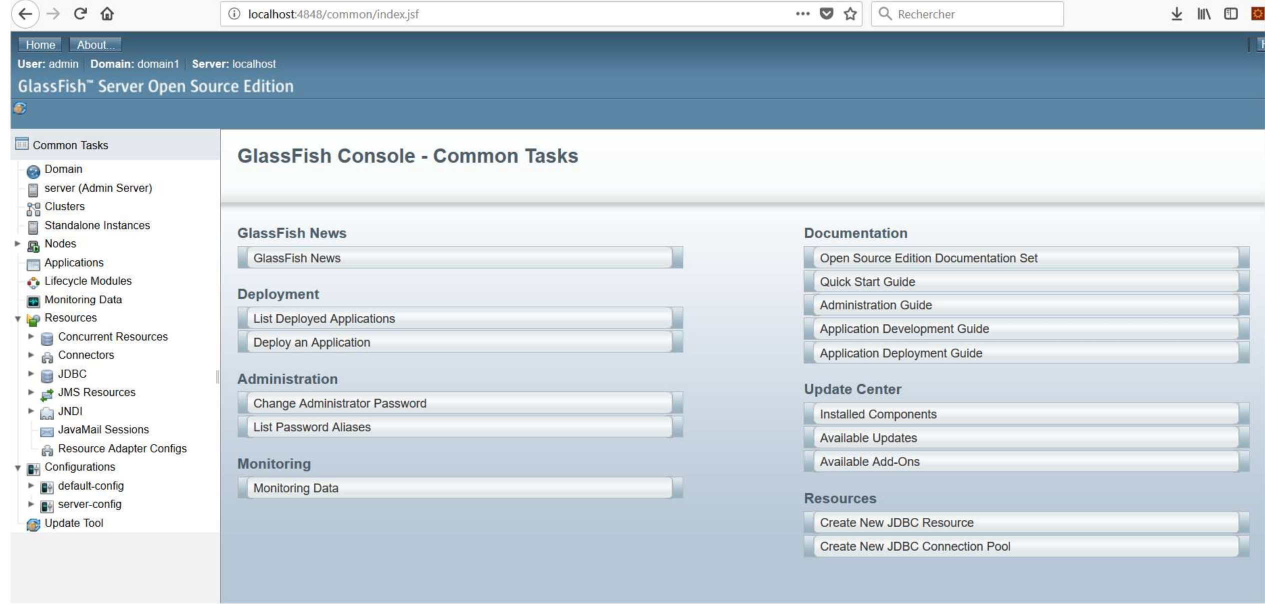
Le serveur **GlassFish** fonctionne souvent sur le port 8080. Pour vérifier qu'il fonctionne et qu'il est bien démaré il suffit d'allez sur http://localhost:8080.



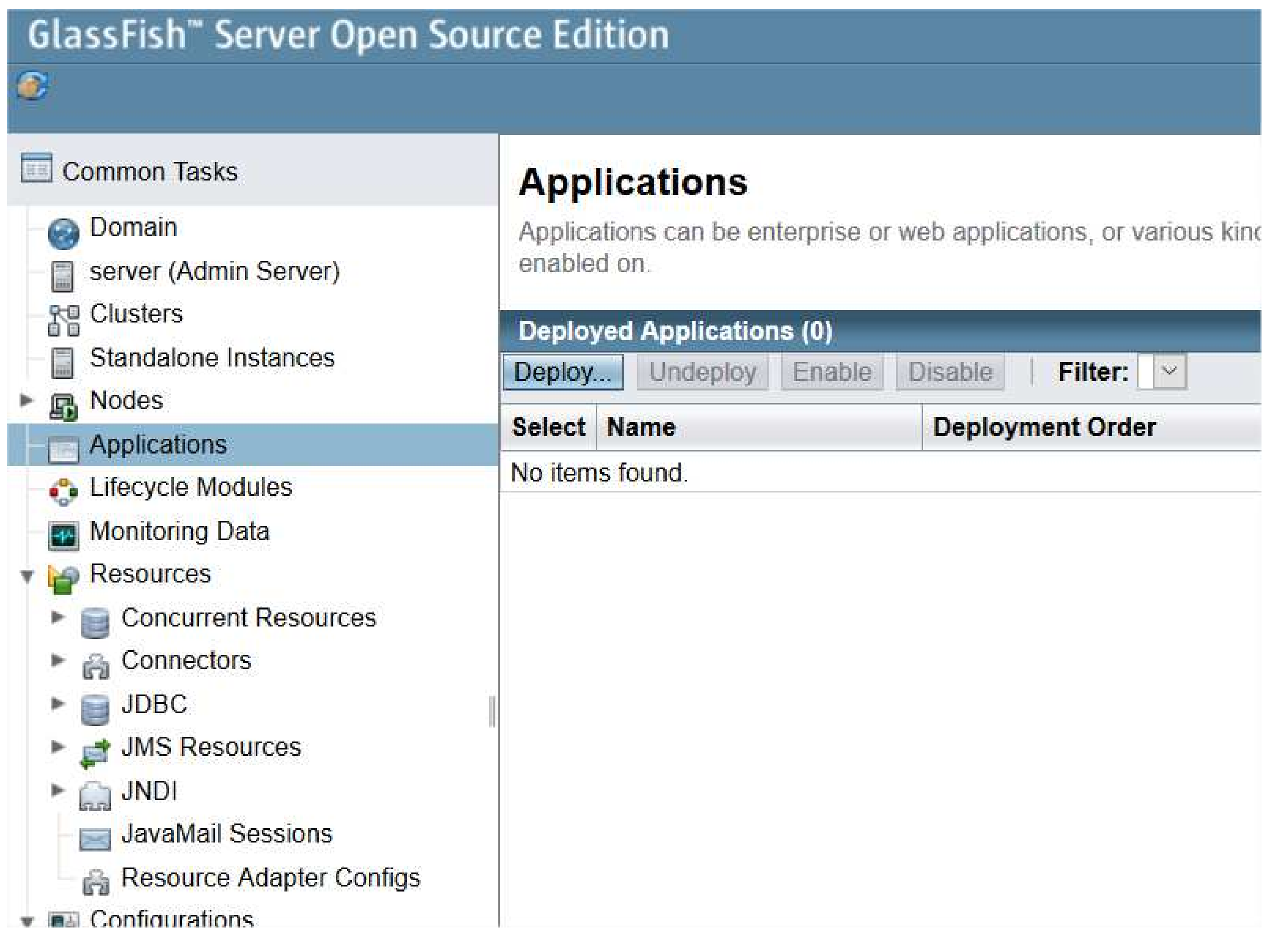
Il est possible de se connecter à la console d'administration du serveur **GlassFish** pour l'administrer.



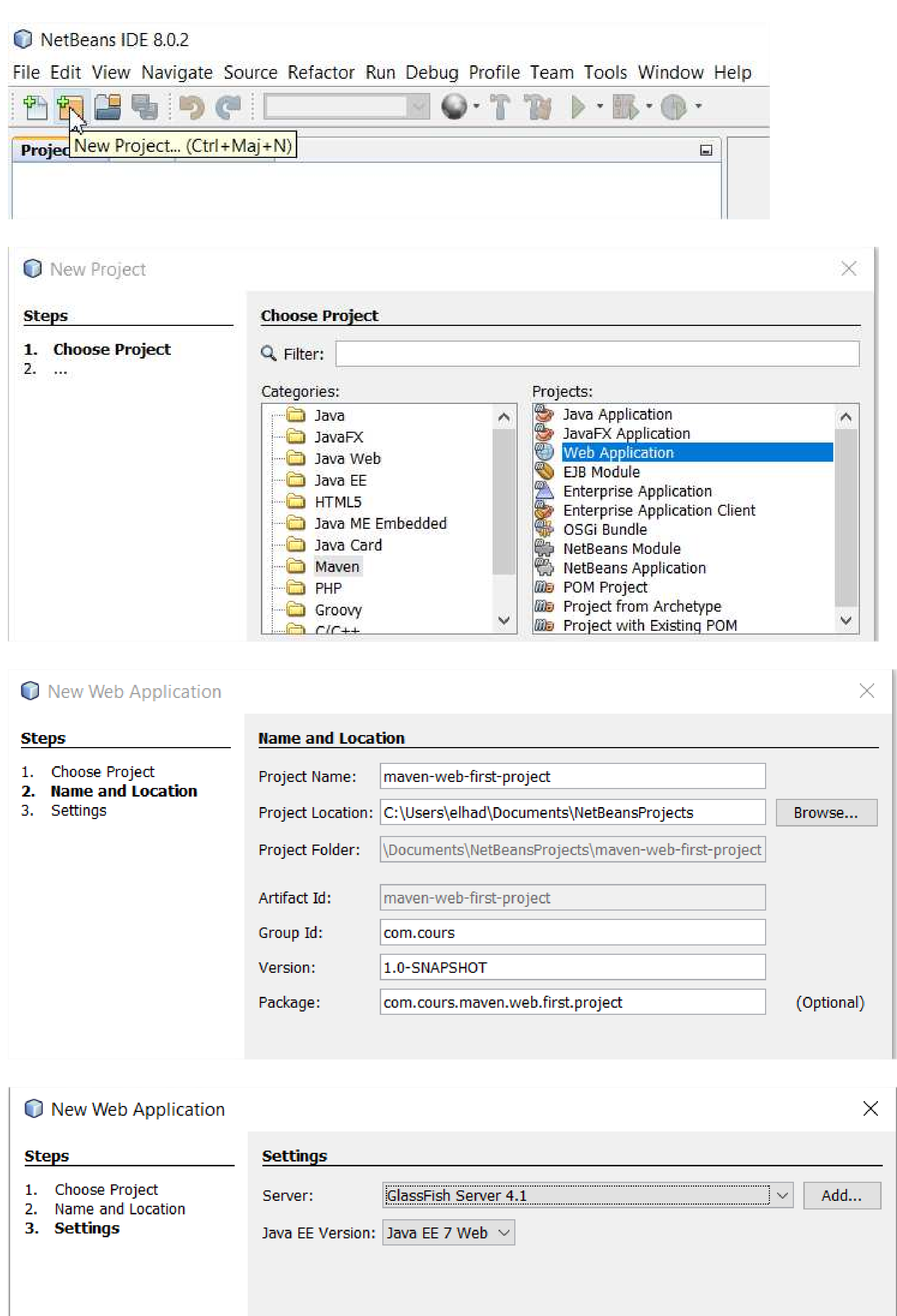
L'action lance sur le navigateur Web l'URL : http://localhost:4848/common/index.jsf



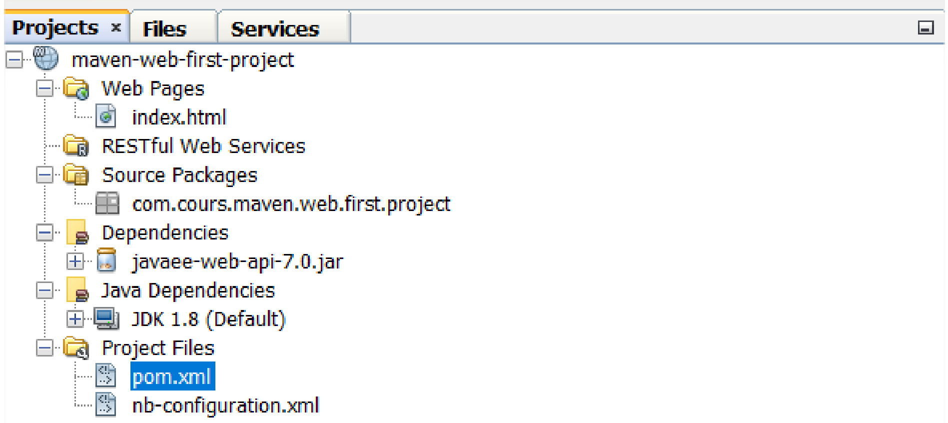
On retrouve les mêmes sections **Applications** et **Resources** de l'IDE NetBeans.



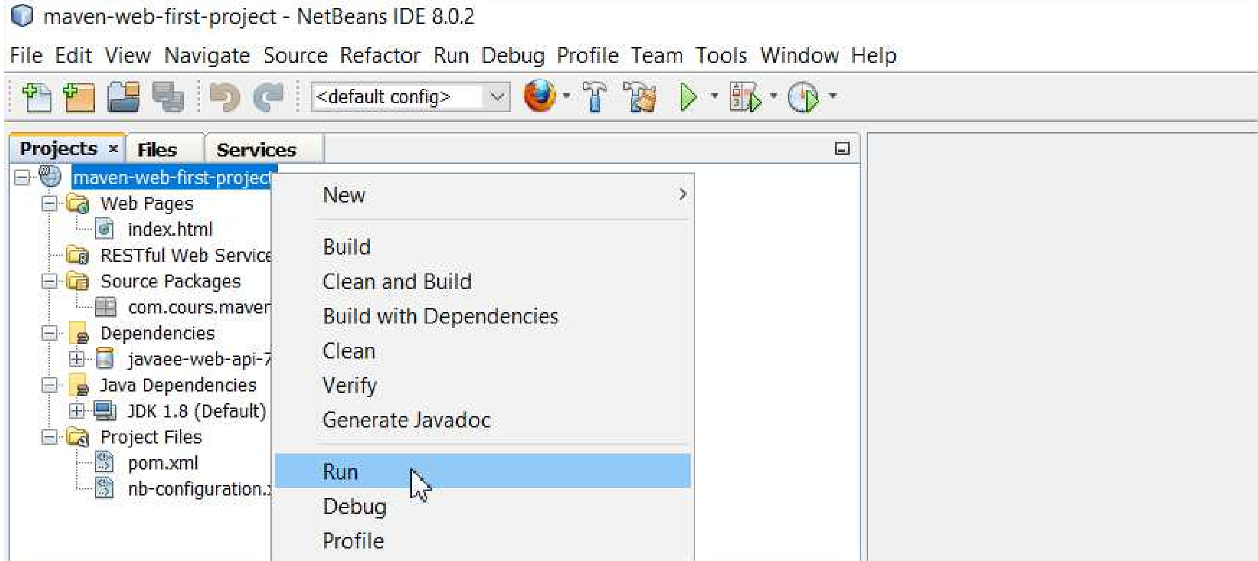
Nous allons maintenant créer notre première application Maven Web qui va s'appeler **maven-web-first-project**.



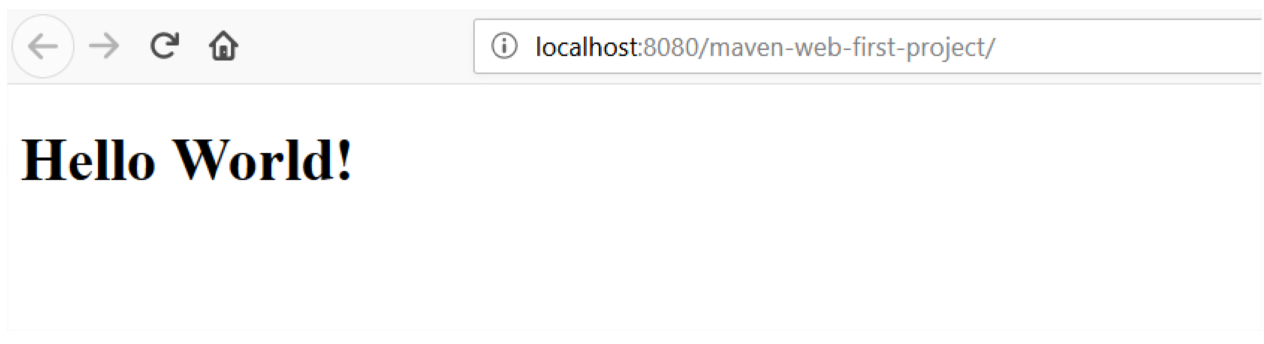
Le projet maven-web-first-project devient :



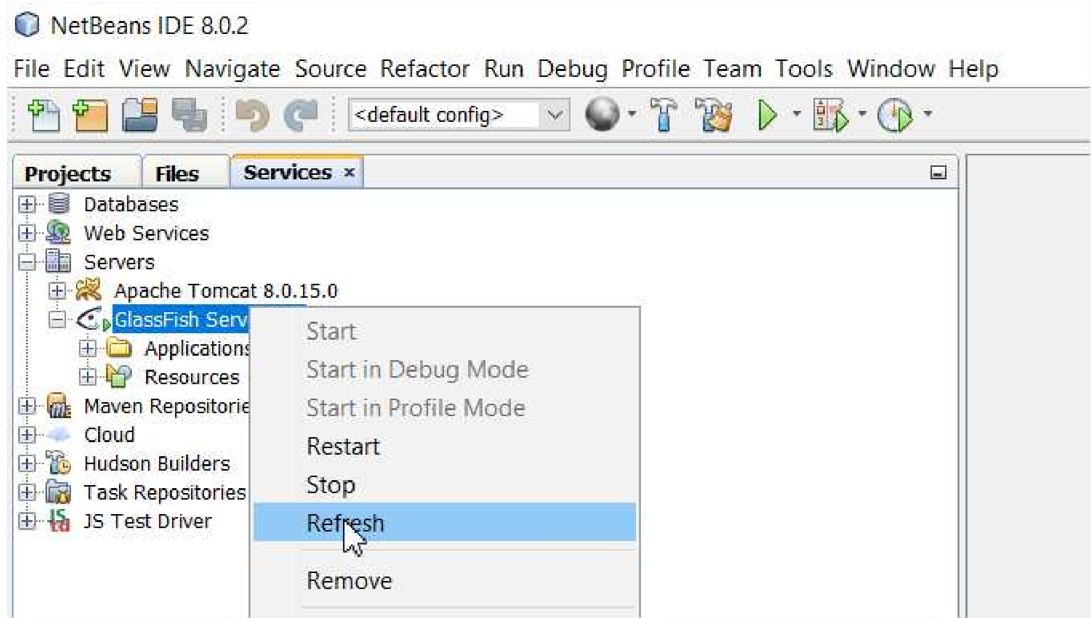
On lance le projet web avec **Run** :



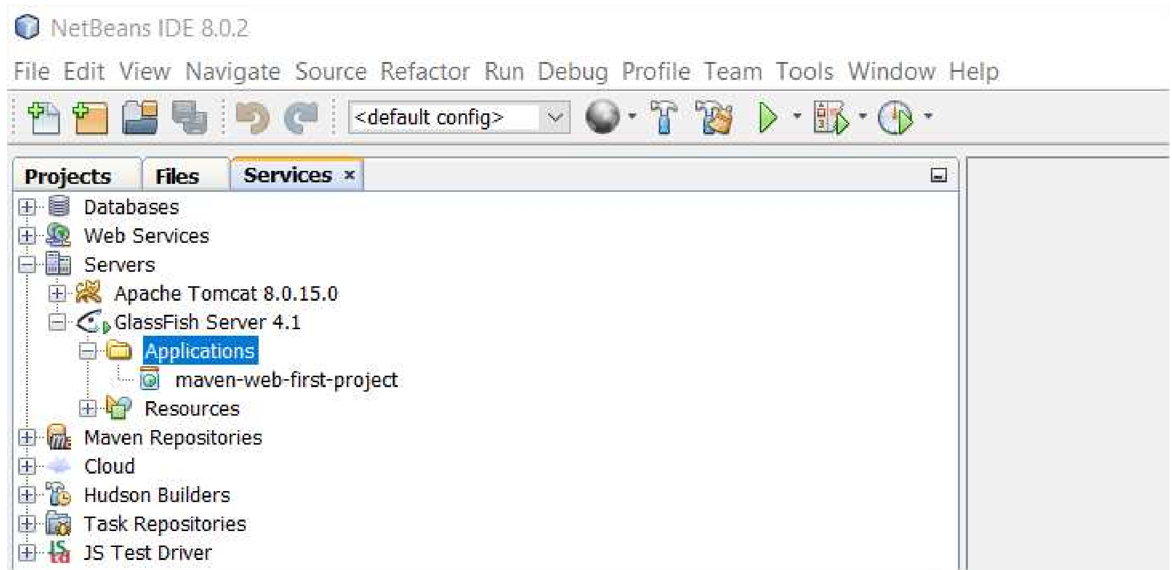
Le navigateur lance l'url http://localhost:8080/maven-web-first-project.



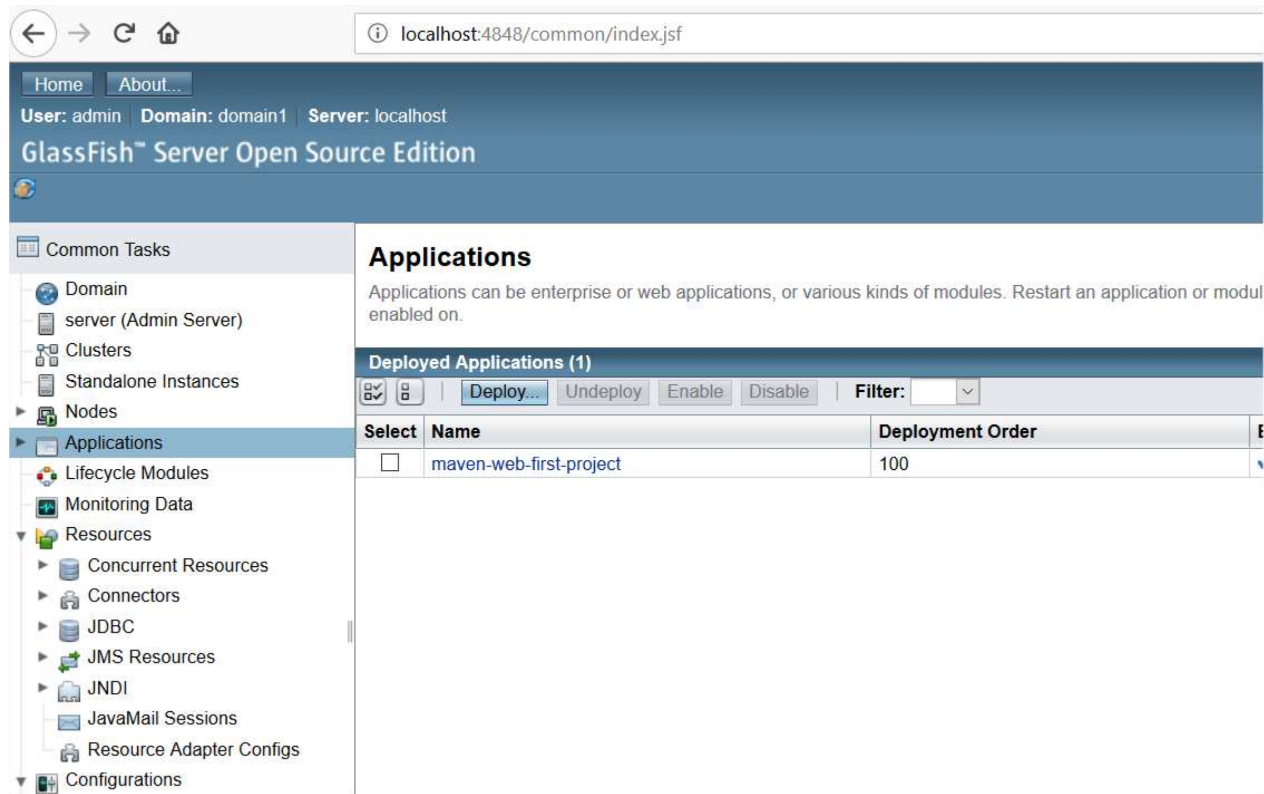
Allons maintenant voir du côté de GlassFish dans NetBeans ce qui se passe. Rafraichissons le serveur d'application pour voir les données actualisées.



En dépliant **Applications** on obtient :

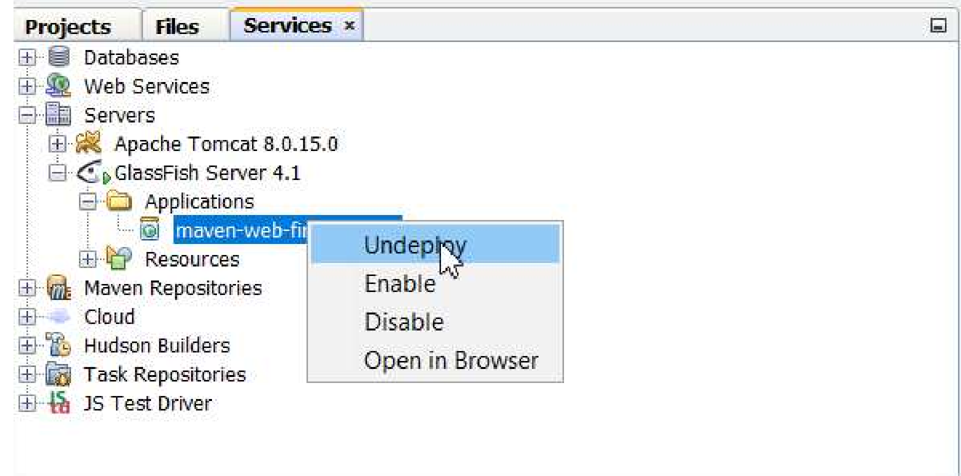


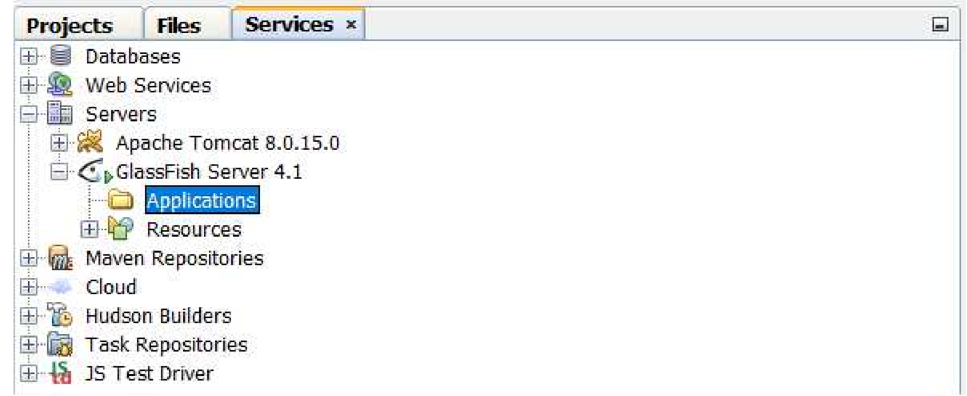
On retrouve la même chose dans la console d'administration de GlassFish sur http://localhost:4848/common/index.jsf.



GlassFish a bien deployé l'application **maven-web-first-project** au sein de ses applications courantes.
Il est possible d'enlever du serveur **GlassFish** l'application Web **maven-web-first-project, p**our cela il suffit de faire un **UnDeploy.**

Attention il est important d'effectuer l'operation du **UnDeploy** de votre projet Web si vous voulez faire un **Clean And Build** de votre projet Web pendant qu'il est déployé dans votre serveur d'application sinon vous allez avoir une erreur comme quoi l'IDE ne peut pas faire un **Clean And Build** de votre projet.



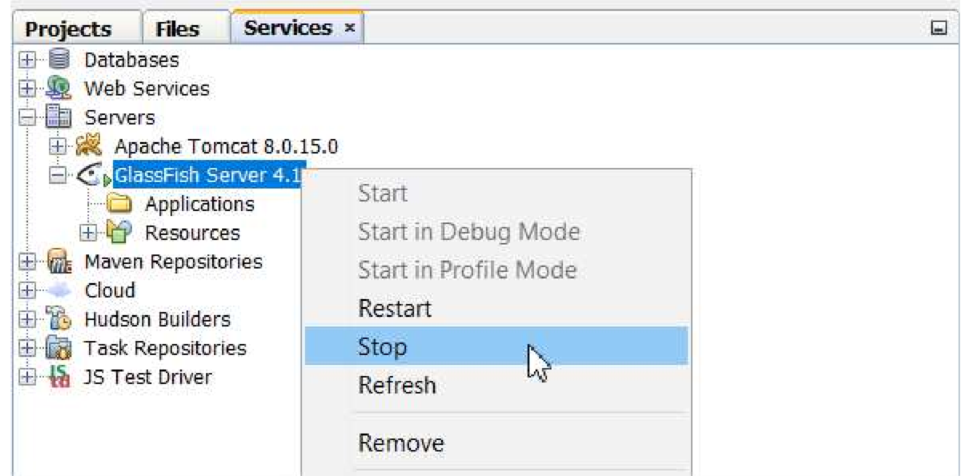


Si on tente d'accéder à l'application **maven-web-first-project** à travers l'Url http://localhost:8080/maven-web-first-project

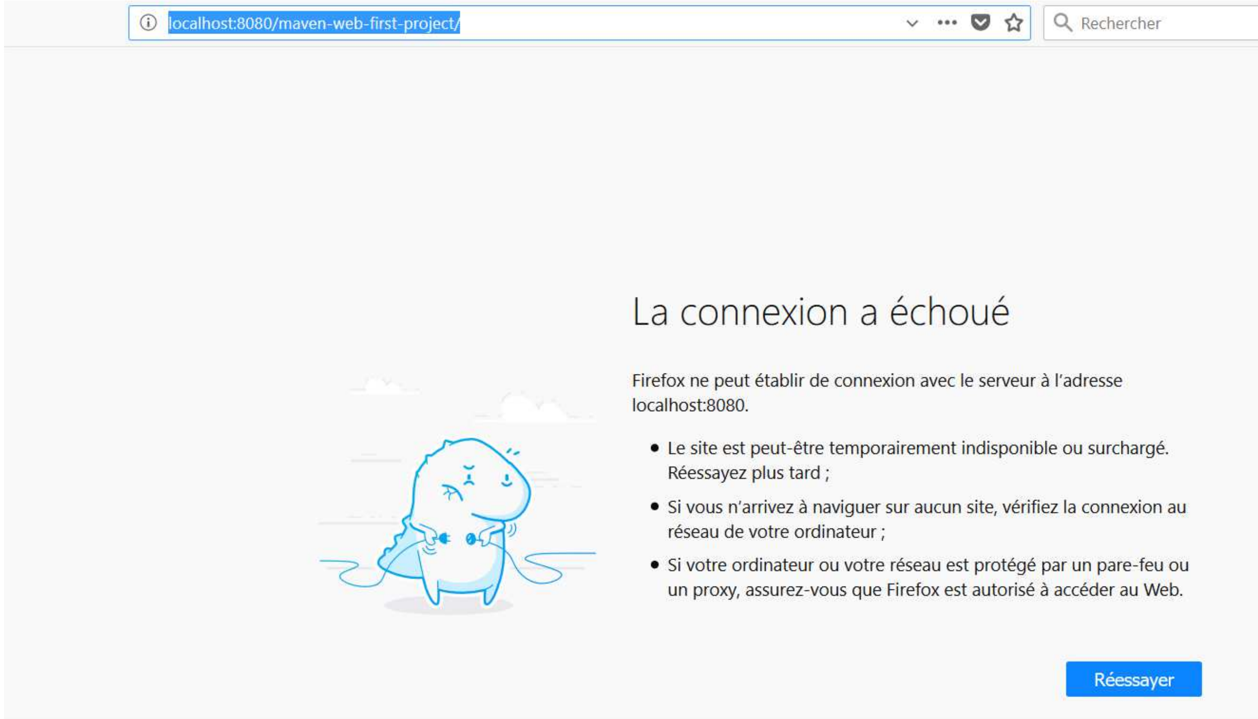
Donc il faut y penser si vous avez une erreur **404** cela veut dire que votre application Web ne s'est pas correctement déployée (en générale parce qu'il y a une erreur dans le code).

Il est important de préciser qu'en Java Standard on peut se permettre certains écarts mais en Java Web il faut être extrêmement rigoureux et comprendre ce que vous faites étape par étape car la moindre erreur dans le code vous bloqur et cela peut vous faire perdre énormément de temps. Donc n'oubliez pas une erreur en rouge = pas de compilation = pas de déploiement dans le serveur d'application donc des 404.

On peut aussi éteindre le serveur d'application :



Si on tente d'accéder à l'application **maven-web-first-project** à travers l'Url
http://localhost:8080/maven-web-first-project



Il est clair que le serveur d'application est maintenant éteint.
Maintenant vous savez tout ce qu'il y a à savoir sur les serveurs d'applications même si vous n'utilisez pas **GlassFish** le principe reste le même.

VI) Introduction aux Servlets.

Une servlet est une classe Java qui permet de créer dynamiquement des données au sein d'un serveur HTTP. Ces données sont le plus généralement présentées au format HTML, mais elles peuvent également l'être au format XML ou tout autre format destiné aux navigateurs web. Les servlets utilisent l'API Java Servlet (package **javax.servlet**). Une servlet s'exécute dynamiquement sur le serveur web et permet l'extension des fonctions de ce dernier, typiquement : accès à des bases de données, transactions d'e-commerce, etc. Une servlet peut être chargée automatiquement lors du démarrage du serveur web ou lors de la première requête du client ; une fois chargées les servlets restent actives dans l'attente d'autres requêtes du client.

En quelques mots, une servlet est juste une classe qui implémente l'interface javax.servlet.Servlet.
En pratique nous ferons dériver notre classe Servlet de la classe javax.servlet.http.HttpServlet qui lui hérite de javax.servlet.GenericServlet qui lui enfin implémente l'interface javax.servlet.Servlet.
Ce qui implique bien sûr que toute classe qui hérite de javax.servlet.http.HttpServlet implémente également l'interface javax.servlet.Servlet et est donc par conséquence belle et bien une Servlet.

La classe javax.servlet.http.HttpServlet possède les méthodes **doGet**, **doPost**, **doHead**, **doOptions**, **doPut**, **doDelete**, **doTrace** etc… . Bien évidement c'est les mêmes méthodes **GET**, **POST**, **HEAD**, **OPTIONS**, **PUT**, **DELETE** et **TRACE** du protocole HTTP standard.
Faisons un petit focus sur les méthodes HTTP de javax.servlet.http.HttpServlet. En effet les méthodes **doGet**, **doPost**, **doHead**, **doOptions**, **doPut**, **doDelete** et **doTrace** ont chacune en paramètre javax.servlet.http.HttpServletRequest request et javax.servlet.http.HttpServletResponse response.

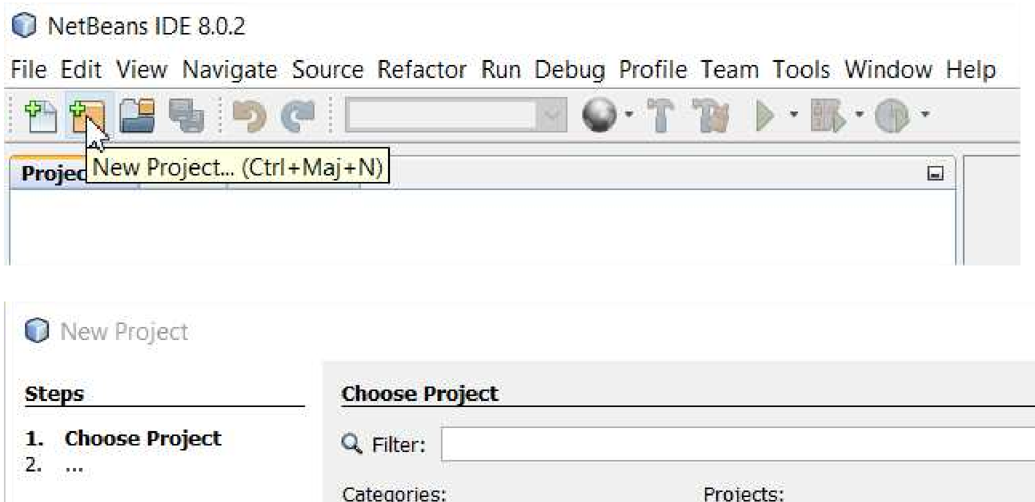
HttpServletRequest est un objet contient la requête HTTP. Il donne accès à certaines informations telles que les en-têtes (headers) et le corps de la requête.

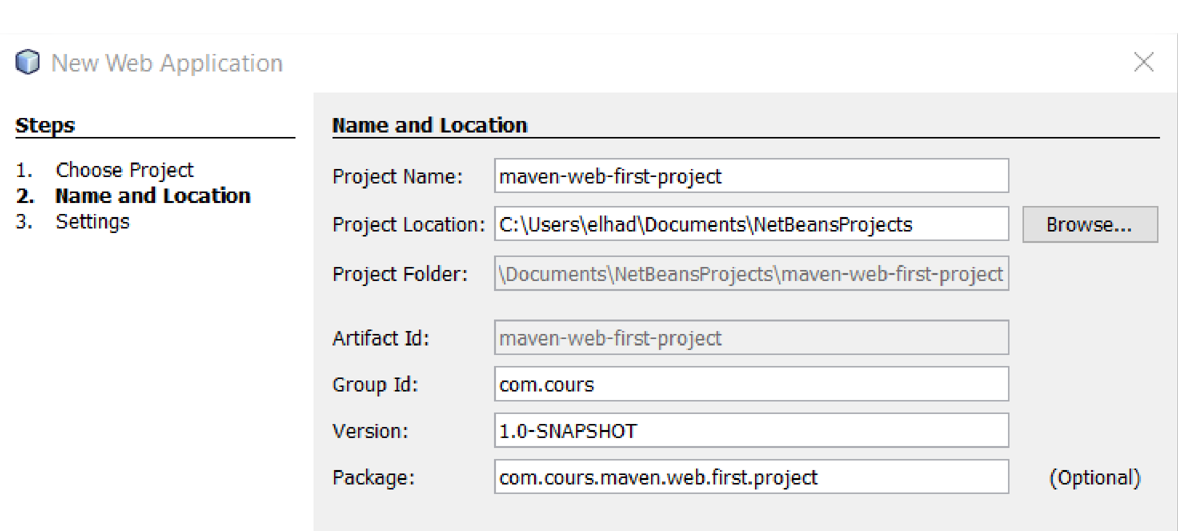
HttpServletResponse est un objet de la réponse HTTP pour le client, il peut être personnalisé.
Exemple : on peut rajouter des informations dans les en-têtes et corps de la réponse.

Toutes les Servlet sont exécutées à travers un **Conteneur à Servlets**.
Un conteneur à **Servlet** est un moteur de servlets qui prend en charge et gère les servlets : chargement de la servlet, gestion de son cycle de vie, passage des requêtes et des réponses ...
Un conteneur à **Servlet** peut être intégré dans un serveur d'applications qui va contenir d'autres conteneurs (conteneur d'EJB : Entreprise Java Bean) et éventuellement proposer d'autres services.

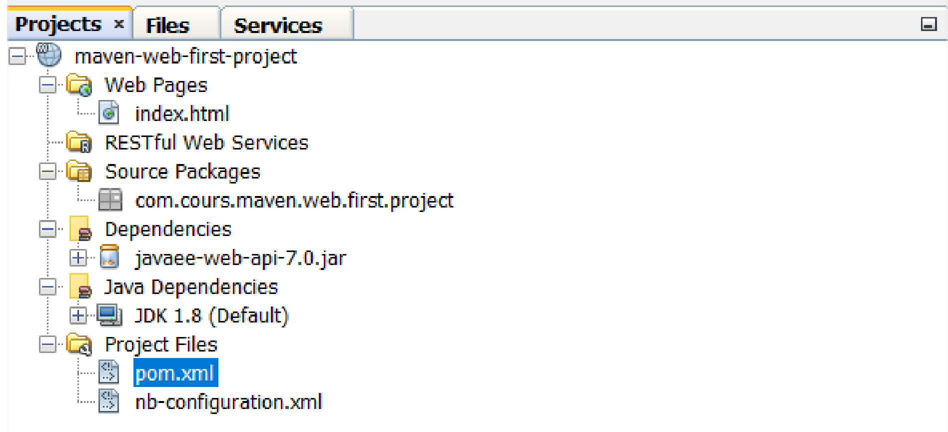
VII) Notre première Servlet.

Nous allons continuer avec le projet **maven-web-first-project** pour la création de notre première Servlet.
Si vous n'avez pas encore crée le projet Web **maven-web-first-project**, vous pouvez le créer avec les instructions ci-dessous :

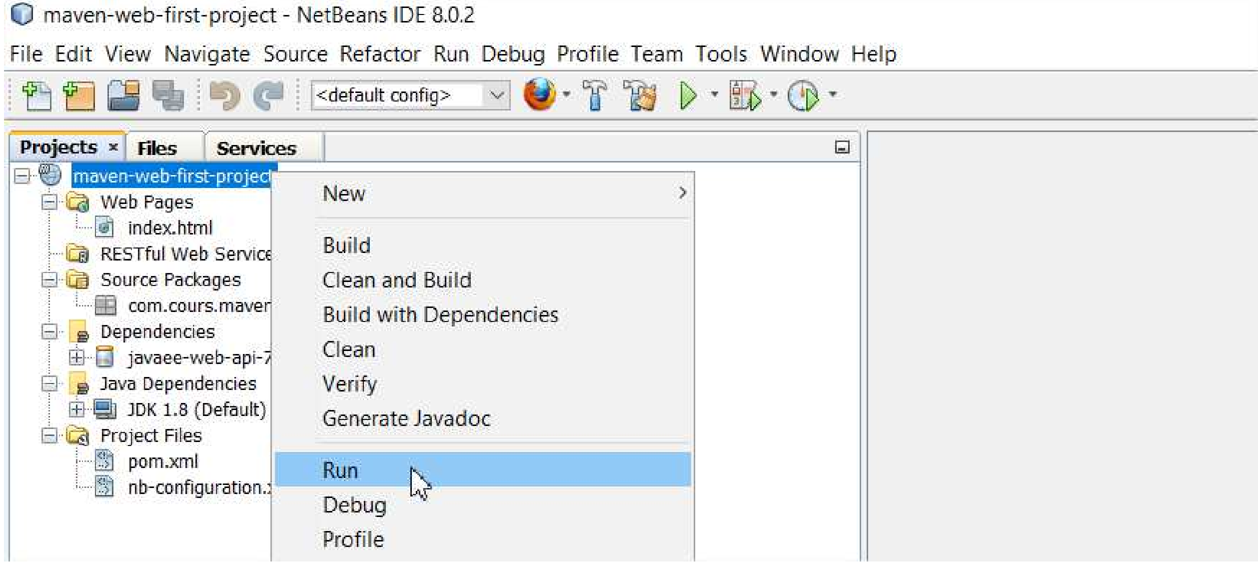




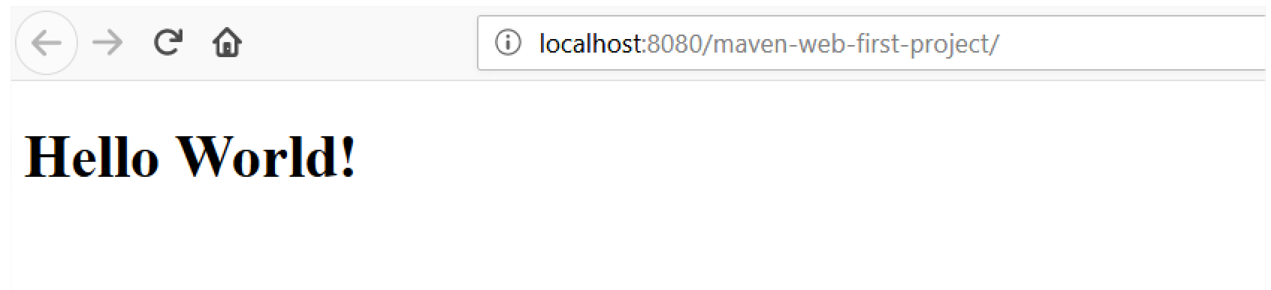
Le projet **maven-web-first-project** devient :



On lance le projet web avec le **Run** :



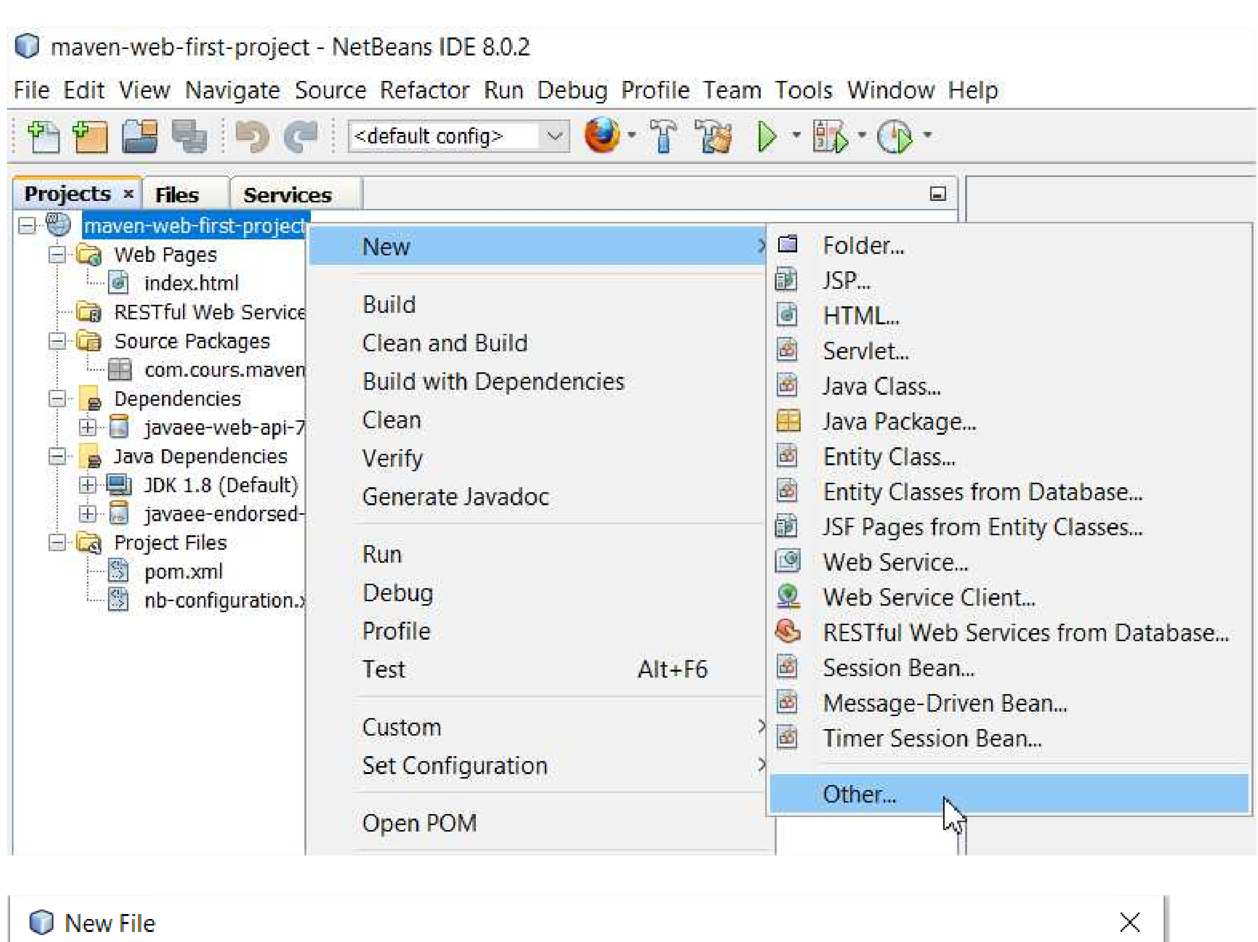
Le navigateur lance l'url http://localhost:8080/maven-web-first-project.

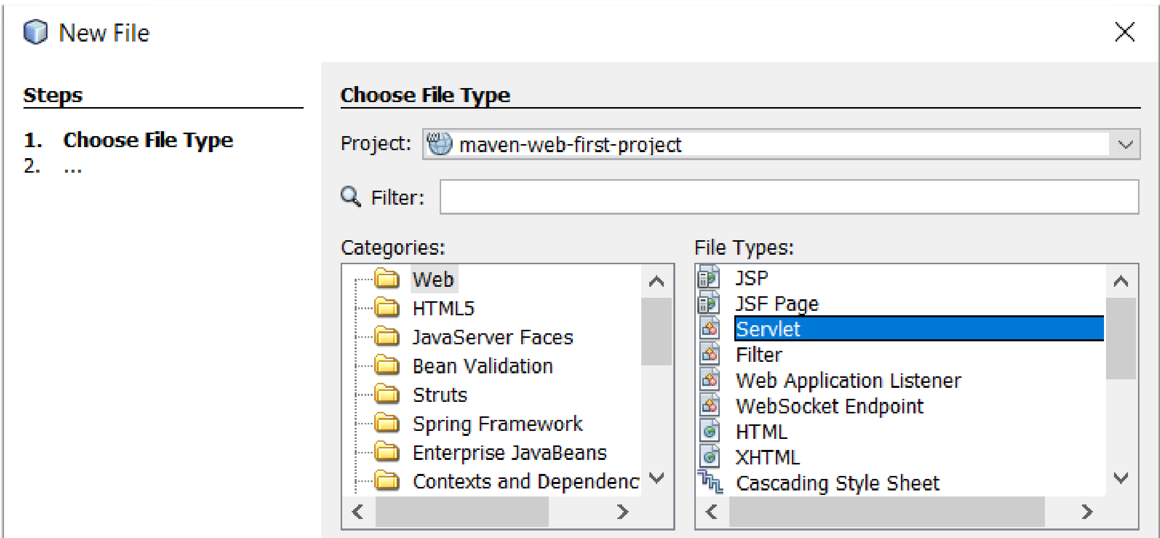


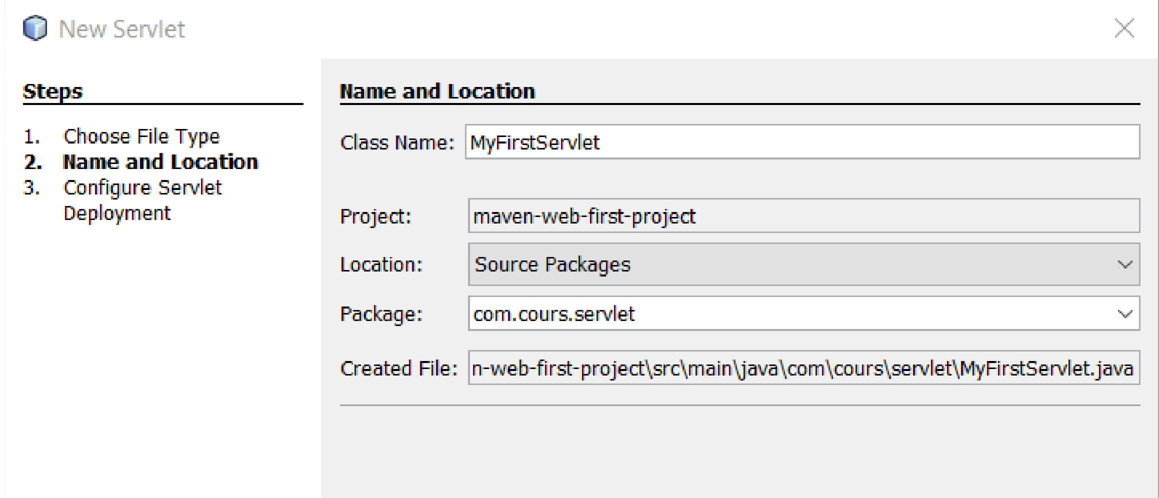
Tout d'abord vérifier que votre server **GlassFish** est installé et démarré alors allons dans l'onglet **Services** de NetBeans.
Pour afficher l'onglet **Services** il est parfois nécessaire de faire **Window → Services**.

Clic droit sur GlassFish → Start si le serveur n'est pas démarré et si tel n'est pas le cas le faire.

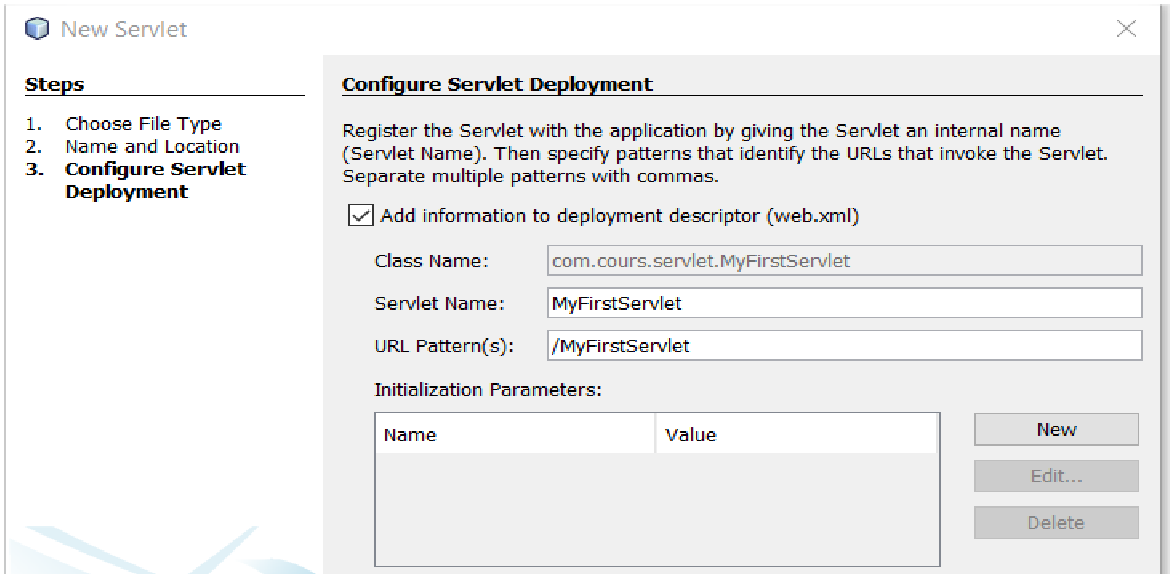
Créons notre première servlet **com.cours.servlet.MyFirstServlet** avec **New → Servlet** ou bien **New → Other→Web→Servlet.**



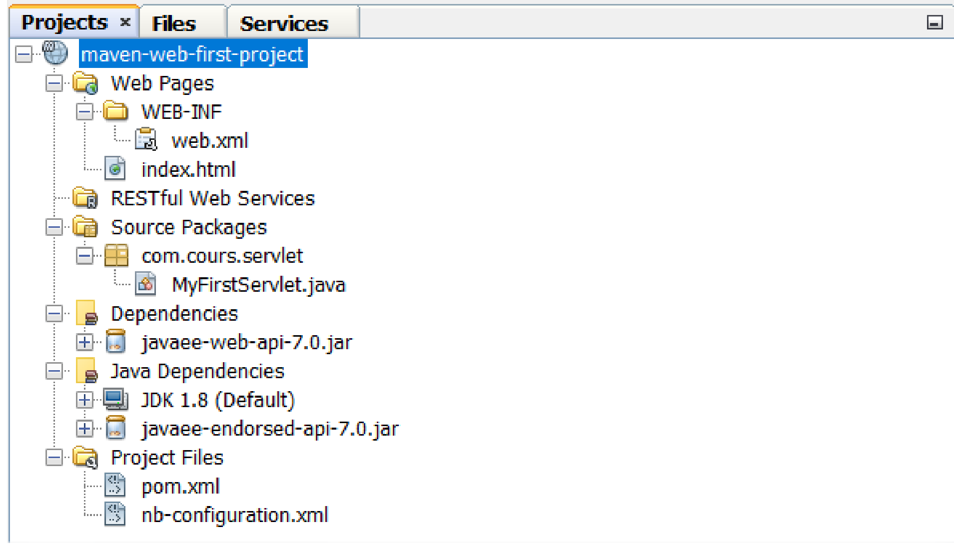




Cocher la case (Add information to descriptor (web.xml))→ Finish.



Le projet maven-web-first-project devient :



La classe **MyFirstServlet** devient :

```java
package com.cours.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * @author elhad
 */
public class MyFirstServlet extends HttpServlet {

    /**
     * Processes requests for both HTTP <code>GET</code> and <code>POST</code>
     * methods.
     * @param request servlet request
     * @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, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet MyFirstServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet MyFirstServlet at " + request.getContextPath() + "</h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }

    // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click on the + sign on the left to edit the code.">
    /**
     * Handles the HTTP <code>GET</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
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * 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
     */
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Returns a short description of the servlet.
     *
     * @return a String containing servlet description
     */
    @Override
    public String getServletInfo() {
        return "Short description";
    }// </editor-fold>

}
```

En version copiable :

```
package com.cours.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author elhad
 */
public class MyFirstServlet extends HttpServlet {

    /**
     * Processes requests for both HTTP <code>GET</code> and <code>POST</code>
     * methods.
     *
     * @param request servlet request
     * @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, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet MyFirstServlet</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet MyFirstServlet at " + request.getContextPath() + "</h1>");
            out.println("</body>");
            out.println("</html>");
        }
    }

    // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click on the + sign on the left to edit the
code.">
    /**
     * Handles the HTTP <code>GET</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
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * 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
     */
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        processRequest(request, response);
    }

    /**
     * Returns a short description of the servlet.
     *
     * @return a String containing servlet description
     */
    @Override
    public String getServletInfo() {
        return "Short description";
    }// </editor-fold>

}
```

On retrouve les fameuses méthodes :

- **doGet** (qui permet de traiter les données reçues par **GET** c'est-à-dire avec l'URL), en effet le chargement de l'URL http://localhost:8080/maven-web-first-project/MyFirstServlet?monParam1=value1&monParam2=value2&monParam3=value3 va automatiquement appeler la méthode **doGet** de la servlet MyFirstServlet. On pourra récupérer les paramètres monParam1, monParam2 et monParam3 par l'intermédiaire de **request** de type **HttpServletRequest** (request.getParameter("monParam1"), request.getParameter("monParam2") et request.getParameter("monParam3")).

- **doPost** (qui permet de traiter les données reçu par **POST** c'est-à-dire avec des formulaires).

Exemple : considérons le formulaire ci-dessous.

```
<form action="monNomAppliWeb/MyFirstServlet " method="post" id="my-first-servlet" name="my-first-servlet">
  <input type="text" name="monParam1" value="value1" id="monParam1" class="ma-class-1" title="Mon Paramêtre numero 1">
  <input type="text" name="monParam2" value="value2" id="monParam2" class="ma-class-2" title="Mon Paramêtre numero 2">
  <input type="text" name="monParam3" value="value3" id="monParam3" class="ma-class-3" title="Mon Paramêtre numero 3">
  <button class="btn" type="submit">Valider</button>
</form>
```

Au clic sur le bouton **Valider** l'application va automatiquement appeler la méthode **doPost** de la servlet MyFirstServlet. On pourra ainsi récupérer les paramètres monParam1, monParam2 et monParam3 par l'intermédiaire de **request** de type **HttpServletRequest** (request.getParameter("monParam1"), request.getParameter("monParam2") et request.getParameter("monParam3")).

- **init** (lancé lors de l'initialisation de la Servlet) qui est une méthode assez importante permet d'initialiser tous les objets nécessaires dans notre Servlet. Cette méthode n'a pas été définie dans notre servlet **MyFirstServlet** mais on aurait pu la rajouter.

Il existe plein d'autres méthodes dans une classe de type Servlet mais les trois les plus importantes sont **doGet**, **doPost** et **init**. Pour les autres méthodes je vous invite à consulter la Java Doc de la classe **javax.servlet.http.HttpServlet**.
Il est important de préciser que **request** de type **HttpServletRequest** gère aussi la session utilisateur.

```
// Mettre une information dans la session
request.getSession().setAttribute("user","monUserEnSession");

// Recuperer l'information dans la session
request.getSession().getAttribute("user");
```

On retrouve enfin la méthode **processRequest** qui est auto-générée par NetBeans avec du code html. On supprimera cette méthode plus tard là on la garde juste pour vérifier que notre nouveau Servlet MyFirstServlet.
Ligne 26 : **HttpServletRequest** contient la requête HTTP. Il donne accès à certaines informations telles que les en-têtes (headers) et le corps de la requête.
HttpServletResponse contient la réponse HTTP pour le client, il peut être personnalisé. Il permet par l'intermédiaire de sa méthode **getWriter()** (de type java.io.PrintWriter) d'envoyer au navigateur web du code html.
Il est aussi possible d'ajouter des informations dans les en-têtes et corps de la réponse par l'intermédiaire de **HttpServletResponse**.

Le fichier **web.xml** (contenu dans maven-web-first-project/src/main/webapp/WEB-INF) devient :

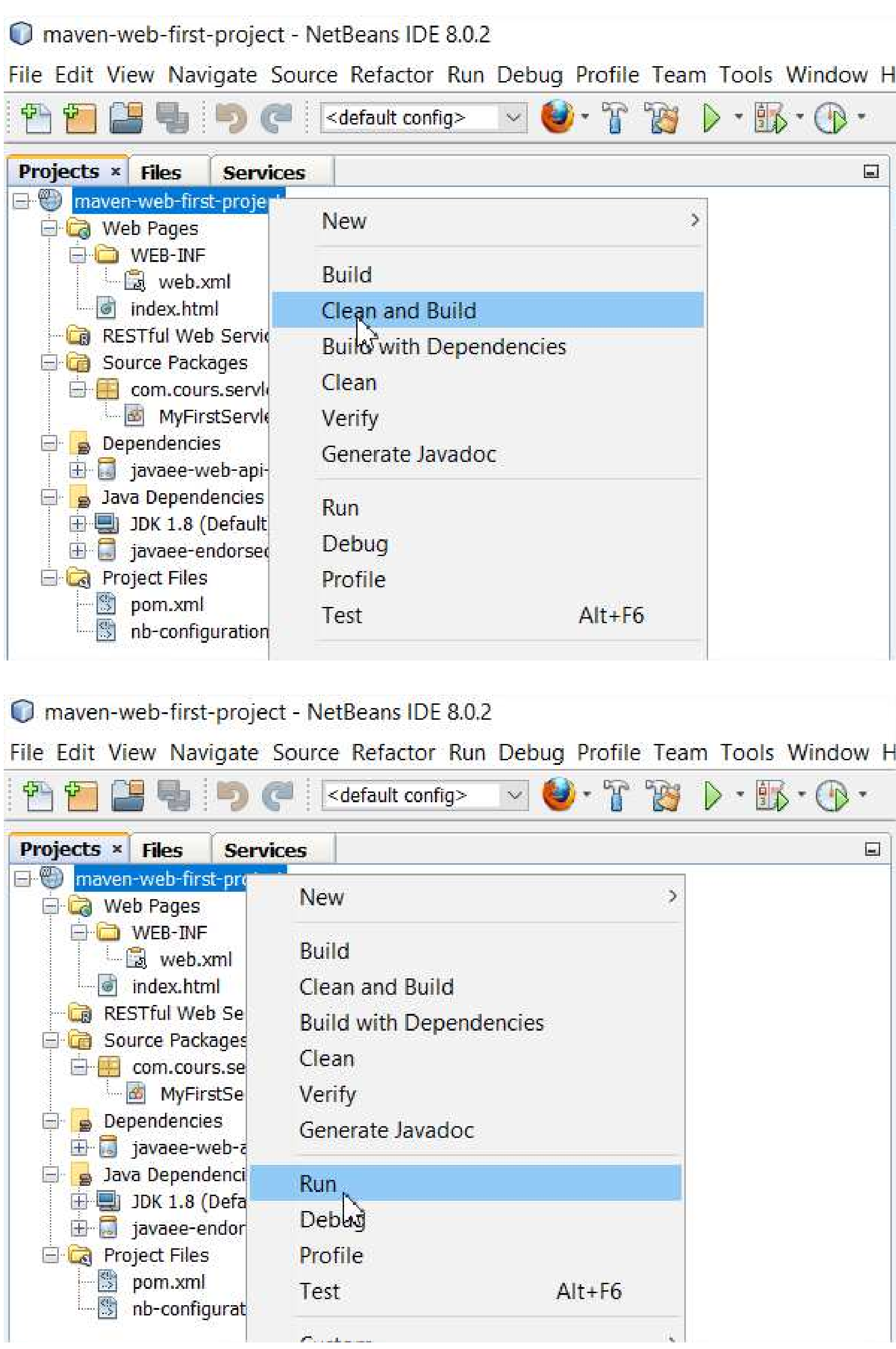
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      <servlet>
4          <servlet-name>MyFirstServlet</servlet-name>
5          <servlet-class>com.cours.servlet.MyFirstServlet</servlet-class>
6      </servlet>
7      <servlet-mapping>
8          <servlet-name>MyFirstServlet</servlet-name>
9          <url-pattern>/MyFirstServlet</url-pattern>
10     </servlet-mapping>
11     <session-config>
12         <session-timeout>
13             30
14         </session-timeout>
15     </session-config>
16 </web-app>
```

Ligne 4 : nom de notre classe servlet.
Ligne 5 : nom complet de classe servlet avec son nom de package.
Ligne 8 : nom de référence de notre servlet.
Ligne 9 : URL relatives au travers desquelles notre servlet sera accessible c'est à dire /MyFirstServlet.

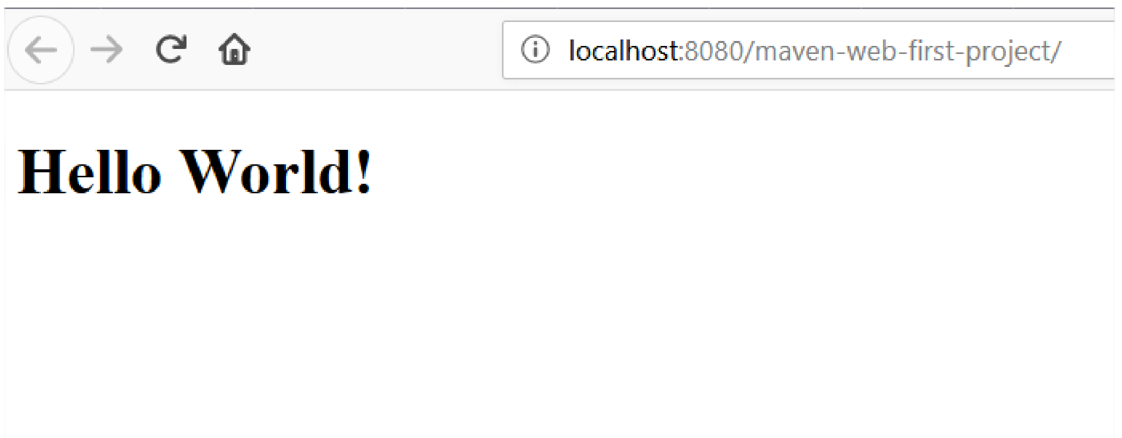
En version copiable :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-
app_3_1.xsd">
  <servlet>
    <servlet-name>MyFirstServlet</servlet-name>
    <servlet-class>com.cours.servlet.MyFirstServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyFirstServlet</servlet-name>
    <url-pattern>/MyFirstServlet</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
</web-app>
```

Faire un **Clean And Build** puis **Run** du projet maven-web-first-project.



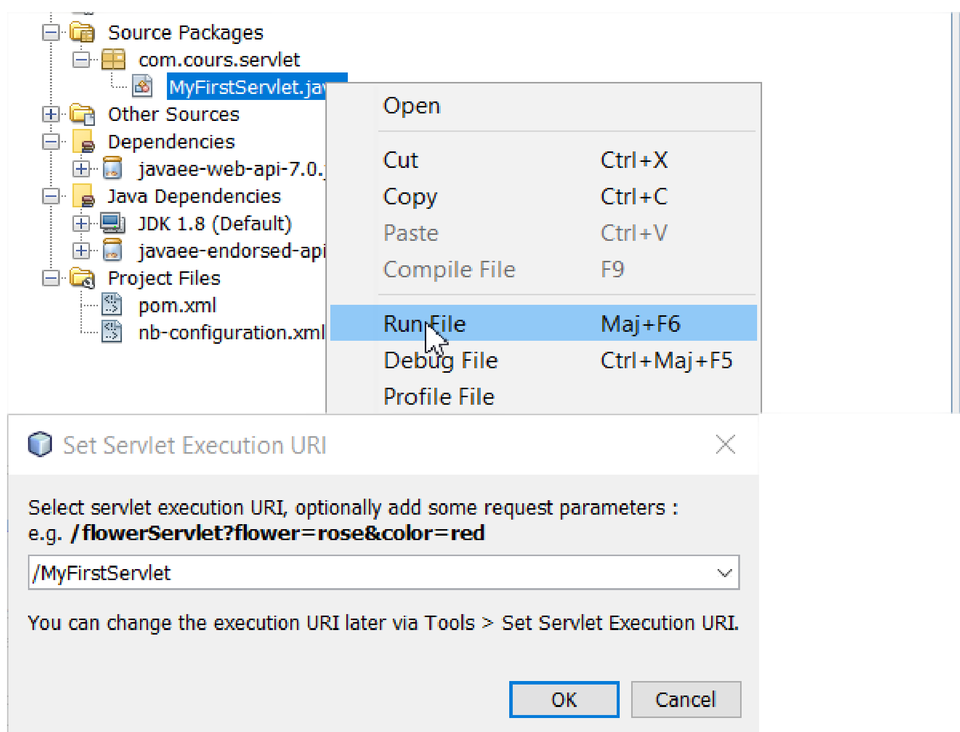
Le **Run** lance l'url http://localhost:8080/maven-web-first-project

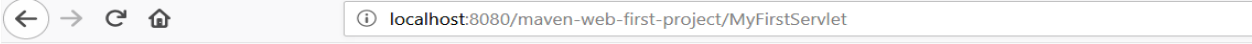


L'application execute par défaut le fichier **index.html** dans **maven-web-first-project /web** dont le contenu est :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Start Page</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

Charger maintenant votre nouveau Servlet **MyFirstServlet** avec :





Servlet MyFirstServlet at /maven-web-first-project

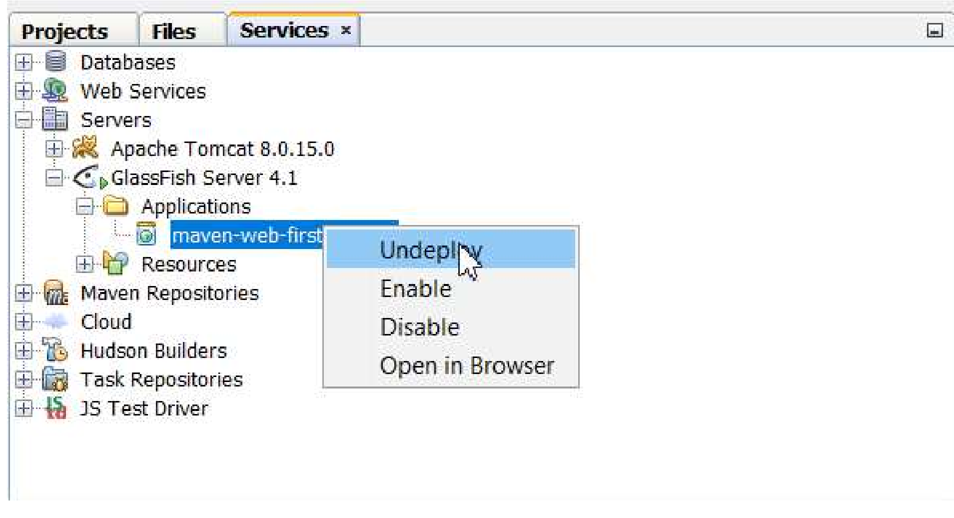
Ce chargement peut faire aussi avec l'Url http://localhost:8080/maven-web-first-project/MyFirstServlet.
On peut aussi définir comme page d'acceuil la Servlet **MyFirstServlet.** C'est-à-dire on veut que lorsqu'on charge l'url http://localhost:8080/maven-web-first-project alors l'application charge la Servlet **MyFirstServlet**.
Pour cela modifions notre fichier **web.xml** en conséquence il devient donc :

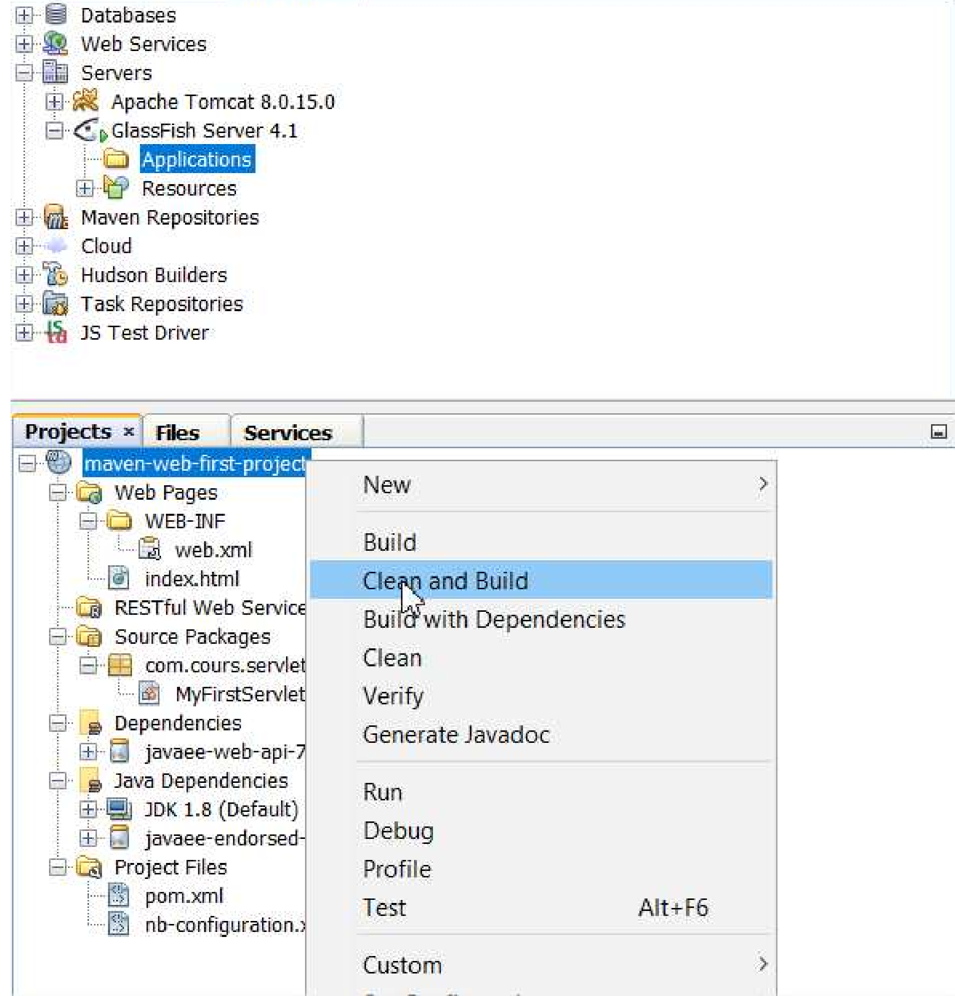
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      <servlet>
4          <servlet-name>MyFirstServlet</servlet-name>
5          <servlet-class>com.cours.servlet.MyFirstServlet</servlet-class>
6      </servlet>
7      <servlet-mapping>
8          <servlet-name>MyFirstServlet</servlet-name>
9          <url-pattern>/MyFirstServlet</url-pattern>
10     </servlet-mapping>
11     <session-config>
12         <session-timeout>
13             30
14         </session-timeout>
15     </session-config>
16     <welcome-file-list>
17         <welcome-file>MyFirstServlet</welcome-file>
18     </welcome-file-list>
19 </web-app>
20
```

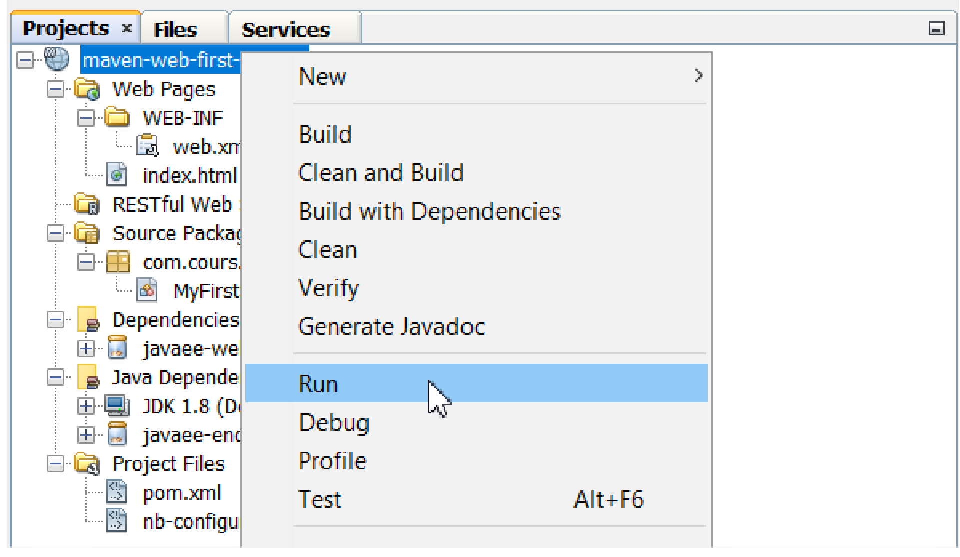
En version copiable :

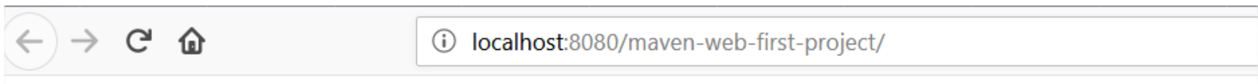
```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
   <servlet>
      <servlet-name>MyFirstServlet</servlet-name>
      <servlet-class>com.cours.servlet.MyFirstServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>MyFirstServlet</servlet-name>
      <url-pattern>/MyFirstServlet</url-pattern>
   </servlet-mapping>
   <session-config>
      <session-timeout>
         30
      </session-timeout>
   </session-config>
   <welcome-file-list>
      <welcome-file>MyFirstServlet</welcome-file>
   </welcome-file-list>
</web-app>
```

Il est possible toute modification soit pris directement à chaud dans le serveur mais cela ne marche pas tout le temps donc il vaudrait mieux enlever le projet **maven-web-first-project** du serveur d'application par le **UnDeploy**, faire un **Clean And Build** et enfin refaire un **Run** du projet **maven-web-first-project** pour le remettre dans le serveur d'aplication.



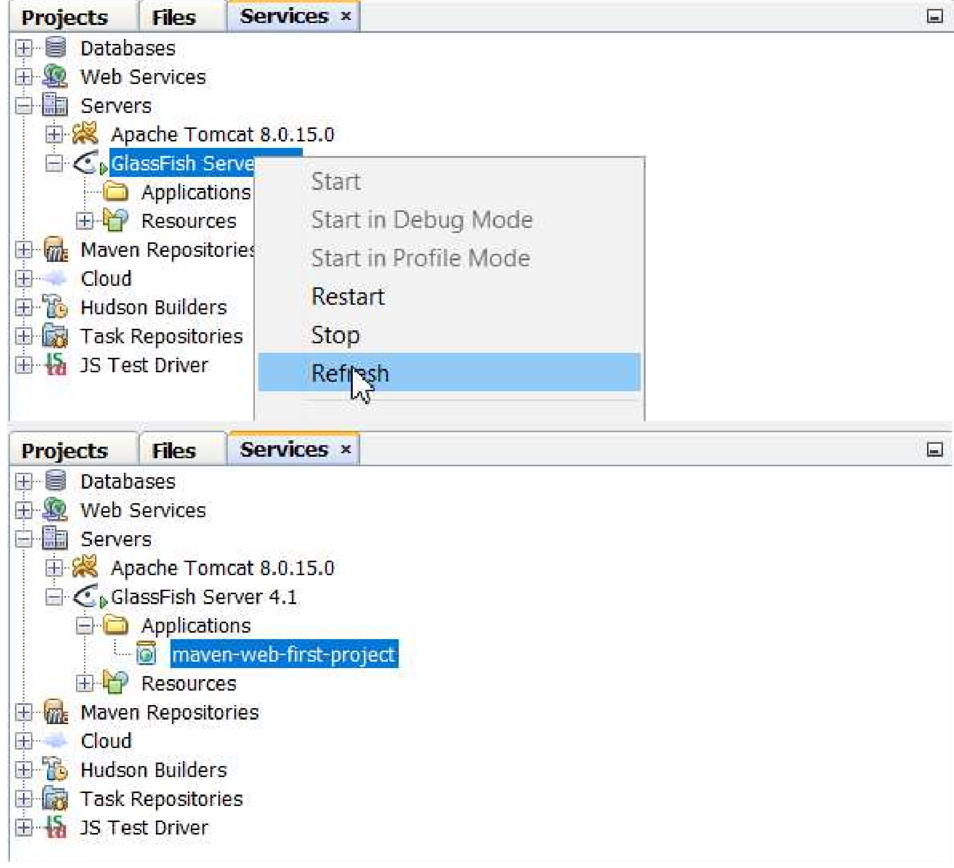






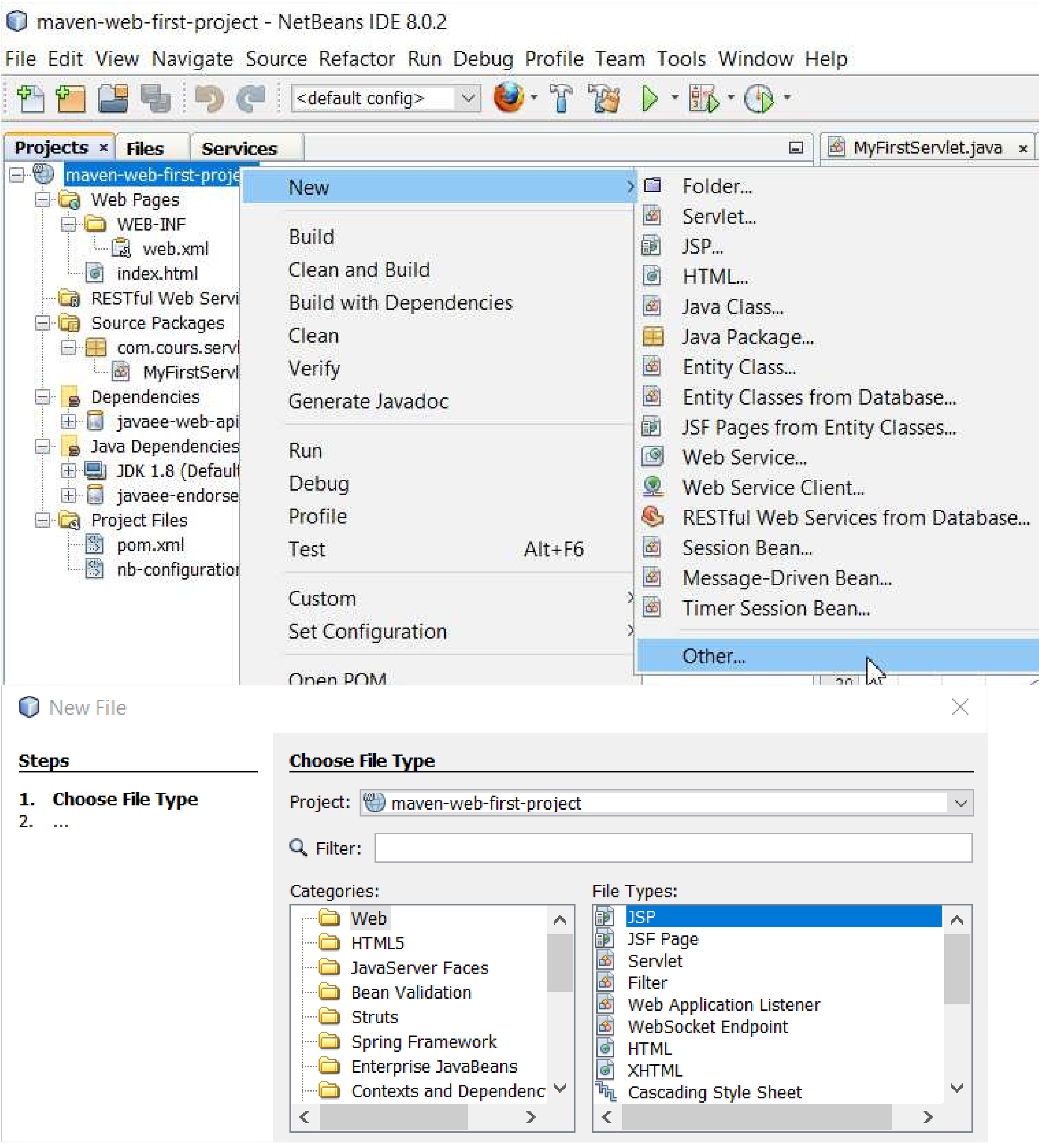
Servlet MyFirstServlet at /maven-web-first-project

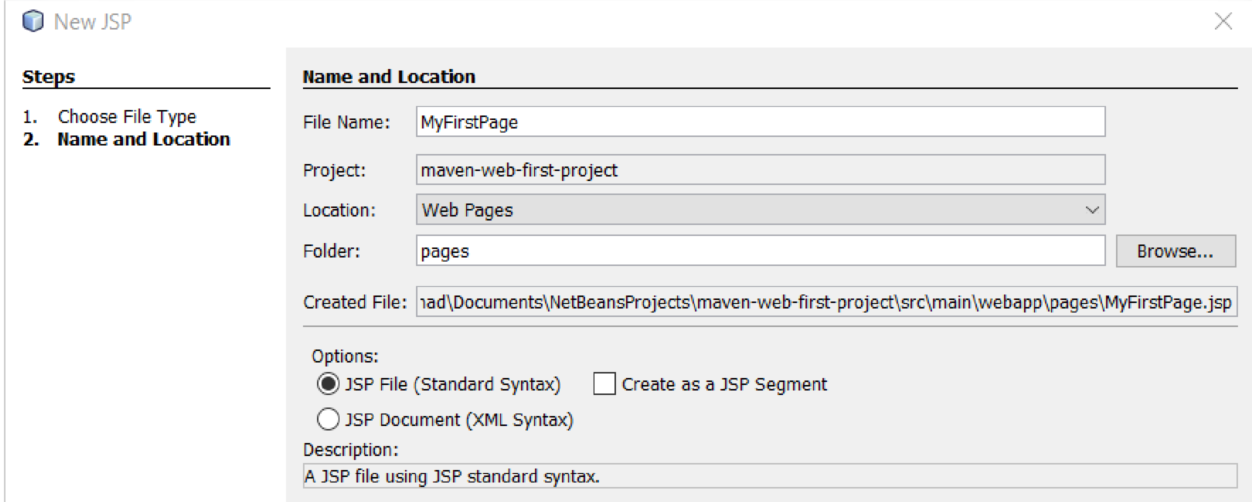
Lorsqu'on retourne dans notre serveur d'application et qu'on fait un refresh on retrouve bien notre application **maven-web-first-project** qui a été déployé.



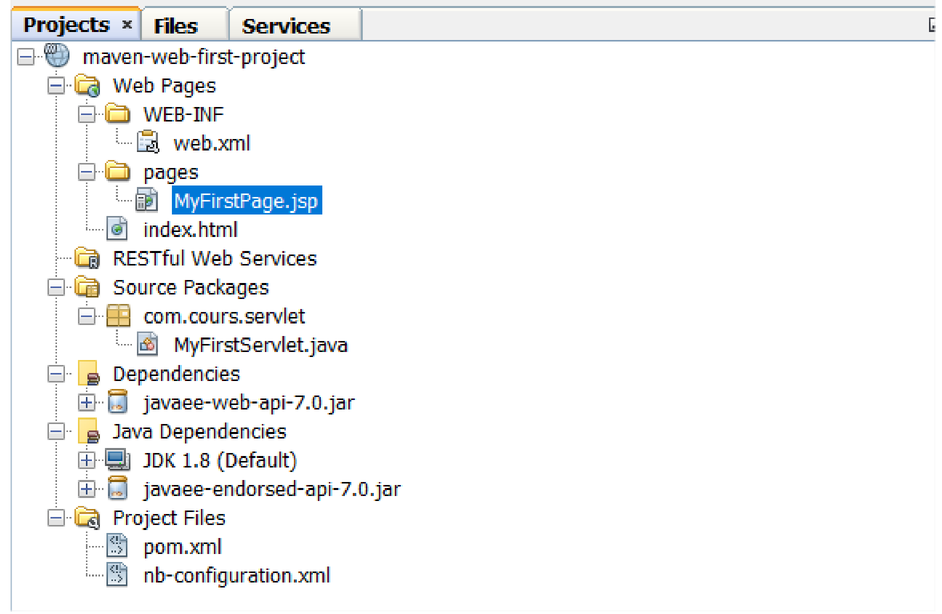
Nous allons maintenant voir un autre aspect du Java Web qui est la vue JSP (Java Server Pages) . Les JSP vont nous permettre de compiler du code Java mais aussi d'y ajouter du code HTML,Css et Javascript qui sera interprété par le navigateur Web. JSP a pour équivalent **Smarty** et **Twig** en PHP ,**Django** en Python et **csharptemplates** en C#. Pour être simple JSP est un outil de templating en Java Web.Il utilise le langage **JSTL** (Java server page Standard Tag Library) pour réaliser ses opérations.

Nous allons donc créer une nouvelle JSP **MyFirstPage.jsp** dans le dossier **pages** qu'on créera en même temps.





Le projet **maven-web-first-project** devient :



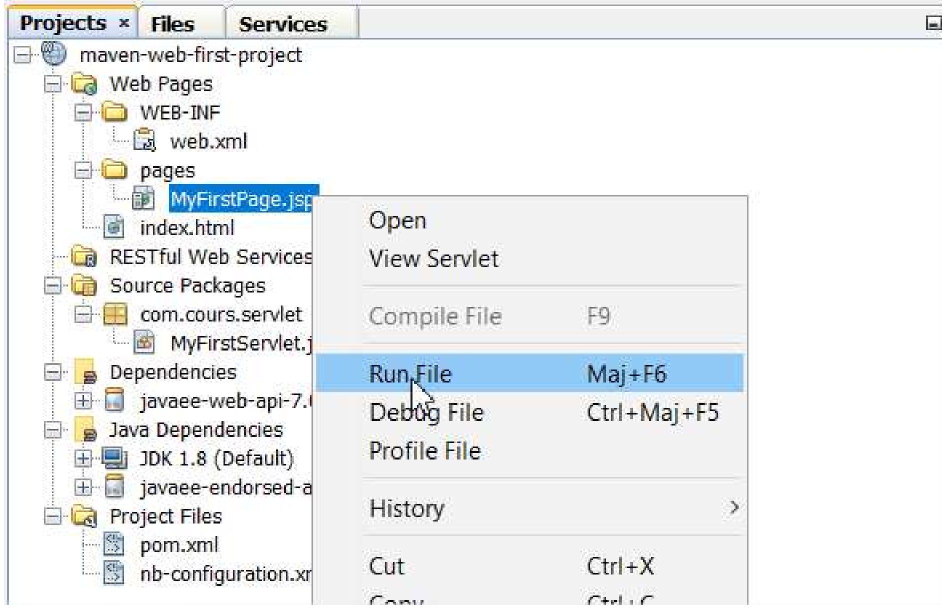
Le contenu par défaut de la JSP est par défaut :

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>JSP Page</title>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

MyFirstPage.jsp peut devenir :

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Notre premier JSP</title>
  </head>
  <body>
    <h1>Ceci est notre premier JSP !</h1>
  </body>
</html>
```

Lancer **MyFirstPage.jsp** avec un **Run File** :



L'opération charge dans le navigateur http://localhost:8080/maven-web-first-project/pages/MyFirstPage.jsp.

Affichons maintenant la vue **MyFirstPage.jsp** à partir de la Servlet **MyFirstServlet**. Pour cela modifier la methode **MyFirstServlet.doGet** (la méthode doGet de la classe **MyFirstServlet**) :

La méthode **MyFirstServlet.doGet** devient :

```
1
2  @Override
3      protected void doGet(HttpServletRequest request, HttpServletResponse response)
4              throws ServletException, IOException {
5          this.getServletContext().getRequestDispatcher("/pages/MyFirstPage.jsp").forward(request, response);
6  }
7
8
9
```

En version copiable :

```
@Override
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
           throws ServletException, IOException {
       this.getServletContext().getRequestDispatcher("/pages/MyFirstPage.jsp").forward(request, response);
}
```

Ligne 2 : L'annotation **@Override** nous signale une re-définition de la methode doGet de la classe abstraite HttpServlet. En effet tous les Servlets hérite de la classe HttpServlet.
Ligne 5 : on affecte à l'instance javax.servlet.RequestDispatcher le chemin de la JSP (aprés le dossier **web** qui est le dossier de référence) , en effet le chemin complet de MyFirstPage.jsp est **maven-web-first-project/web/pages/ MyFirstPage.jsp**.

La classe **MyFirstServlet** donc peut devenir :

```java
package com.cours.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author elhad
 */
public class MyFirstServlet extends HttpServlet {

    /**
     * Méthode appelée lors d'une requête HTTP GET
     *
     * @param request L'objet requête contenant les informations de la requête
     * http
     * @param response L'objet réponse contenant les informations de la réponse
     * http
     * @throws ServletException
     * @throws IOException
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        this.getServletContext().getRequestDispatcher("/pages/MyFirstPage.jsp").forward(request, response);
    }

    /**
     * Méthode appelée lors d'une requête HTTP POST
     *
     * @param request L'objet requête contenant les informations de la requête
     * http
     * @param response L'objet réponse contenant les informations de la réponse
     * http
     * @throws ServletException
     * @throws IOException
     */
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

    }

    /**
     * Méthode d'initialisation de la Servlet
     *
     * @throws ServletException
     */
    @Override
    public void init() throws ServletException {

    }

    @Override
    public String getServletInfo() {
        return " Ceci est ma Servlet MyFirstServlet";
    }

}
```

En version copiable :

```
package com.cours.servlet;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author elhad
 */
public class MyFirstServlet extends HttpServlet {

    /**
     * Méthode appelée lors d'une requête HTTP GET
     *
     * @param request L'objet requête contenant les informations de la requête
     * http
     * @param response L'objet réponse contenant les informations de la réponse
     * http
     * @throws ServletException
     * @throws IOException
     */
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        this.getServletContext().getRequestDispatcher("/pages/MyFirstPage.jsp").forward(request, response);
    }

    /**
     * Méthode appelée lors d'une requête HTTP POST
     *
     * @param request L'objet requête contenant les informations de la requête
     * http
     * @param response L'objet réponse contenant les informations de la réponse
     * http
     * @throws ServletException
     * @throws IOException
     */
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

    }
```

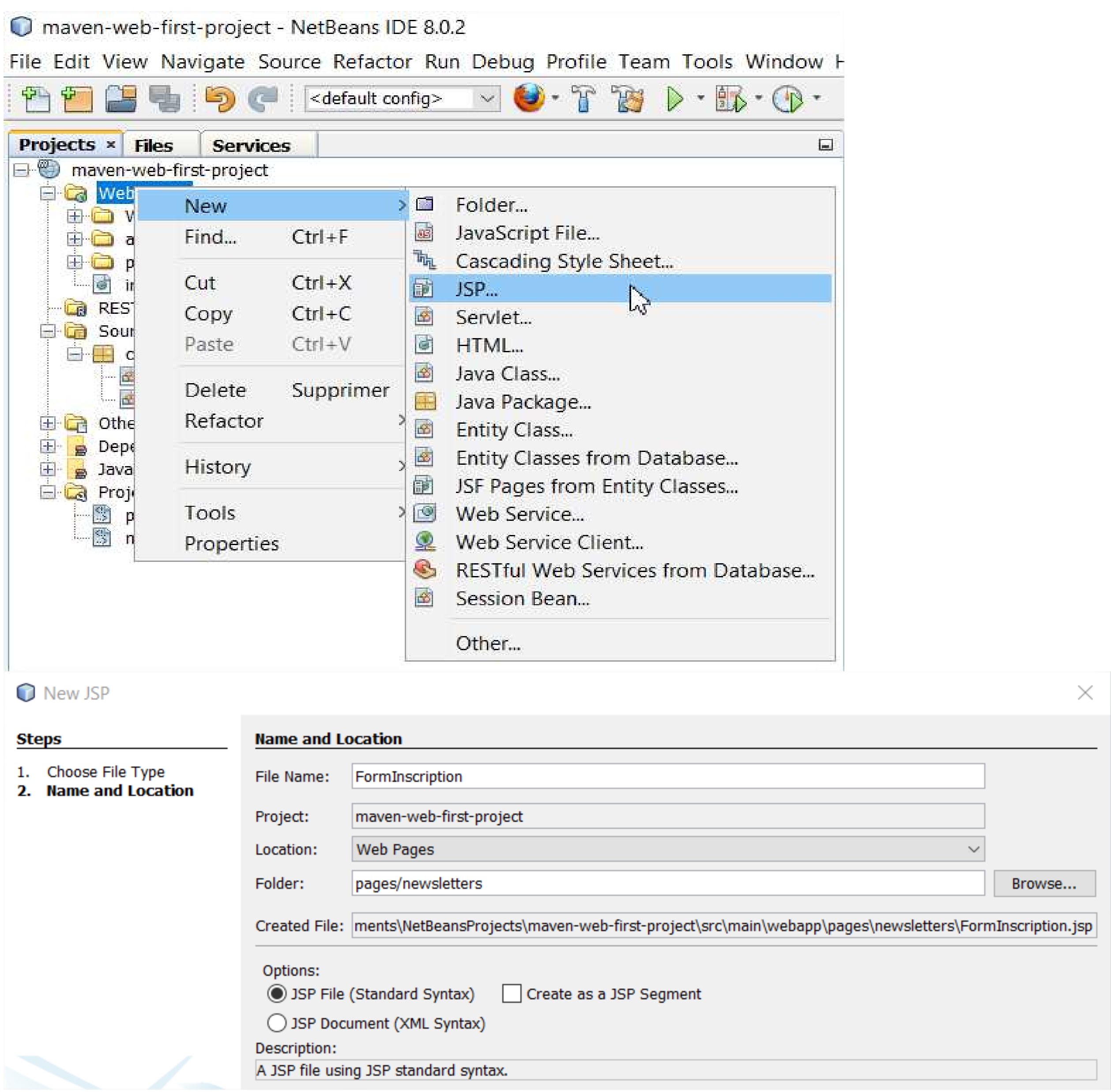
```
/**
 * Méthode d'initialisation de la Servlet
 *
 * @throws ServletException
 */
@Override
public void init() throws ServletException {

}

@Override
public String getServletInfo() {
    return " Ceci est ma Servlet MyFirstServlet";
}

}
```

Ligne 43 : On retrouve la fameuse méthode init qui est lancée au chargement de la Servlet, donc c'est là où vous devez initialiser vos differentes **DAO** ou votre façade (Exemple **IServiceFacade** qui a sa classe d'implémentation **ServiceFacade** dans lequel on retrouve tous les **DAO** des differentes tables de votre source de données).

Un **Clean And Build** puis **Run** du projet **maven-web-first-project** donne :

Nous allons enfin voir comment envoyer et recevoir des informations d'une servlet à la vue et vice-versa. Nous allons pour cela créer une petite fonctionnalité de **Newletters** qui va fournir des informations telles que le nom, l'email et un message. Si ces informations sont saisies correctement alors on envoie les données saisies dans une autre page de confirmation.

Créer la JSP **pages/newsletters/FormInscription.jsp**.

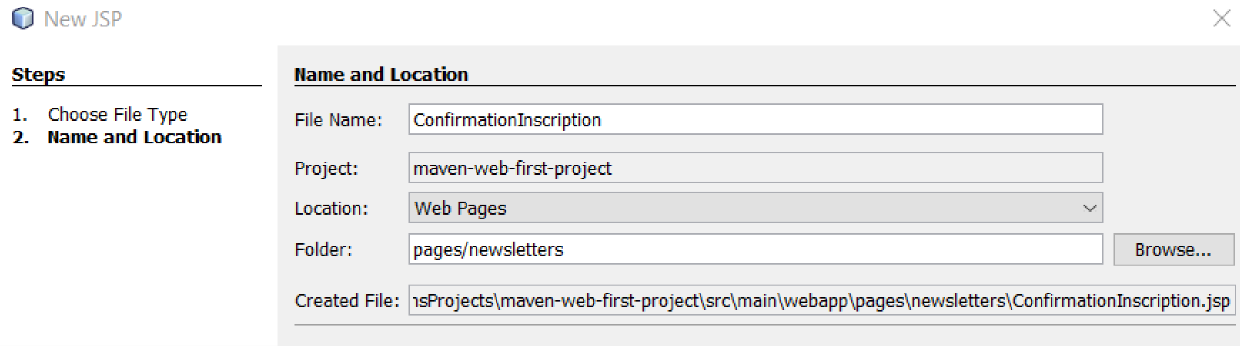


Son contenu sera :

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
     <link rel="stylesheet" href="./assets/css/style.css">
     <script src="./assets/js/script.js" type="text/javascript"></script>
    <title>Formulaire d'inscription</title>
  </head>
  <body>
    <h1>Bienvenue dans votre formulaire d'inscription !</h1>
    <%-- ${pageContext.request.contextPath} --%>
    <form action="/maven-web-first-project/MyFirstServlet" method="post">
      <div>
        <label for="nom">Nom :</label>
        <input type="text" id="nom" name="userName" />
      </div>
      <div>
        <label for="courriel">Courriel :</label>
        <input type="email" id="courriel" name="userEmail" />
      </div>
      <div>
        <label for="message">Message :</label>
        <textarea id="message" name="userMessage"></textarea>
      </div>
      <div class="button">
        <button type="submit">Envoyer votre message</button>
      </div>
    </form>
  </body>
</html>
```

Il faut mettre dans l'attribut name de la balise **form** du formulaire **/maven-web-first-project/MyFirstServlet** qui est le chemin commplet vers la Servlet car en effet **MyFirstServlet** tout seul n'aurait pas suffit à contruire la route en terme d'URL vers la Servlet **MyFirstServlet**. Il était possible de mettre **${pageContext.request.contextPath}/MyFirstServlet**. **${pageContext.request.contextPath}** donne le chemin complet de l'application courante qui est ici **maven-web-first-project**.

Créer la JSP **pages/newsletters/ConfirmationInscription.jsp**.



Son contenu sera :

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%
  String userName = "";
  String userEmail = "";
  String userMessage = "";

  if (request.getAttribute("userName") != null) {
    userName = (String) request.getAttribute("userName");
  }
  if (request.getAttribute("userEmail") != null) {
    userEmail = (String) request.getAttribute("userEmail");
  }
  if (request.getAttribute("userMessage") != null) {
    userMessage = (String) request.getAttribute("userMessage");
  }
%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Page de confirmation de Newletters</title>
  </head>
  <body>
    <h1>Confirmation de votre inscription à la Newsletters !</h1>
    <div>Nom :  <% out.println(userName);%></div>
    <div>Courriel :  <% out.println(userEmail);%></div>
    <div>Message :  <% out.println(userMessage);%></div>
  </body>
</html>
```

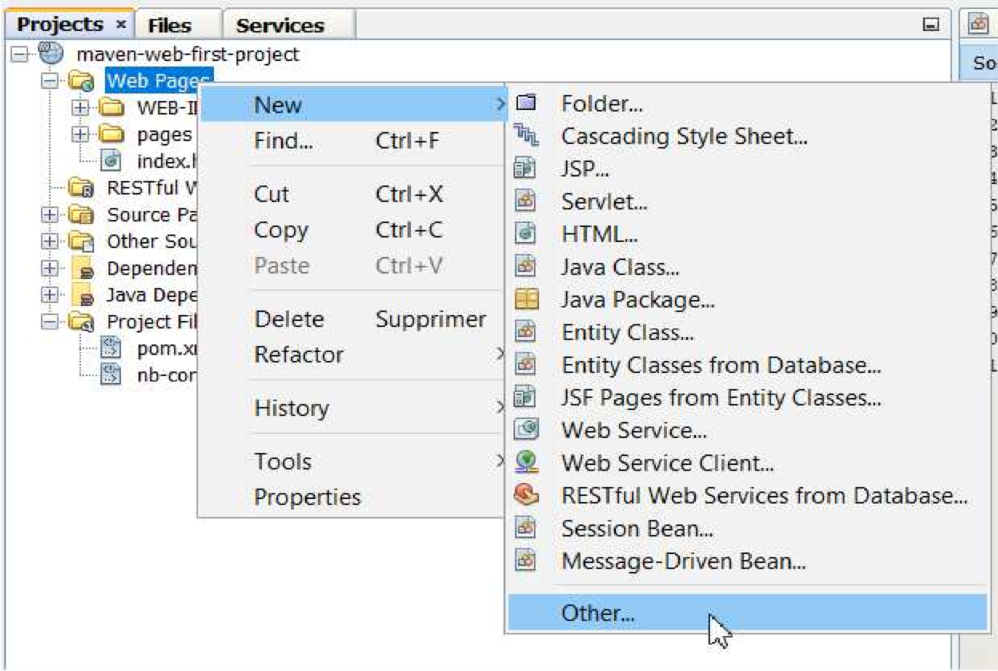
Remarque : il est possible d'inserer des blocs de scriptlet dans les JSP avec :

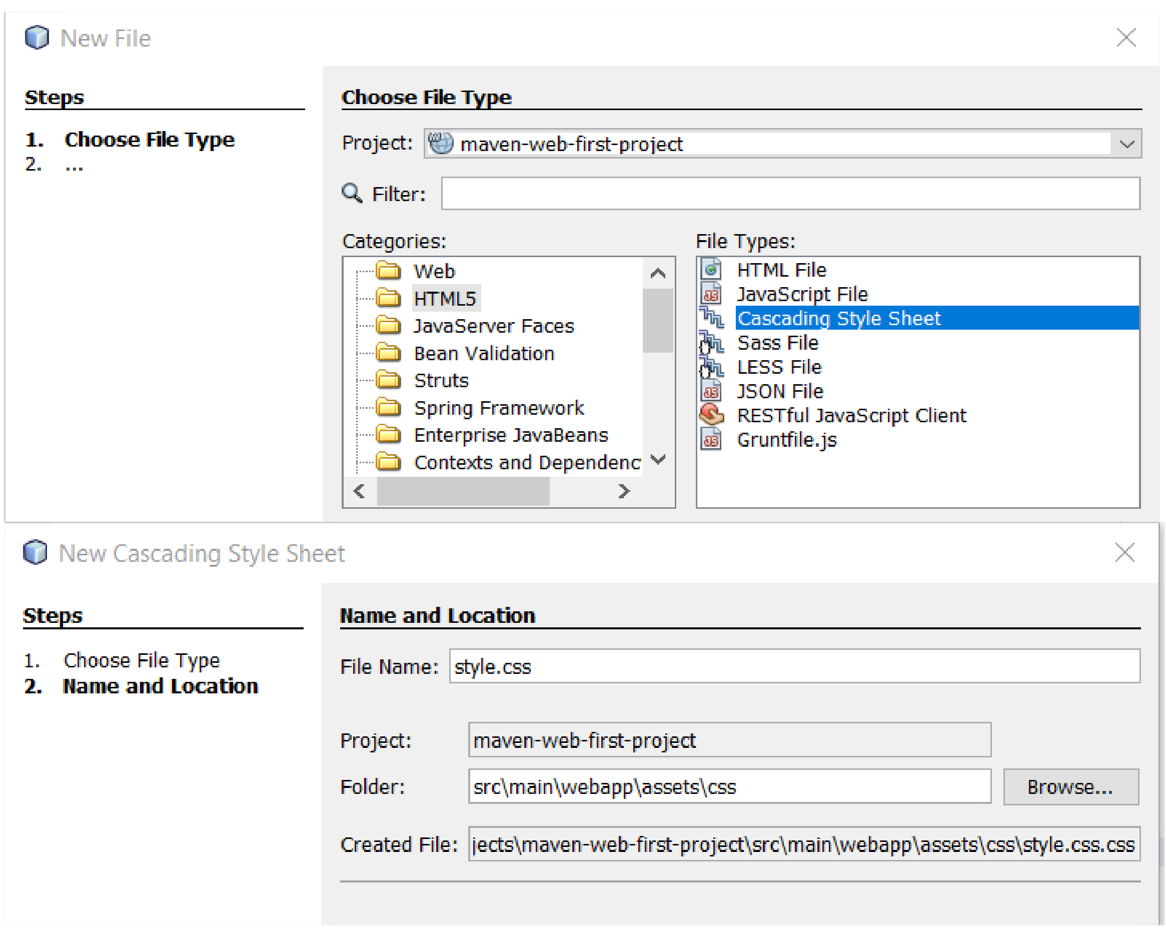
`<% Mon Code Java.  %>`

Exemple :

`<div>`**Hello World!** `</div>` est équivalent à `<div><% out.println("`**Hello World!**`"); %></div>`

Créer le fichier de style css **assets/css/style.css**.

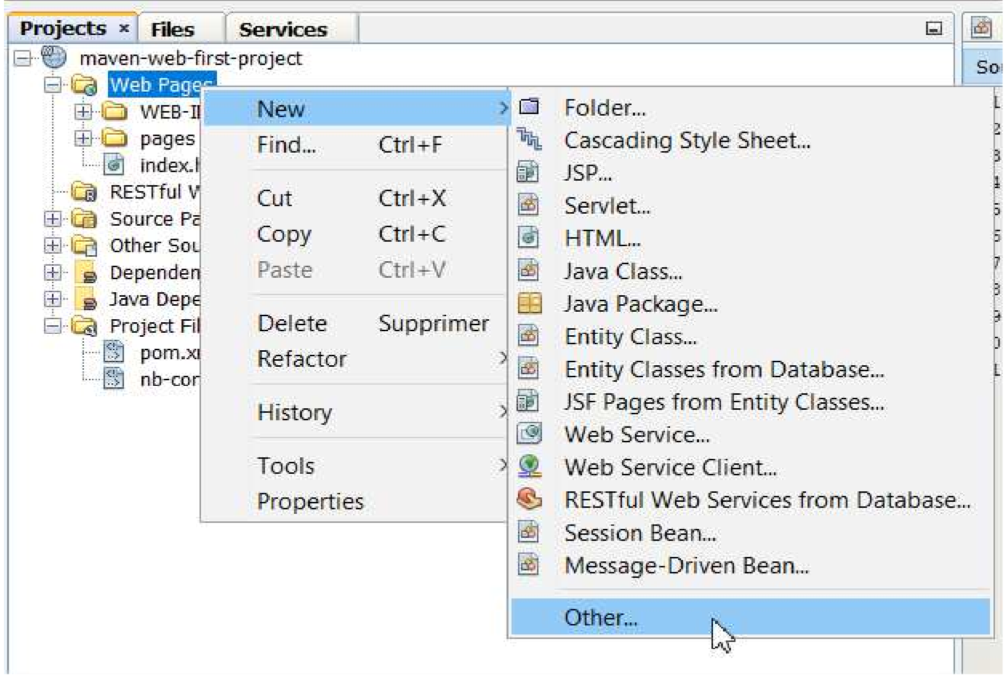


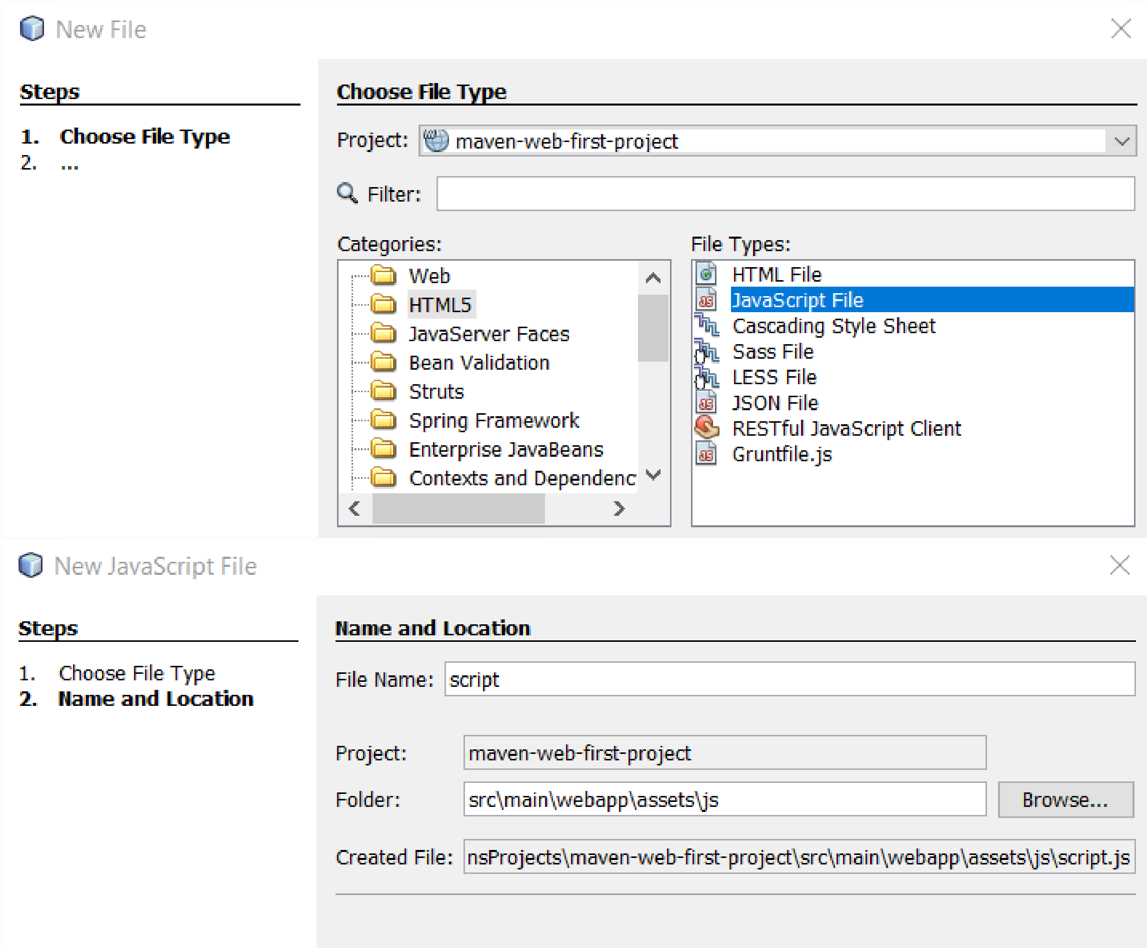


Le fichier **style.css** aura pour contenu :

```
form {
   margin: 0 auto;
   width: 400px;
   padding: 1em;
   border: 1px solid #CCC;
   border-radius: 1em;
}
form div + div {
   margin-top: 1em;
}
label {
   display: inline-block;
   width: 90px;
   text-align: right;
}
input, textarea {
   font: 1em sans-serif;
   width: 300px;
   -moz-box-sizing: border-box;
   box-sizing: border-box;
   border: 1px solid #999;
}
input:focus, textarea:focus {
   border-color: #000;
}
textarea {
   vertical-align: top;
   height: 5em;
   resize: vertical;
}
.button {
   padding-left: 90px;
}
button {
   margin-left: .5em;
}
```

Créer le fichier javascript **assets/js /script.js**.





Le contenu du fichier **script.js** sera :

```
alert("Bravo vous venez de lancer votre premier script dans un projet Web Java");
```

Les méthodes doPost et doGet de la classe **MyFirstServlet** deviennent :

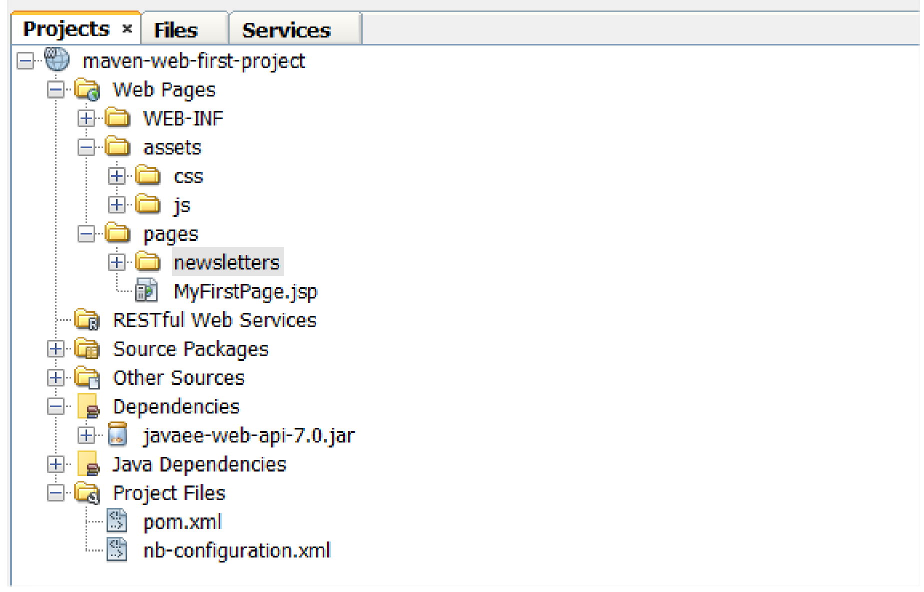
```
1
2  @Override
3  protected void doGet(HttpServletRequest request, HttpServletResponse response)
4          throws ServletException, IOException {
5      this.getServletContext().getRequestDispatcher("/pages/newsletters/FormInscription.jsp").forward(request, response);
6  }
7
8  @Override
9  protected void doPost(HttpServletRequest request, HttpServletResponse response)
10         throws ServletException, IOException {
11     String userName = request.getParameter("userName");
12     String userEmail = request.getParameter("userEmail");
13     String userMessage = request.getParameter("userMessage");
14     System.out.println("userName: " + userName + ", userEmail: " + userEmail + ", userMessage: " + userMessage);
15     request.setAttribute("userName", userName);
16     request.setAttribute("userEmail", userEmail);
17     request.setAttribute("userMessage", userMessage);
18     this.getServletContext().getRequestDispatcher("/pages/newsletters/ConfirmationInscription.jsp").forward(request, response);
19 }
20
```

En version copiable :

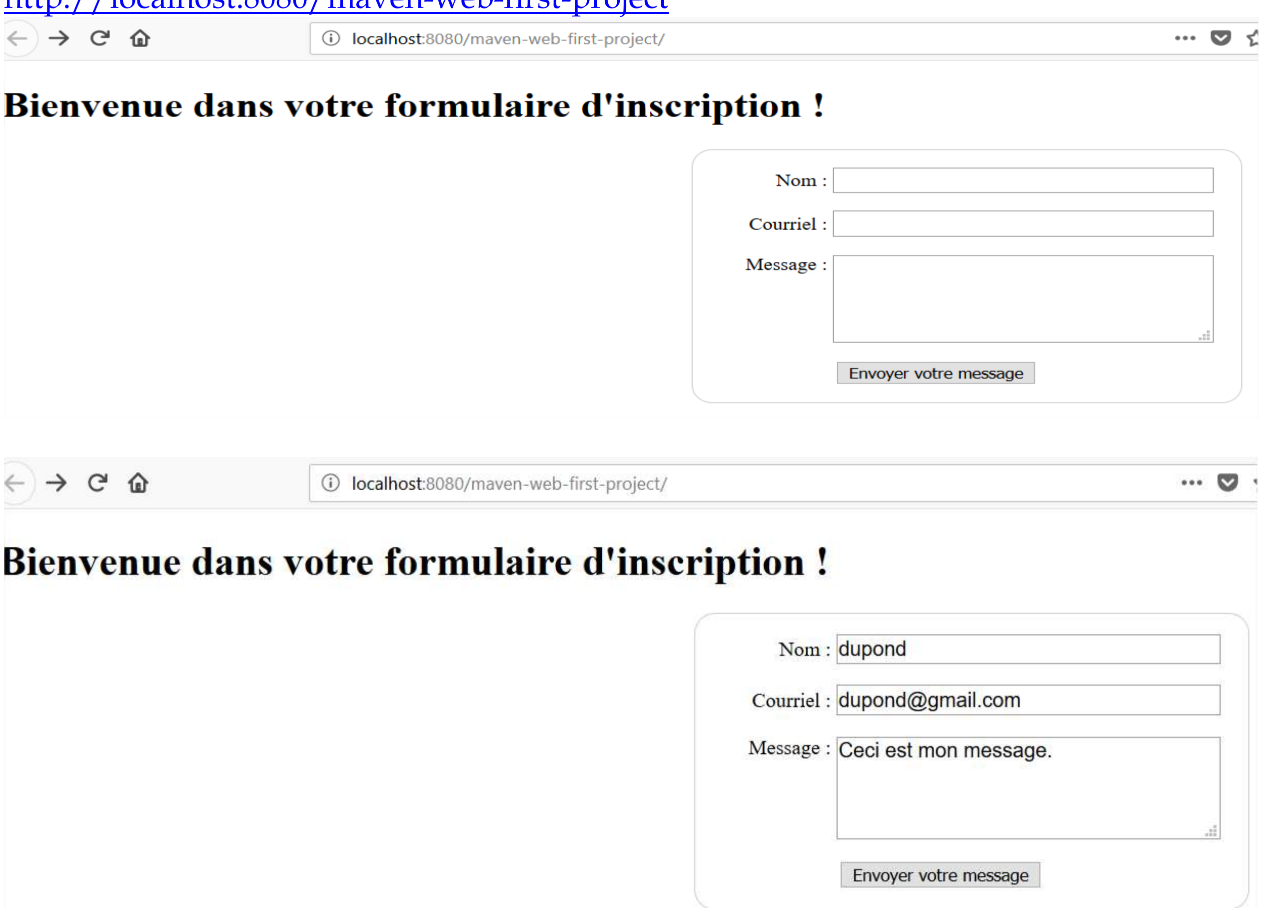
```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    this.getServletContext().getRequestDispatcher("/pages/newsletters/FormInscription.jsp").forward(request, response);
}

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String userName = request.getParameter("userName");
    String userEmail = request.getParameter("userEmail");
    String userMessage = request.getParameter("userMessage");
    System.out.println("userName: " + userName + ", userEmail: " + userEmail + ", userMessage: " + userMessage);
    request.setAttribute("userName", userName);
    request.setAttribute("userEmail", userEmail);
    request.setAttribute("userMessage", userMessage);
    this.getServletContext().getRequestDispatcher("/pages/newsletters/ConfirmationInscription.jsp").forward(request, response);
}
```

Ligne 5 : Chargement du formulaire d'inscription de la Newsletters.
Ligne 11 : Récupération du paramètre **userName** à partir du formulaire .

Le projet **maven-web-first-project** devient :



On obtient au lancement de l'application (c'est-à-dire **Clean And Build + Run**) à travers http://localhost:8080/maven-web-first-project



En cliquant sur **Envoyer votre message** on obtient :

localhost:8080/maven-web-first-project/MyFirstServlet

Confirmation de votre inscription !

Nom : dupond
Courriel : dupond@gmail.com
Message : Ceci est mon message.

VIII) Notion de requête HTTP avec HttpServletRequest.

HttpServletRequest est l'interface qui modélise une requête HTTP. Il expose donc des méthodes qui permettent d'accéder à l'ensemble des paramètres, les en-têtes, et le contenu de la requête.

1. Accès aux paramètres d'une requête

Une requête HTTP peut porter un certain nombre de paramètres. Dans le cas d'une requête de type GET ces paramètres sont codés sur l'URL de requête. Dans le cas d'une requête de type POST, ils se trouvent dans le flux HTTP.

Dans tous les cas, l'API servlet décode ces paramètres, et les expose à travers quatre méthodes :

- getParameterNames() : retourne l'ensemble des noms de paramètres disponibles sur la requête, sous forme d'une énumération.
- getParameter(String) : retourne la valeur du paramètre dont on a donné le nom, s'il existe.
- getParameteValues(String) : retourne l'ensemble des valeurs associées au paramètre dont on a donné le nom, sous forme d'un tableau de String.
- getParameterMap() : retourne une Map dont les clés sont les noms des paramètres, et les valeurs celles des paramètres associés.

Notons que les valeurs des paramètres sont toujours de type String, tout simplement parce que c'est le type utilisé dans l'en-tête HTTP. Il appartient donc à l'application de convertir ces chaînes de caractères dans les bons types.

2. Accès aux éléments de l'en-tête HTTP

Plusieurs méthodes permettent d'accéder aux éléments de l'en-tête.

- getHeaderNames(): retourne les noms des paramètres d'en-tête déclarés dans cette requête, sous forme d'une énumération.
- getHeader(String), getIntHeader(String) et getDateHeader(String) : retournent la valeur du paramètre de l'en-tête dont on donne le nom, sous forme d'une String, d'un entier ou d'une Date respectivement.
- getHeaders() : retourne l'ensemble des noms des éléments de l'en-tête HTTP disponibles dans cette requête, sous forme d'une énumération.
- getMethod() : retourne le nom de la méthode HTTP utilisée pour cette requête (GET, POST, etc...).
- getCharacterEncoding() : retourne l'encodage utilisé pour cette requête. Par défaut l'encodage utilisé par HTTP est ISO-8859-1, mais les autres encodages, notamment l'UTF-8 peut aussi être utilisé. Notons qu'il existe une méthode setCharacterEncoding() qui permet de forcer l'encodage. Elle doit obligatoirement être appelée avant tout accès au contenu de la requête HTTP.
- getLocale() et getLocales() retournent le contenu de l'élément d'en-tête Accept-Language, qui indique la locale par défaut supportée par le navigateur. C'est à partir de ce paramètre qu'une application peut décider d'envoyer des pages en français ou en anglais par exemple.

- getContentType() et getContentLength() permettent d'accéder au type MIME du contenu de la requête, et au nombre d'octets qui la composent.
- getInputStream() retourne un flux binaire de type ServletInputStream sur la requête directement, qu'il est donc possible de traiter sans passer par toutes ces méthodes.

3. Accès aux éléments de l'URL

Les éléments de l'URL d'accès à la ressource que nous sommes en train de traiter sont exposés à travers de trois variables : contextPath, servletPath et pathInfo. Ces trois variables sont accessibles via leurs *getters* standards.

De plus, deux variables sont disponibles : requestURI et requestURL.

- getRequestURI() : URI de la requête, c'est-à-dire ce qui se trouve entre le groupe {nom du protocole, nom du serveur, port} et les paramètres de la requête.
- getRequestURL() : retourne l'URL complète de la requête. Notons que le type de retour est StringBuffer, ce qui permet de modifier cette URL facilement et efficacement.
- getContextPath() : retourne le chemin sous lequel se trouve l'application web dans laquelle se trouve cette servlet. Il correspond à l'attribut path de l'élément Context du fichier context.xml de cette application web, dans le cas de Tomcat. Si la valeur de path vaut /, alors le contextPath est vide.
- getServletPath() : retourne le chemin sous lequel se trouve cette ressource, sous le chemin de l'application web. Cette valeur correspond à l'élément servlet-mapping du fichier web.xml de cette application web. Si servlet-mapping vaut / alors servletPath est vide.
- getPathInfo() : retourne ce qui reste à retourner. Dans le cas d'une servlet, pathInfo est vide. Dans le cas d'une ressource statique (une image, ou une page HTML), pathInfo correspond au nom de cette ressource.

Chacune de ces trois variables, si elle n'est pas vide, commence toujours par le caractère /.

Notons qu'une page JSP est une servlet, que son servletPath porte le nom de cette page et que son pathInfo est vide.

Notons enfin que l'on a toujours :

requestURI = contextPath + servletPath + pathInfo

4. Accès aux paramètres du client

Trois méthodes nous permettent d'accéder aux paramètres du client qui fait la requête :

- getRemoteHost() : nous fournit le nom complet du client. Plus précisément, il s'agit du nom du dernier proxy utilisé si ce client fait sa requête au travers d'un proxy.
- getRemoteAddr() : même méthode que la précédente, sauf qu'elle retourne l'adresse IP du client.
- getRemotePort() : même méthode que la précédente, nous retourne le port du client.

5. Accès aux informations de sécurité

Tous les serveurs de servlets exposent un mécanisme d'authentification standard, qui permet de fixer l'identité d'un utilisateur. La configuration de la sécurité associe les utilisateurs à des rôles, et c'est à ces rôles que le serveur donne des droits et impose des restrictions. La requête expose deux méthodes permettant d'accéder à des informations sur les rôles que possède un utilisateur authentifié :

- getUserPrincipal() : retourne un objet Principal qui encapsule le nom de l'utilisateur authentifié.
- isUserInRole(String) : retourne true si l'utilisateur authentifié qui est à l'origine de cette requête est déclaré dans le rôle passé en paramètre.

6. Accès à la session, au contexte et aux informations d'initialisation

Une méthode permet d'accéder à la session HTTP dans laquelle cette requête se place.

- getSession() : retourne un objet de type HttpSession, détaillé dans la suite de ce chapitre. Notons que l'appel à cette méthode crée une session s'il n'en existe pas déjà une, ce qui est en général indésirable. La même méthode existe dans une seconde version, qui prend un booléen en paramètre. Si ce booléen est false, alors aucune session n'est créée. Dans ce cas la méthode retourne nulle.
- getServletContext() : retourne un objet de type ServletContext, qui modèlise l'application web.
- getServletConfig() : retourne un objet de type ServletConfig, qui encapsule les paramètres d'initialisation de la servlet. Cet objet expose deux méthodes : getInitParameterNames() et getInitParameter(String), qui permettent d'accèder aux noms des paramètres d'initialisation d'une part, et aux valeurs associées d'autre part.

IX) Notion de réponse HTTP avec ServletResponse et HttpServletResponse.

La réponse d'une servlet à un client est un flux HTTP modélisé par deux classes : ServletResponse et HttpServletResponse. Ces deux classes exposent des méthodes qui permettent de fixer cette réponse et de la paramétrer, notamment de fixer le type MIME de cette réponse, ou l'encodage des caractères dans le cas d'une réponse textuelle.

Le contenu de la réponse d'une servlet n'est pas envoyé au client directement ; il est tout d'abord enregistré dans un buffer. La classe Response expose quelques méthodes qui permettent de contrôler ce buffer.

Une réponse peut être contrôlée en mode caractère, via un PrintWriter, ou en mode binaire, via un ServletOutputStream.

Enfin, la classe HttpServletResponse expose quelques méthodes qui permettent de contrôler les en-têtes HTTP associés à cette réponse. Voyons tout ceci en détails.

1. Contrôle du buffer de sortie

Le buffer de sortie est contrôlé par la classe ServletResponse. Cette classe expose les méthodes suivantes :

- reset() et resetBuffer() : ces méthodes vident le buffer de son contenu. La méthode reset() efface également les paramètres de l'en-tête HTTP qui auraient été fixés.
- setBufferSize(int) et getBufferSize() : contrôlent la taille du buffer.
- setContentType(String) et getContentType() : contrôlent le type MIME du contenu porté par ce buffer. Ce type doit être fixé avant que le buffer ne soit envoyé au client, même partiellement.
- setCharacterEncoding(String) et getCharacterEncoding() : contrôlent le codage des caractères de ce buffer. Même chose : cet encodage doit être fixé avant que le buffer ne soit retourné au client, même partiellement.
- setContentLength() : fixe la taille du contenu envoyé au client. Cette méthode est portée par la classe ServletResponse, et fixe l'attribut HTTP Content-length dans le cas d'une réponse HTTP. Il n'y a pas de méthode getContentLength().
- setLocale(Locale) et getLocale() : contrôlent la langue dans laquelle la réponse est envoyée. Comme pour les autres méthodes, cette *locale* doit être fixée avant l'envoi du buffer.
- flushBuffer() : envoi le contenu du buffer au client. La méthode isCommited permet de tester si cet envoi à eut lieu.

Enfin les deux méthodes getWriter() et getOuputStream() permettent d'accéder au contenu de la réponse au travers d'un objet de type PrintWriter (extension de Writer), ou d'un objet ServletOutputStream (extension de OutputStream).

2. **Contrôle de la réponse HTTP**

La classe HttpServletResponse offre trois contrôles sur la réponse HTTP :

- la possibilité de fixer les paramètres de l'en-tête. Les méthodes qui prennent en charge cette fonctionnalité sont addHeader(String, String), addIntHeader(String, int), etc... On peut également fixer le statut de cette réponse par appel à la méthode setStatus(int).
- le retour d'un code HTTP, afin de signaler une erreur ou un problème dans le traitement de la requête. Ce point est géré par la méthode sendError(int, String).
- enfin la redirection de la requête vers une autre URL : sendRedirect(String).

X) L’API JSTL (Tags JSTL).

JSTL est l'acronyme de Java server page Standard Tag Library. C'est un ensemble de tags personnalisés développés qui propose des fonctionnalités souvent rencontrées dans les JSP :

- Tag de structure (affichage, itération, conditionnement ...)
- Internationalisation
- Exécution de requêtes SQL
- Utilisation de documents XML

On peut citer quatre grandes bibliothèques de tags JSTL :

Rôle	TLD	Uri
Fonctions de base	c.tld	http://java.sun.com/jsp/jstl/core
Traitements XML	x.tld	http://java.sun.com/jsp/jstl/xml
Internationalisation	fmt.tld	http://java.sun.com/jsp/jstl/fmt
Traitements SQL	sql.tld	http://java.sun.com/jsp/jstl/sql

Nous pouvons par exemple utiliser le Tag **Core** afin de simplifier tout ce qui est affichage de données.
Ces librairies sont normalement déjà incluses mais il peut arriver qu’on soit obligé de les rajouter directement au projet.
Par exemple si vous utilisez le serveur d’application **GlassFish** vous n’avez rien à ajouter dans votre **pom.xml** pour utiliser l’API JSTL.
Si vous utilisez **Tomcat** vous aurez peut-être à ajouter la dépendance ci-dessous dans votre **pom.xml**.

```
<dependency>
  <groupId>org.glassfish.web</groupId>
  <artifactId>jstl-impl</artifactId>
  <version>1.2</version>
  <exclusions>
    <exclusion>
      <artifactId>servlet-api</artifactId>
      <groupId>javax.servlet</groupId>
    </exclusion>
    <exclusion>
      <artifactId>jsp-api</artifactId>
      <groupId>javax.servlet.jsp</groupId>
    </exclusion>
  </exclusions>
</dependency>
```

Le langage **EL** (Expression Language) est le langage utilisé dans les JSP avec la librairie JSTL.
Ce langage propose un certain nombre d'opérateurs :
EL propose aussi différents opérateurs :

<table>
<tr><th>Operateur</th><th>Rôle</th></tr>
<tr><td>.</td><td>Obtenir une propriété d'un objet : ${param.nom}</td></tr>
<tr><td>[]</td><td>Obtenir une propriété par son nom ou son indice :
${param[" nom "]}
${row[1]}</td></tr>
<tr><td>Empty</td><td>Teste si un objet est null ou vide si c'est une chaîne de caractère. Renvoie un booléen : ${empty param.nom}</td></tr>
<tr><td>==
eq</td><td>teste l'égalité de deux objets</td></tr>
<tr><td>!=
ne</td><td>teste l'inégalité de deux objets</td></tr>
<tr><td><
lt</td><td>test strictement inférieur</td></tr>
<tr><td>>
gt</td><td>test strictement supérieur</td></tr>
<tr><td><=
le</td><td>test inférieur ou égal</td></tr>
<tr><td>>=
ge</td><td>test supérieur ou égal</td></tr>
<tr><td>+</td><td>Addition</td></tr>
<tr><td>-</td><td>Soustraction</td></tr>
<tr><td>*</td><td>Multiplication</td></tr>
<tr><td>/
div</td><td>Division</td></tr>
<tr><td>%
mod</td><td>Modulo</td></tr>
<tr><td>&&
and</td><td></td></tr>
<tr><td>||
or</td><td></td></tr>
<tr><td>!
not</td><td>Négation d'une valeur</td></tr>
</table>

Il existe aussi un certain nombre de **variables de contexte** contenu dans des variables JSTL :

Variable	Rôle	Exemple
pageScope	variable contenue dans la portée de la page (PageContext)	${pageScope. maVarPage}
requestScope	variable contenue dans la portée de la requête (HttpServletRequest)	${requestScope.maVarRequest}
sessionScope	variable contenue dans la portée de la session (HttpSession)	${sessionScope.maVarSession}
applicationScope	variable contenue dans la portée de l'application (ServletContext)	${applicationScope.maVarApplication}
param	paramètre de la requête http	${param.momParam}
paramValues	paramètres de la requête sous la forme d'une collection	
header	en-tête de la requête	
headerValues	en-têtes de la requête sous la forme d'une collection	
initParam	paramètre d'initialisation	
cookie	cookie	
pageContext	objet PageContext de la page	

L'utilisation de la librairie « **Core** » est assez simple il suffit de l'inclure dans votre JSP avec :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

On utilise la librairie « Core » avec le mot clef « **c :** ».

<c:out value="Mon Texte" /> affiche dans le navigateur « **Mon Texte ».**
<c:out value="${maVariable}" /> affiche dans le navigateur le contenu de la variable « **maVariable».**

On peut aussi faire des tests en suivant des conditions avec le «**c:if**».

```
<c:if test="${not empty maVariable}">
</c:if>
```

Des blocs **if-else** imbriqués avec le «c:when» :

```
<c:choose>
   <c:when test="${maVariable == 1}">
      <c:out value="${maVariable} vaut 1" />
   </c:when>
   <c:when test="${maVariable == 2}">
      <c:out value="${maVariable} vaut 2" />
   </c:when>
   <c:when test="${maVariable == 3}">
      <c:out value="${maVariable} vaut 3" />
   </c:when>
   <c:otherwise>
      <c:out value="${maVariable} ne vaut ni 1 ni 2 ni 3" />
   </c:otherwise>
</c:choose>
```

On peut aussi faire des boucles avec le «c:forEach» :

```
<c:forEach begin="1" end="12" var="i" step="3">
   <c:out value="${i}"/><br>
</c:forEach>
```

Les dépendances dans le fichier **pom.xml** de **maven-web-first-project** deviennent :

```
<dependencies>
   <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-web-api</artifactId>
      <version>7.0</version>
      <scope>provided</scope>
   </dependency>
   <!-- Pour les utilisateur du serveur d'application Tomcat ajouter la dependance org.glassfish.web -->
   <dependency>
      <groupId>org.glassfish.web</groupId>
      <artifactId>jstl-impl</artifactId>
      <version>1.2</version>
      <exclusions>
         <exclusion>
            <artifactId>servlet-api</artifactId>
            <groupId>javax.servlet</groupId>
         </exclusion>
         <exclusion>
            <artifactId>jsp-api</artifactId>
            <groupId>javax.servlet.jsp</groupId>
         </exclusion>
      </exclusions>
   </dependency>
```

```
</dependencies>
```

ConfirmationInscription.jsp devient donc plus simple avec l'API JSTL :

```
1  <%@page contentType="text/html" pageEncoding="UTF-8"%>
2  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
3  <!DOCTYPE html>
4  <html>
5      <head>
6          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7          <title><c:out value="Page de confirmation" /></title>
8      </head>
9      <body>
10         <h1>Confirmation de votre inscription !</h1>
11         <div>Nom :  <c:out value="${userName}" /></div>
12         <div>Courriel :  <c:out value="${userEmail}" /></div>
13         <div>Message :  <c:out value="${userMessage}" /></div>
14     </body>
15 </html>
```

En version copiable :

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title><c:out value="Page de confirmation" /></title>
   </head>
   <body>
      <h1>Confirmation de votre inscription !</h1>
      <div>Nom :  <c:out value="${userName}" /></div>
      <div>Courriel :  <c:out value="${userEmail}" /></div>
      <div>Message :  <c:out value="${userMessage}" /></div>
   </body>
</html>
```

Ligne 2 : inclusion du taglib jstl **core**.

Au lancement de l'application on obtient la même chose que tout à l'heure.

XI) Notion de portée de variable JSTL.

Le tag set permet de stocker une variable dans une portée particulière (page, requête, session ou application).

Il possède plusieurs attributs :

Attribut	Rôle
value	valeur à stocker
target	nom de la variable contenant un Bean dont la propriété doit être modifiée
property	nom de la propriété à modifier
var	nom de la variable qui va stocker la valeur
scope	portée de la variable qui va stocker la valeur

```
<c:set var="maVariablePage" value="test" scope="page" />
```

Ou encore

```
<c:set var="maVariablePage" value=" test " scope="page"> test </c:set>
```

On peut aussi affecter une valeur dynamiquement.

```
<c:set var="maVariablePage" value="${param.id}" scope="page" />
```

Il existe 4 type de portées

La portée **page** ou scope **page** veut dire que la variable est uniquement valable pour la page JSP correspondante.

```
<c:set var="maVariablePage" value="toto" scope="page" />
```

La portée **request** ou scope **request** veut dire que la variable est valable durant toute la requête HTTP.

```
<c:set var="maVariableRequest" value="tata" scope="request" />
```

La portée **session** ou scope **session** veut dire que la variable est valable durant toute la session utilisateur.

```
<c:set var="maVariableSession" value="titi" scope="session" />
```

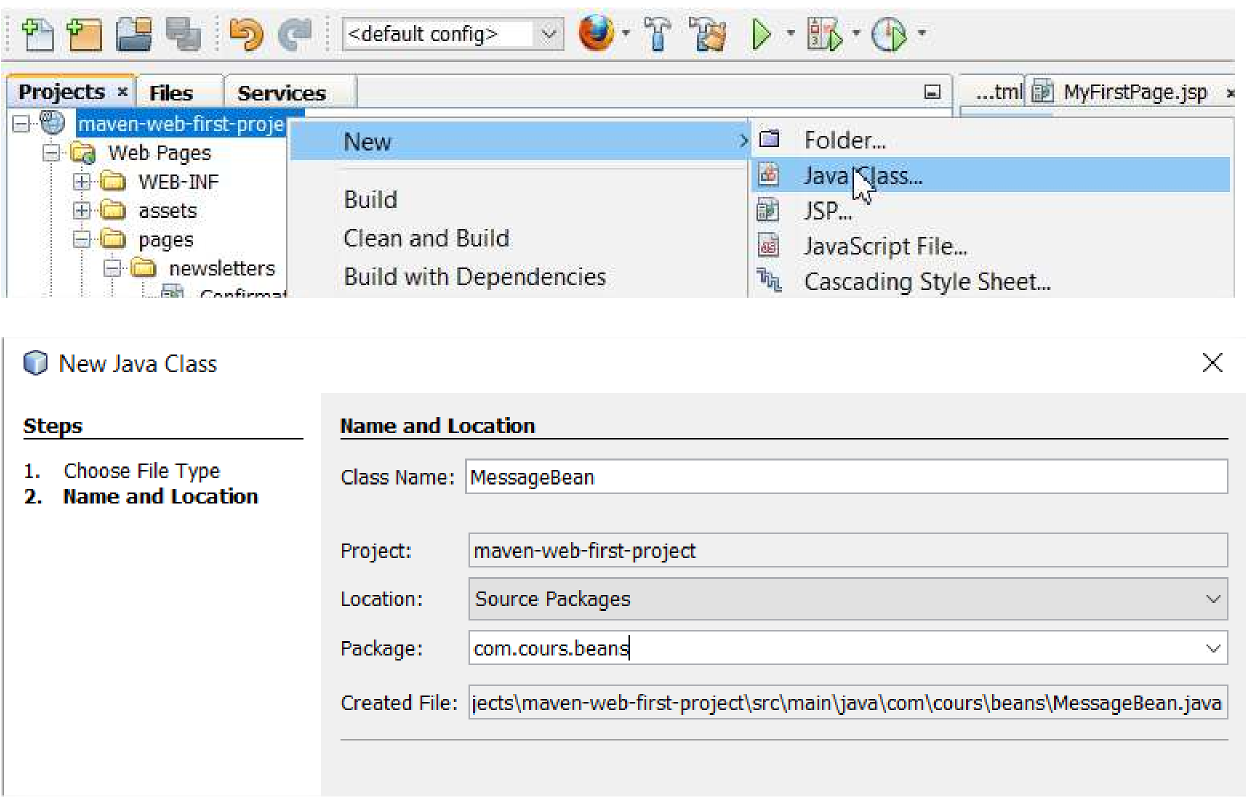
La porté **application** ou scope **application** veut dire que la variable est valable partout c'est-à-dire dans toutes les pages de l'application.

```
<c:set var="maVariableAppication" value="popo" scope="application" />
```

XII) Manipuler un JavaBean.

Un Java Bean est une classe publique qui a des champs privés (**private**) ou protégés(**protected**) qui sont accessibles via des méthodes publiques getter et setter, suivant des règles de nommage de Sun(get ou set puis première lettre en majuscule et les premières lettres des autres mot en majuscule et le reste en miniscule. Exemple pour un attribut « prenomPersonneFamille » nous aurons les methode getPrenomPersonneFamille et setPrenomPersonneFamille).
Le JavaBean peut implémenter l'interface Serializable et dans ce cas il devient ainsi persistant et son état peut être sauvegardé en base de données par exemple.

Soit **MessageBean** notre JavaBean.



Son contenu sera le suivant :

```java
package com.cours.beans;

public class MessageBean {

    private String nom = null;
    private String courriel = null;
    private String message = null;

    public MessageBean() {

    }

    public MessageBean(String nom, String courriel, String message) {
        this.nom = nom;
        this.courriel = courriel;
        this.message = message;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public String getCourriel() {
        return courriel;
    }

    public void setCourriel(String courriel) {
        this.courriel = courriel;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    @Override
    public String toString() {
        return String.format("[nom=%s, courriel=%s, message=%s]", nom, courriel, message);
    }
}
```

Si **MessageBean** était sérialisable (implémente l'interface java.io.Serializable donc peut circuler à travers un réseau informatique) alors sa déclaration serait :

```
public class MessageBean implements Serializable
```

Avec l'utilisation du Bean **MessageBean**,la méthode **MyFirstServlet.doPost** devient :

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    MessageBean messageBean = new MessageBean(request.getParameter("userName"), request.getParameter("userEmail"),
request.getParameter("userMessage"));
    request.setAttribute("messageBean", messageBean);
    this.getServletContext().getRequestDispatcher("/pages/newsletters/ConfirmationInscription.jsp").forward(request, response);
}
```

ConfirmationInscription.jsp devient :

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title><c:out value="Page de confirmation" /></title>
  </head>
  <body>
    <h1>Confirmation de votre inscription !</h1>
    <div>Nom :  <c:out value="${messageBean.nom}" /></div>
    <div>Courriel :  <c:out value="${messageBean.courriel}" /></div>
    <div>Message :  <c:out value="${messageBean.message}" /></div>
    <div>MessageBean :  <c:out value="${messageBean}" /></div>
  </body>
</html>
```

On obtient à la confirmation :

L'instruction **`<c:out value="${messageBean}" />`** affiche le Bean « **messageBean** » donc il exécute implicitement la méthode « toString » de la classe « **MessageBean** ».

Il est aussi possible d'instancier le bean **MessageBean** dans la JSP **ConfirmationInscription.jsp**.

ConfirmationInscription.jsp devient :

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
      <title><c:out value="Page de confirmation" /></title>
   </head>
   <body>
      <h1>Confirmation de votre inscription !</h1>
      <div>Nom :  <c:out value="${messageBean.nom}" /></div>
      <div>Courriel :  <c:out value="${messageBean.courriel}" /></div>
      <div>Message :  <c:out value="${messageBean.message}" /></div>
      <jsp:useBean id="myOtherMessageBean" class="com.cours.beans.MessageBean" scope="request" >
        <jsp:setProperty name="myOtherMessageBean" property="nom" value="Mon Nom" />
        <jsp:setProperty name="myOtherMessageBean" property="courriel" value="mail@mail.com" />
        <jsp:setProperty name="myOtherMessageBean" property="message" value="Mon Message" />
      </jsp:useBean>
      <div>Mon autre MessageBean :   <c:out value="${myOtherMessageBean}" /></div>
   </body>
</html>
```

On obtient à la confirmation :

Confirmation de votre inscription !

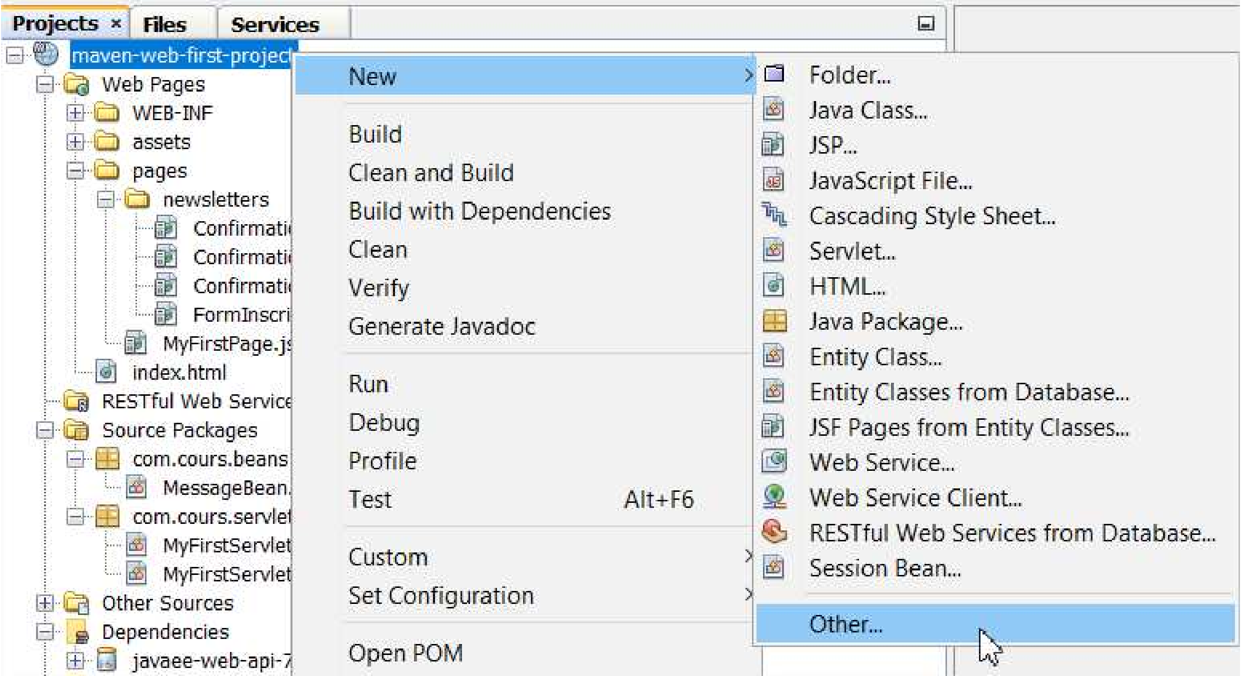
Nom : dupond
Courriel : dupond@gmail.com
Message : Ceci est mon message.
Mon autre MessageBean : [nom=Mon Nom, courriel=mail@mail.com, message=Mon Message]

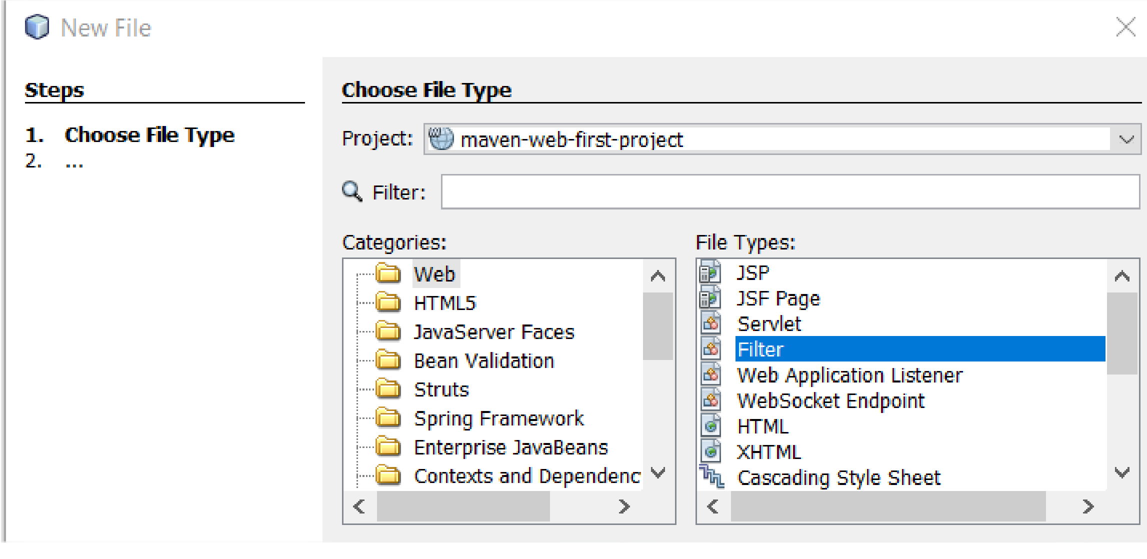
XIII) Notion de Filtre de Servlet.

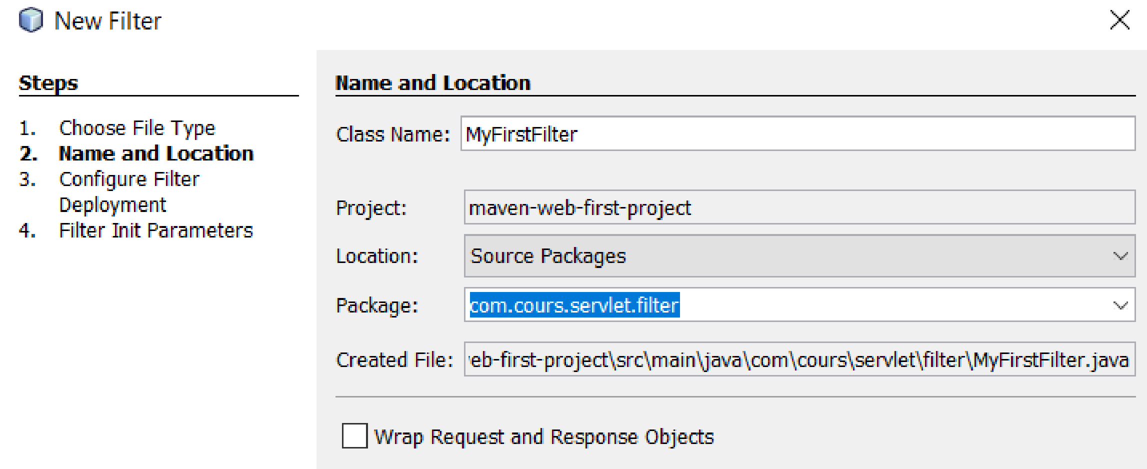
Un filtre HTTP de servlet est un composant d'une application Web qui agit comme un intercepteur sur une servlet. Un filtre est une classe Java qui implémente l'interface javax.servlet.Filter. Il est déclaré dans le descripteur de l'application **web.xml**, et posé sur une ou plusieurs servlets.
Lorsqu'une requête HTTP doit être traitée par une servlet sur laquelle un filtre est appliqué, alors le serveur va exécuter la méthode **doFilter** du Filtre avant d'exécuter la méthode **doGet** ou **doPost** de la servlet.
La méthode **doFilter** reçoit en paramètres :

- Un objet de type **javax.servlet.ServletRequest** qui est la requête qui auraient été envoyés à la servlet.
- Un objet de type **javax.servlet.ServletResponse** qui est la réponse qui auraient été envoyés à la servlet.
- Un objet de type **javax.servlet.FilterChain** qui modélise la servlet interceptée. Comme plusieurs filtres peuvent être déclarés en cascade, il se peut que cet objet modélise le filtre à invoquer après celui-ci.

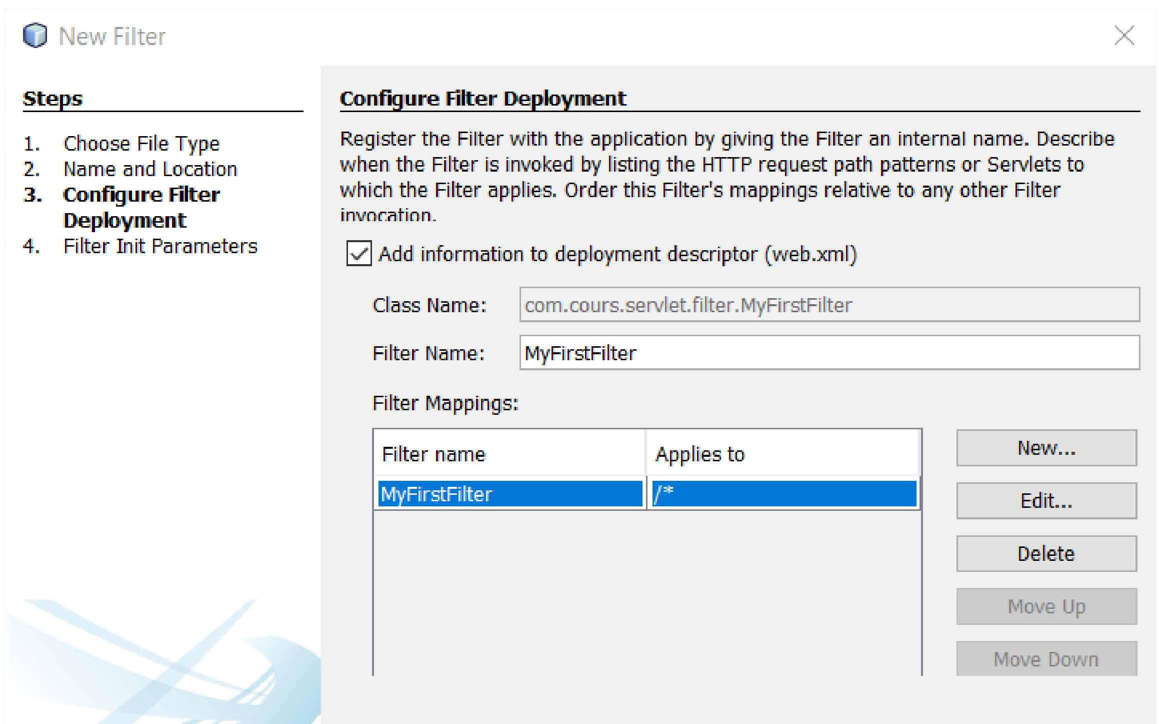
Le filtre peut donc agir sur la requête, il peut aussi autoriser le traitement de cette dernière par la servlet (il doit juste invoquer la méthode doFilter de l'objet **javax.servlet.FilterChain** en lui passant l'objet de type **javax.servlet.ServletRequest** et l'objet de type **javax.servlet.ServletResponse**.
Le filtre peut jouer le rôle de tampon de sécurité pour autoriser ou non une requête HTTP à attaquer une servlet ou pas.
Soit **com.cours.servlet.filter.MyFirstFilter** notre nouveau filtre HTTP que nous allons créer avec NetBeans.







Cocher **Add information to descriptor (web.xml)** puis **Finish**.



Le fichier web.xml devient :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    <filter>
        <filter-name>MyFirstFilter</filter-name>
        <filter-class>com.cours.servlet.filter.MyFirstFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>MyFirstFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <servlet>
        <servlet-name>MyFirstServlet</servlet-name>
        <servlet-class>com.cours.servlet.MyFirstServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>MyFirstServlet</servlet-name>
        <url-pattern>/MyFirstServlet</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>MyFirstServlet</welcome-file>
    </welcome-file-list>
</web-app>
```

En version copiable :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <filter>
    <filter-name>MyFirstFilter</filter-name>
    <filter-class>com.cours.servlet.filter.MyFirstFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>MyFirstFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <servlet>
    <servlet-name>MyFirstServlet</servlet-name>
    <servlet-class>com.cours.servlet.MyFirstServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyFirstServlet</servlet-name>
    <url-pattern>/MyFirstServlet</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>MyFirstServlet</welcome-file>
  </welcome-file-list>
</web-app>
```

Ligne 4 : Définition du nom de notre nouveau filtre.
Ligne 5 : nom complet de la classe du filtre.
Ligne 8 : nom du filtre appliqué aux servlets.
Ligne 9 : type d'urls appliquées au filtre (**/*** veut dire toutes les url) donc notre filtre va être appliqué pour tous les url.
Ligne 21 : durée de la session utilisateur, ici 30 minutes. Lors des différentes requêtes, un utilisateur est suivi par un jeton de session qui lui a été attribué à sa première requête et qu'il envoie ensuite systématiquement à chaque nouvelle requête. Cela permet au serveur web de le reconnaître et de gérer une " mémoire " pour l'utilisateur qu'on appelle la session. Si entre deux requêtes s'écoulent plus de 30 mn, un nouveau jeton de session est généré pour l'utilisateur qui perd ainsi sa " mémoire " et en recommence une nouvelle.

Regardons de plus prés la classe **MyFirstFilter** :

```
package com.cours.servlet.filter;

import java.io.IOException;
import java.io.PrintStream;
import java.io.PrintWriter;
import java.io.StringWriter;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

/**
 *
 * @author elhad
 */
public class MyFirstFilter implements Filter {

    private static final boolean debug = true;

    // The filter configuration object we are associated with.  If
    // this value is null, this filter instance is not currently
    // configured.
    private FilterConfig filterConfig = null;

    public MyFirstFilter() {
    }

    private void doBeforeProcessing(ServletRequest request, ServletResponse response)
            throws IOException, ServletException {
        if (debug) {
            log("MyFirstFilter:DoBeforeProcessing");
        }
```

```
}

private void doAfterProcessing(ServletRequest request, ServletResponse response)
        throws IOException, ServletException {
    if (debug) {
        log("MyFirstFilter:DoAfterProcessing");
    }
}

/**
 *
 * @param request The servlet request we are processing
 * @param response The servlet response we are creating
 * @param chain The filter chain we are processing
 *
 * @exception IOException if an input/output error occurs
 * @exception ServletException if a servlet error occurs
 */
public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain)
        throws IOException, ServletException {

    if (debug) {
        log("MyFirstFilter:doFilter()");
    }

    doBeforeProcessing(request, response);

    Throwable problem = null;
    try {
        chain.doFilter(request, response);
    } catch (Throwable t) {
    // If an exception is thrown somewhere down the filter chain,
        // we still want to execute our after processing, and then
        // rethrow the problem after that.
        problem = t;
        t.printStackTrace();
    }

    doAfterProcessing(request, response);

// If there was a problem, we want to rethrow it if it is
    // a known type, otherwise log it.
    if (problem != null) {
        if (problem instanceof ServletException) {
            throw (ServletException) problem;
        }
        if (problem instanceof IOException) {
            throw (IOException) problem;
```

```java
        }
        sendProcessingError(problem, response);
    }
}

/**
 * Return the filter configuration object for this filter.
 */
public FilterConfig getFilterConfig() {
    return (this.filterConfig);
}

/**
 * Set the filter configuration object for this filter.
 *
 * @param filterConfig The filter configuration object
 */
public void setFilterConfig(FilterConfig filterConfig) {
    this.filterConfig = filterConfig;
}

/**
 * Destroy method for this filter
 */
public void destroy() {
}

/**
 * Init method for this filter
 */
public void init(FilterConfig filterConfig) {
    this.filterConfig = filterConfig;
    if (filterConfig != null) {
        if (debug) {
            log("MyFirstFilter:Initializing filter");
        }
    }
}

/**
 * Return a String representation of this object.
 */
@Override
public String toString() {
    if (filterConfig == null) {
        return ("MyFirstFilter()");
    }
    StringBuffer sb = new StringBuffer("MyFirstFilter(");
    sb.append(filterConfig);
```

```
    sb.append(")");
    return (sb.toString());
}

private void sendProcessingError(Throwable t, ServletResponse response) {
    String stackTrace = getStackTrace(t);

    if (stackTrace != null && !stackTrace.equals("")) {
        try {
            response.setContentType("text/html");
            PrintStream ps = new PrintStream(response.getOutputStream());
            PrintWriter pw = new PrintWriter(ps);
            pw.print("<html>\n<head>\n<title>Error</title>\n</head>\n<body>\n"); //NOI18N

            // PENDING! Localize this for next official release
            pw.print("<h1>The resource did not process correctly</h1>\n<pre>\n");
            pw.print(stackTrace);
            pw.print("</pre></body>\n</html>"); //NOI18N
            pw.close();
            ps.close();
            response.getOutputStream().close();
        } catch (Exception ex) {
        }
    } else {
        try {
            PrintStream ps = new PrintStream(response.getOutputStream());
            t.printStackTrace(ps);
            ps.close();
            response.getOutputStream().close();
        } catch (Exception ex) {
        }
    }
}

public static String getStackTrace(Throwable t) {
    String stackTrace = null;
    try {
        StringWriter sw = new StringWriter();
        PrintWriter pw = new PrintWriter(sw);
        t.printStackTrace(pw);
        pw.close();
        sw.close();
        stackTrace = sw.getBuffer().toString();
    } catch (Exception ex) {
    }
    return stackTrace;
}

public void log(String msg) {
```

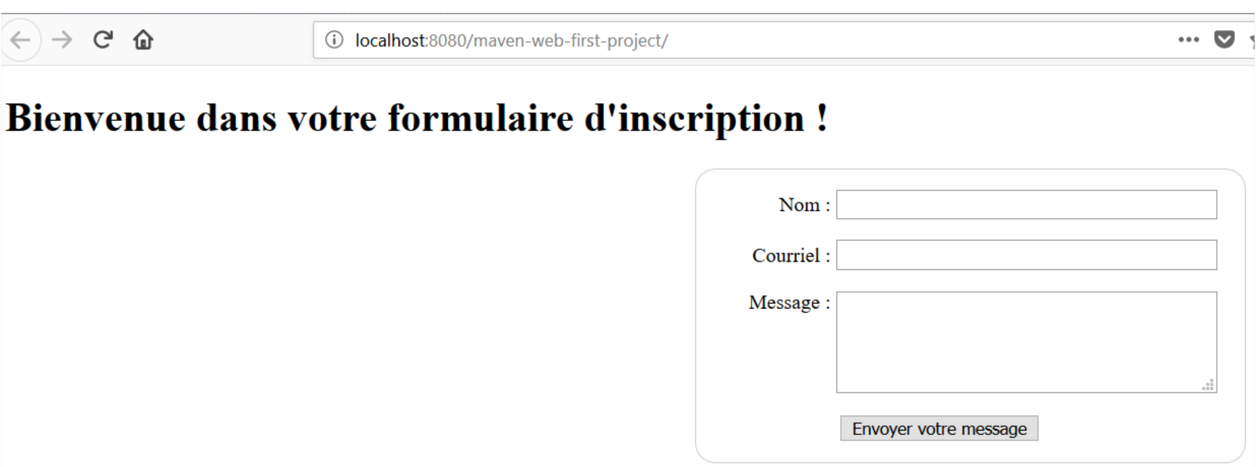
```
        filterConfig.getServletContext().log(msg);
    }

}
```

La méthode principale de notre filtre est sans conteste la méthode **doFilter** qui transmet la requête HTTP à la servlet **MyFirstServlet**. Les méthodes **doBeforeProcessing** et **doAfterProcessing** sont les méthodes qui sont exécutées respectivement avant et après « **chain.doFilter (request, response)** ».

La méthode **doGet** de la classe **MyFirstServlet** peut devenir :

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String methodName = "doGet";
    System.out.println("*************  Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + "
*************");
    this.getServletContext().getRequestDispatcher("/pages/newsletters/FormInscription.jsp").forward(request, response);
}
```

A l'exécution on obtient toujours :



En consultant les logs du serveur d'application GlassFish on obtient :

L'analyse des logs nous permet de voir que l'application appelle d'abord MyFirstFilter.doFilter, puis la méthode MyFirstFilter.DoBeforeProcessing puis la méthode Servlet MyFirstServlet.doGet et enfin MyFirstFilter.DoAfterProcessing.

Donc avec un filtre à Servlet il est possible :

- D'ajouter des informations dans la requête Http dans MyFirstFilter.DoBeforeProcessing et de les retrouver dans la servlet MyFirstFilter.
- D'enlever des information de la requête http résultant de la servlet MyFirstFilter dans la méthode MyFirstFilter:DoAfterProcessing.
- De bloquer l'éxécution de la servlet MyFirstFilter dans la méthode MyFirstFilter.DoBeforeProcessing.

Il est possible de passer des objets de type « **javax.servlet.ServletRequest** » et « **javax.servlet.ServletResponse** » aux objets conventionnels de type « javax.servlet.http.HttpServletRequest » et « javax.servlet.http.HttpServletResponse» avec :

```
HttpServletResponse httpResponse = (HttpServletResponse) response;
HttpServletRequest httpServletRequest = (HttpServletRequest) request;
```

XIV) Notion de Listener de Servlet.

Comme son nom le laisse supposer, un objet **listener** est un objet qui écoute certains événements dans une application. Lorsque l'événement qu'il écoute se déclenche, alors ce **listener** s'active : en général, une de ses méthodes particulière est invoquée, avec en paramètre un objet portant les informations sur cet événement.
Les objets de type **listeners** sont enregistrés auprès du serveur d'application. Ces objets doivent implémenter des interfaces standard, fournis par l'API Servlet, et être déclarés dans le descripteur d'une application web c'est-à-dire le fichier **web.xml**.

1. Les différents évènements écoutés dans l'API Servlet

Les événements définis par l'API Servlet sont au nombre de sept :

- Trois événements concernent l'ajout ou le retrait d'un attribut sur le contexte, la session ou une requête : **ServletContextEvent**, **HttpSessionEvent** et **ServletRequestEvent**.
- Deux événements sont associés à la création ou à la destruction du contexte, ou d'une requête : **ServletContextEvent** et **ServletRequestEvent**.
- Un événement est associé au cycle de vie d'une session : **HttpSessionEvent**. Cet événement est utilisé par deux ***listeners*** : **HttpSessionListener** et **HttpSessionActivationListener**. Ces deux *listeners* sont associés au cycle de vie particulier des sessions, que nous allons voir dans la suite.
- Un dernier événement permet de signaler à un objet qu'il a été ajouté à une session, ou qu'il va en être retiré : **HttpSessionBindingEvent**.

2. Evènements d'ajout ou retrait d'un attribut

Les trois interfaces que l'on peut implémenter sont ServletContextAttributeListener, ServletRequestAttributeListener et HttpSessionAttributeListener.

Ces trois interfaces exposent les trois mêmes méthodes :

- attributeAdded() : appelée lorsqu'un attribut est ajouté au contexte considéré ;
- attributeRemoved() : appelé lorsqu'un attribut est retiré du contexte considéré ;
- attributeReplaced() : appelé lorsqu'un attribut a été remplacé par un autre.

Ces trois méthodes *callback* sont appelées une fois que l'événement d'ajout, de retrait, ou de remplacement a été effectué.

Ces trois méthodes prennent des paramètres différents en fonction du contexte, qui sont les événements que nous avons vus. Ces objets exposent tous les trois les mêmes méthodes :

- getName() : le nom de l'attribut concerné ;
- getValue() : la valeur de l'attribut concerné.

L'objet HttpSessionBindingEvent, utilisé pour signaler à un objet qu'il a été ajouté ou retiré d'une session expose en plus une méthode getSession().

3. évènements de création et destruction d'un contexte

Chaque contexte possède son interface **listener**. On peut citer 3 types de contextes.

- Contexte de l'application

Le listener de création et destruction de l'application web doit implémenter l'interface ServletContextListener. Cette interface expose deux méthodes : contextInitialized() et contextDestroyed().

Ces deux méthodes sont invoquées avant toute invocation de filtre ou de servlet, et après l'appel à toutes les méthodes destroy() des filtres et servlets de cette application, respectivement.

Elles prennent en paramètre le même objet, de type ServletContextEvent. Cet objet expose une unique méthode : getServletContext(), qui retourne l'objet ServletContext. Cet objet modèlise l'application web proprement dite.

- Contexte de la requête

Le listener de ce contexte implémente l'interface ServletRequestListener. Elle expose deux méthodes : requestInitialized() et requestDestroyed().

Ces deux méthodes prennent en paramètre le même objet, instance de ServletRequestEvent. Cet objet permet d'accèder au contexte de l'application web par la méthode getServletContext(), et à l'objet requête par la méthode getServletRequest().

- Contexte de la session

La session a un cycle de vie un peu particulier, du fait de sa nature. Rappelons tout d'abord qu'une application web n'a pas nécessairement besoin de la notion de session. Une session devient nécessaire lorsque l'on a besoin de reconnaître un client donné d'une requête à l'autre.

Dans ce cas, une session est créée lors de la première requête de ce client. Cette session a une durée de vie, au-delà de laquelle elle est détruite. Cette durée de vie peut être plus ou moins longue : de quelques heures à plusieurs mois.

Si les requêtes d'une même session se succèdent rapidement, alors cette session sera probablement conservée en mémoire. Si le client reste plusieurs heures sans revenir sur le site, et que le site choisit de conserver tout de même sa session, celle-ci sera probablement déchargée de la mémoire, et conservée sur un support persistant, probablement une base de données.

On voit que quatre types d'événements peuvent alors exister pour une session :

- La création : elle intervient lors de la première requête.
- La destruction : elle peut ne jamais arriver, ou au bout d'un temps très long.
- La passivation : c'est ce qui se passe lorsqu'une session est déplacée de la mémoire vers un espace de stockage persistant. La session n'est pas détruite, elle passe dans un état de type "inactif".

- L'activation : ce qui se passe lorsqu'un client revient après une longue période d'absence. Sa session est rechargée de l'espace de stockage persistant vers la mémoire.

Notons que la passivation d'une session peut intervenir en environnement clusterisé, lorsqu'une session doit passer d'un nœud du cluster à un autre.

Ces quatre événements sont gérés par deux interfaces.

- HttpSessionListener : expose les méthodes sessionCreated() et sessionDestroyed(), qui prennent en paramètre un objet HttpSessionEvent.
- HttpSessionActivationListener : expose les méthodes sessionDidActivate() et sessionWillPassivate(). Ces deux méthodes prennent en paramètre le même objet HttpSessionEvent.

L'objet HttpSessionEvent n'expose qu'une unique méthode : getSession(), qui retourne la session concernée.

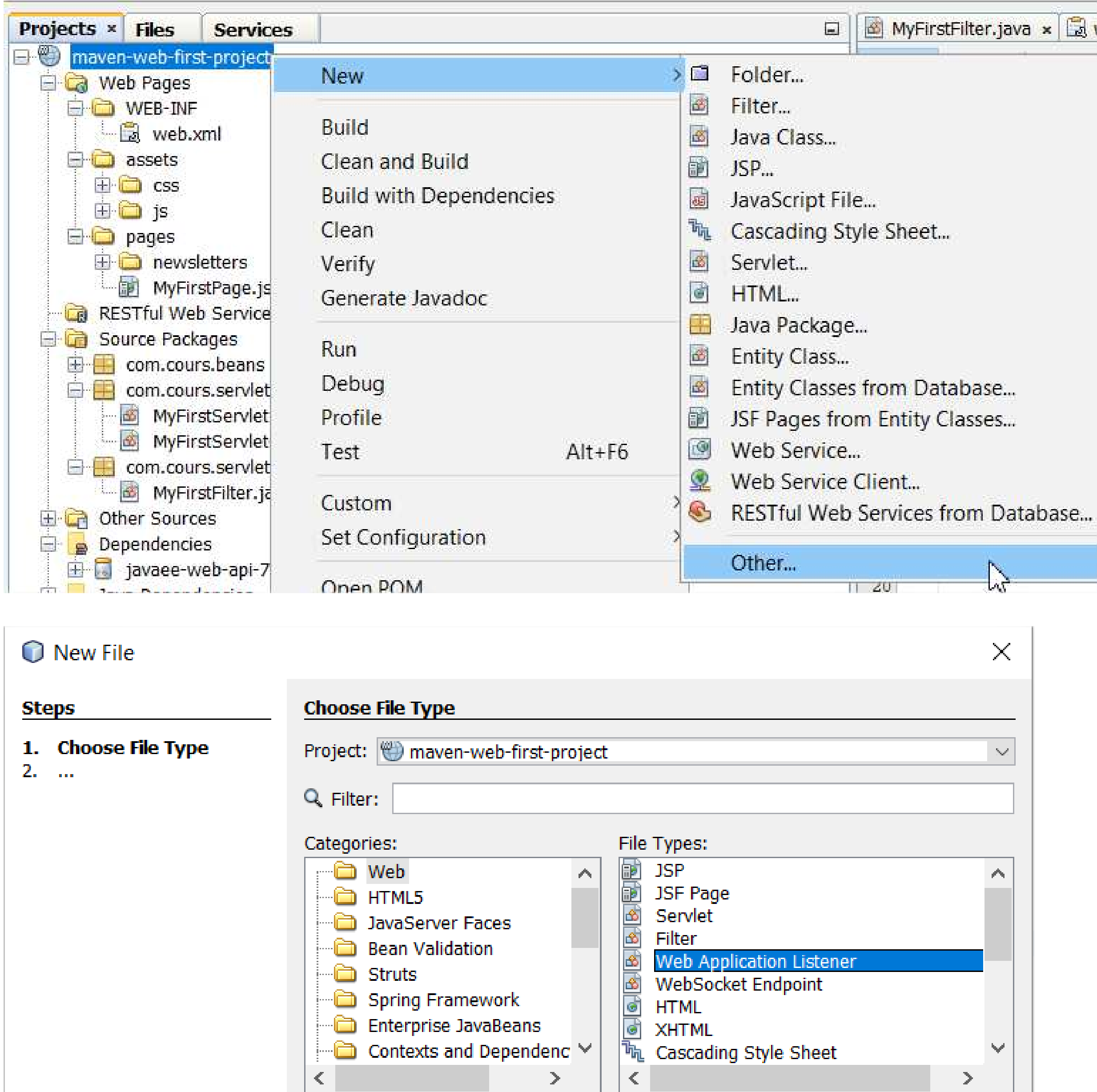
4. évènements de notification d'attache d'un objet à un contexte

Enfin, un objet peut être notifié du fait qu'il a été ajouté à une session. Il doit pour cela implémenter l'interface HttpSessionBindingListener. Cette interface expose deux méthodes :

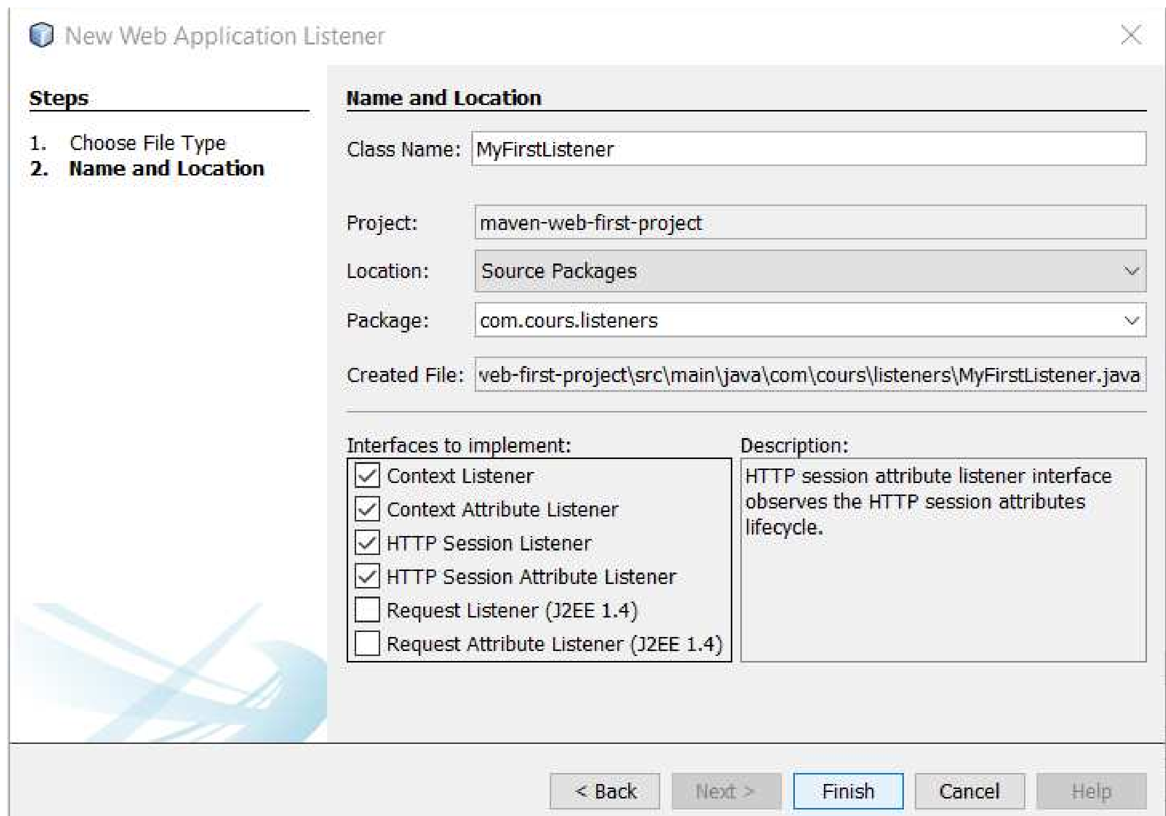
- valueBound(HttpSessionBindingEvent) : appelée lorsque l'objet est attaché à une session ;
- valueUnbound(HttpSessionBindingEvent) : appelée lorsque l'objet est détaché de la session.

Ces deux méthodes reçoivent en paramètre un objet de type HttpSessionBindingEvent. Cet événement expose les deux méthodes classiques getName() et getValue(). Il expose également getSession(), qui retourne la session à laquelle cet objet est attaché, ou de laquelle il est détaché.

Nous allons voir ces aspects théoriques en pratique à travers la création de notre premier listener **com.cours.listeners.MyFirstListener**.



Cocher par exemple **Context Listener**, **Context Attribute Listener**, **HTTP Session Listener** et **HTTP Session Attribute Listener** puis **Finish**.



Le fichier **web.xml** devient :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/j
    <filter>
        <filter-name>MyFirstFilter</filter-name>
        <filter-class>com.cours.servlet.filter.MyFirstFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>MyFirstFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <listener>
        <description>ServletContextListener, ServletContextAttributeListener, HttpSessionListener, HttpSessionAttributeListener</description>
        <listener-class>com.cours.listeners.MyFirstListener</listener-class>
    </listener>
    <servlet>
        <servlet-name>MyFirstServlet</servlet-name>
        <servlet-class>com.cours.servlet.MyFirstServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>MyFirstServlet</servlet-name>
        <url-pattern>/MyFirstServlet</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>MyFirstServlet</welcome-file>
    </welcome-file-list>
</web-app>
```

En version copiable :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <filter>
    <filter-name>MyFirstFilter</filter-name>
    <filter-class>com.cours.servlet.filter.MyFirstFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>MyFirstFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <listener>
    <description>ServletContextListener, ServletContextAttributeListener, HttpSessionListener, HttpSessionAttributeListener</description>
    <listener-class>com.cours.listeners.MyFirstListener</listener-class>
  </listener>
  <servlet>
    <servlet-name>MyFirstServlet</servlet-name>
    <servlet-class>com.cours.servlet.MyFirstServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyFirstServlet</servlet-name>
    <url-pattern>/MyFirstServlet</url-pattern>
  </servlet-mapping>
  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>MyFirstServlet</welcome-file>
  </welcome-file-list>
</web-app>
```

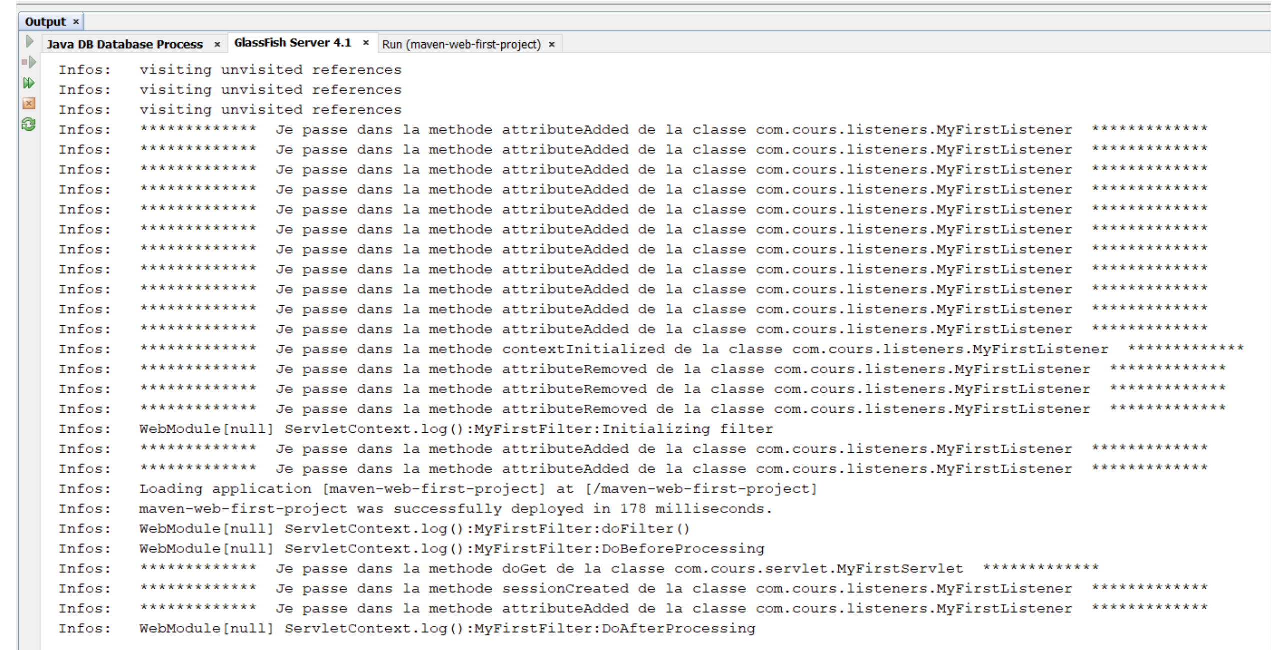
Ligne 11 : définition du nouveau listener.
Ligne 12 : définition des types de listener en l'occurrence ici **ServletContextListener**, **ServletContextAttributeListener**, **HttpSessionListener**, **HttpSessionAttributeListener** qui sont respectivement les listener de contexte, d'attribut de contexte,de sessions et d'attribut de sessions.
Ligne 13 : définition de la classe d'implémentation du listener qui est ici **MyFirstListener**.

La classe **MyFirstListener** aura comme contenu :

```
package com.cours.listeners;

import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class MyFirstListener implements ServletContextListener, ServletContextAttributeListener, HttpSessionListener,
HttpSessionAttributeListener {

  @Override
  public void contextInitialized(ServletContextEvent sce) {
    String methodName = "contextInitialized";
    System.out.println("************* Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + "
*************");
  }

  @Override
  public void contextDestroyed(ServletContextEvent sce) {
    String methodName = "contextDestroyed";
    System.out.println("************* Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + "
*************");
  }

  @Override
  public void attributeAdded(ServletContextAttributeEvent event) {
    String methodName = "attributeAdded";
    System.out.println("************* Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + "
*************");
  }

  @Override
  public void attributeRemoved(ServletContextAttributeEvent event) {
    String methodName = "attributeRemoved";
    System.out.println("************* Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + "
*************");
  }

  @Override
  public void attributeReplaced(ServletContextAttributeEvent arg0) {
    String methodName = "attributeReplaced";
    System.out.println("************* Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + "
*************");
```

```
    }

    @Override
    public void sessionCreated(HttpSessionEvent se) {
        String methodName = "sessionCreated";
        System.out.println("************* Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + " *************");
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        String methodName = "sessionDestroyed";
        System.out.println("************* Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + " *************");
    }

    @Override
    public void attributeAdded(HttpSessionBindingEvent event) {
        String methodName = "attributeAdded";
        System.out.println("************* Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + " *************");
    }

    @Override
    public void attributeRemoved(HttpSessionBindingEvent event) {
        String methodName = "attributeRemoved";
        System.out.println("************* Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + " *************");
    }

    @Override
    public void attributeReplaced(HttpSessionBindingEvent event) {
        String methodName = "attributeReplaced";
        System.out.println("************* Je passe dans la methode " + methodName + " de la classe " + this.getClass().getName() + " *************");
    }
}
```

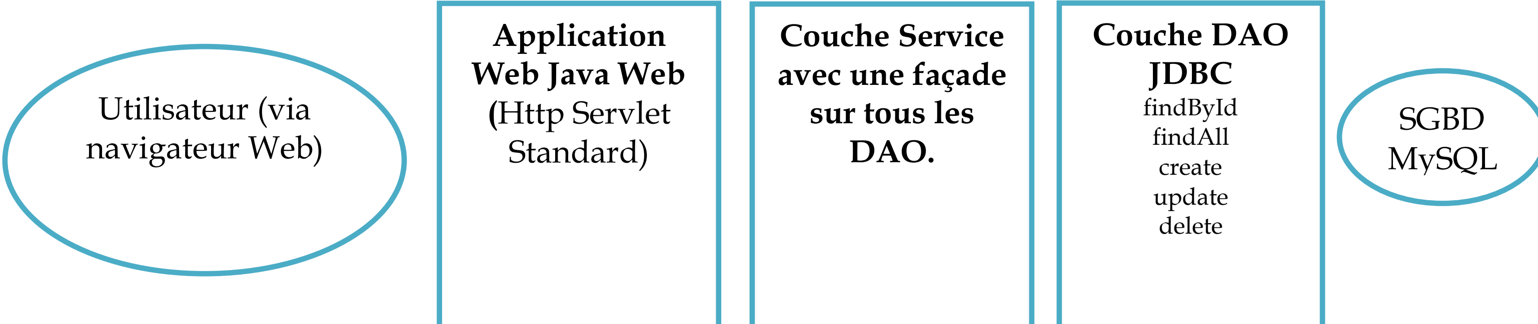
Faire un **Clean And Build** puis faire un **Clear** de la console de GlassFish puis un **Run** et vous aurez les logs ci-dessous qui montre bien les passages dans notre listener **MyFirstListener** suivant les différentes situations (Ajout d'un nouveau attribut dans l'application, Ajout d'une nouvelle session attribut dans l'application ect….).



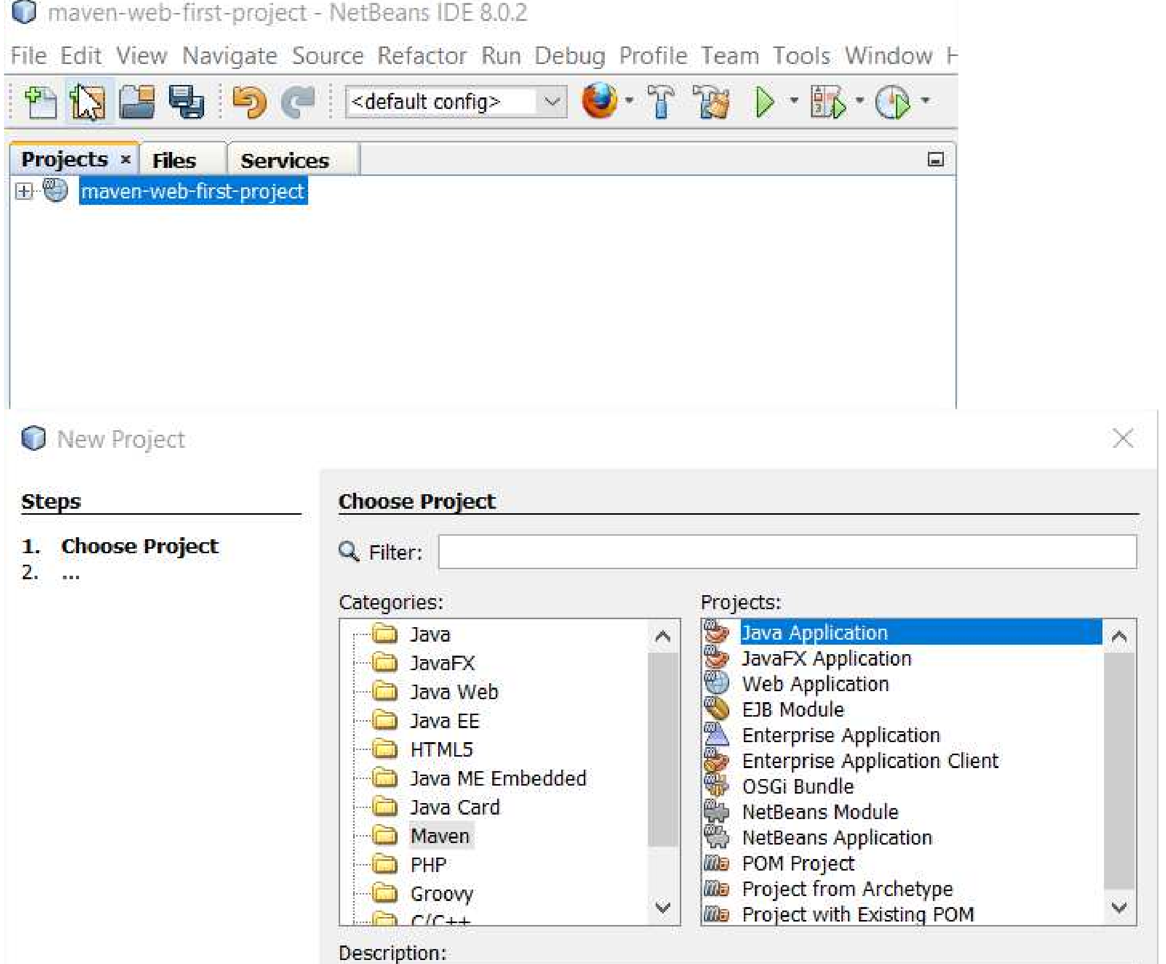
Output
Java DB Database Process
GlassFish Server 4.1
Run (maven-web-first-project)
Infos: visiting unvisited references
Infos: visiting unvisited references
Infos: visiting unvisited references
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode contextInitialized de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeRemoved de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeRemoved de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeRemoved de la classe com.cours.listeners.MyFirstListener *************
Infos: WebModule[null] ServletContext.log():MyFirstFilter:Initializing filter
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: Loading application [maven-web-first-project] at [/maven-web-first-project]
Infos: maven-web-first-project was successfully deployed in 178 milliseconds.
Infos: WebModule[null] ServletContext.log():MyFirstFilter:doFilter()
Infos: WebModule[null] ServletContext.log():MyFirstFilter:DoBeforeProcessing
Infos: ************* Je passe dans la methode doGet de la classe com.cours.servlet.MyFirstServlet *************
Infos: ************* Je passe dans la methode sessionCreated de la classe com.cours.listeners.MyFirstListener *************
Infos: ************* Je passe dans la methode attributeAdded de la classe com.cours.listeners.MyFirstListener *************
Infos: WebModule[null] ServletContext.log():MyFirstFilter:DoAfterProcessing

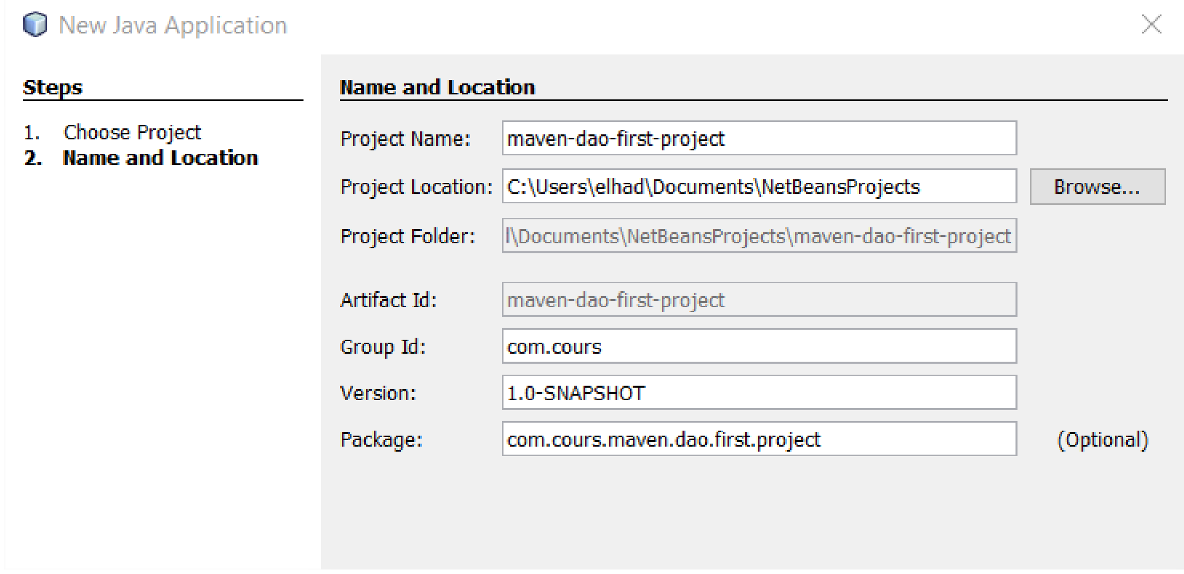
XV) Application 3 couches en Java Web.

Nous allons voir dans cette partie du cours comment interconnecter une application Java de type 3 couches.



Soit l'application Java **maven-dao-first-project** qui contient le code notre DAO.
Soit l'application Java **maven-service-first-project** qui contient le code notre Service (la façade).
Soit l'application Java **maven-web-first-project** qui contient le code notre application Web.
Créer le projet **maven-dao-first-project**.

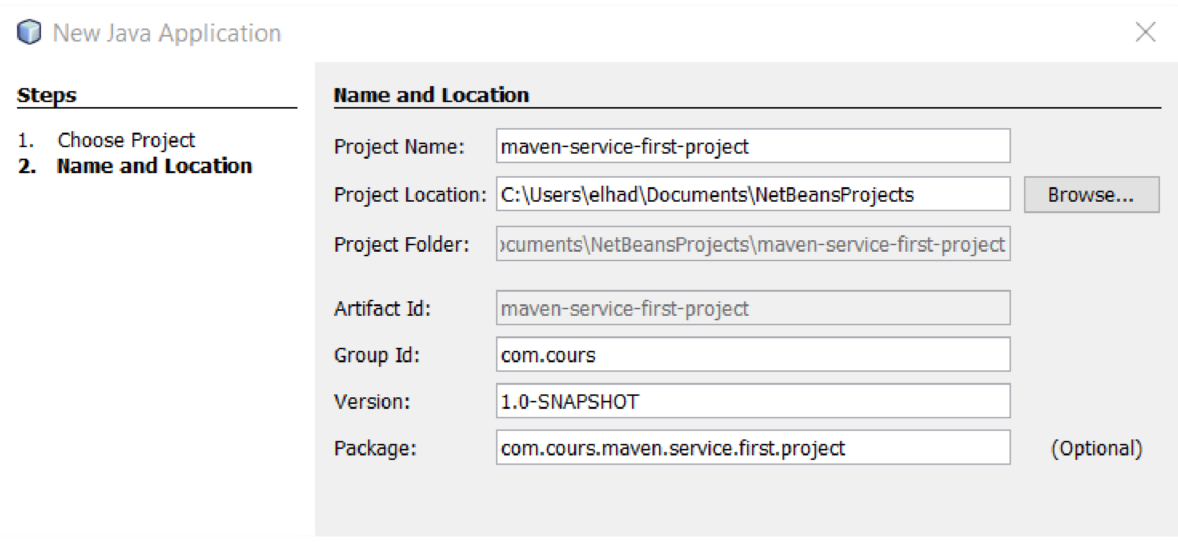




Le fichier **pom.xml** du projet **maven-dao-first-project** est par défaut :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.cours</groupId>
    <artifactId>maven-dao-first-project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.7</maven.compiler.source>
        <maven.compiler.target>1.7</maven.compiler.target>
    </properties>
</project>
```

Créer le projet **maven-service-first-project**.



Le fichier **pom.xml** du projet **maven-service-first-project** est par défaut :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.cours</groupId>
    <artifactId>maven-service-first-project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.7</maven.compiler.source>
        <maven.compiler.target>1.7</maven.compiler.target>
    </properties>
</project>
```

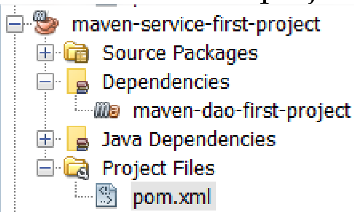
Récupérer le projet **maven-web-first-project** que vous avez utilisé tout au long de ce cours.

Si on veut inclure le projet le projet **maven-dao-first-project** dans le projet **maven-service-first-project**, il suffit d'inclure le projet **maven-dao-first-project** dans le pom.xml de **maven-service-first-project.**

Le fichier **pom.xml** du projet **maven-service-first-project** devient :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.cours</groupId>
   <artifactId>maven-service-first-project</artifactId>
   <version>1.0-SNAPSHOT</version>
   <packaging>jar</packaging>
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <maven.compiler.source>1.7</maven.compiler.source>
      <maven.compiler.target>1.7</maven.compiler.target>
   </properties>
   <dependencies>
      <dependency>
         <groupId>com.cours</groupId>
         <artifactId>maven-dao-first-project</artifactId>
         <version>1.0-SNAPSHOT</version>
      </dependency>
   </dependencies>
</project>
```

En faisant un **Clean And Build** du projet **maven-dao-first-project** puis **maven-service-first-project** on obtient dans le projet **maven-service-first-project** :



On voit bien que dans le projet **maven-dao-first-project** il s'est rajouté dans les dépendances du projet **maven-service-first-project**.

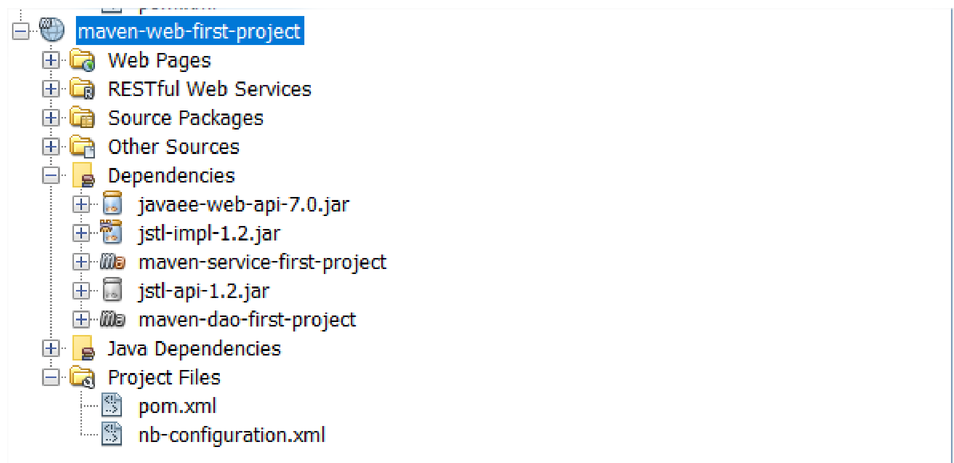
Toujours dans la même logique si on veut ajouter le projet **maven-service-first-project** dans le projet **maven-web-first-project**, le fichier pom.xml de **maven-web-first-project** devient :

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.cours</groupId>
  <artifactId>maven-web-first-project</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>maven-web-first-project</name>

  <properties>
    <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>com.cours</groupId>
      <artifactId>maven-service-first-project</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>javax</groupId>
      <artifactId>javaee-web-api</artifactId>
      <version>7.0</version>
      <scope>provided</scope>
    </dependency>
    <!-- Pour les utilisateur du serveur d'application Tomcat ajouter la dependance org.glassfish.web -->
    <dependency>
      <groupId>org.glassfish.web</groupId>
      <artifactId>jstl-impl</artifactId>
      <version>1.2</version>
      <exclusions>
        <exclusion>
          <artifactId>servlet-api</artifactId>
          <groupId>javax.servlet</groupId>
        </exclusion>
        <exclusion>
          <artifactId>jsp-api</artifactId>
          <groupId>javax.servlet.jsp</groupId>
        </exclusion>
      </exclusions>
    </dependency>
```

```xml
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
        <compilerArguments>
          <endorseddirs>${endorsed.dir}</endorseddirs>
        </compilerArguments>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <version>2.3</version>
      <configuration>
        <failOnMissingWebXml>false</failOnMissingWebXml>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>2.6</version>
      <executions>
        <execution>
          <phase>validate</phase>
          <goals>
            <goal>copy</goal>
          </goals>
          <configuration>
            <outputDirectory>${endorsed.dir}</outputDirectory>
            <silent>true</silent>
            <artifactItems>
              <artifactItem>
                <groupId>javax</groupId>
                <artifactId>javaee-endorsed-api</artifactId>
                <version>7.0</version>
                <type>jar</type>
              </artifactItem>
            </artifactItems>
          </configuration>
        </execution>
      </executions>
    </plugin>
```

```
    </plugins>
  </build>
</project>
```

En faisant un **Clean And Build** du projet **maven-dao-first-project** puis **maven-service-first-project** et enfin **maven-web-first-project** (en faisant l'UnDeploy avant) on obtient dans le projet **maven-web-first-project** :



On voit bien que dans le projet **maven-web-first-project** il s'est rajouté dans les dépendances des projets **maven-dao-first-project** et **maven-service-first-project**.

XVI) Exercice d'Application sur l'implémentation 3 couches.

Dans cette partie du cours nous allons réaliser l'application **3 couches** suivante :

Utilisateur (via navigateur Web)	**Application Web Java Web (**Http Servlet Standard) **maven-web-first-project**	**Couche Service avec une façade sur tous les DAO.** **maven-service-first-project**	**Couche DAO JDBC** findById,findAll,create,update,delete **maven-dao-first-project**	SGBD MySQL

Si vous avez déjà fait l'exercice 3 (**DAO**) du PDF **Exercices-Design-Pattern.pdf** des design Pattern il suffit de récupérer tout le code de votre DAO pour la partie **SQL** et le copier dans le projet **maven-dao-first-project** précédemment créé.
Si vous n'avez pas fait l'exercice voici ci-dessous une petite adaptation de l'exercice pour vous permettre d'avancer rapidement.

1) Créer dans le projet **maven-dao-first-project** avec la classe **com.cours.entities.Personne** avec les attributs suivants :

« idPersonne» de type « int », « prenom » de type « String », « nom » de type « String », « poids» de type « double », « taille » de type « double », « rue» de type « String », « ville» de type « String », « codePostal» de type « String ».

Méthodes :
« toString () » qui retourne tous les informations sur les attributs de la classe « Personne ».
« equals () » qui retourne un boolean.
« hashCode () » qui retourne un entier.
« getImc () » qui retourne un double qui représente l'indice de masse corporelle.
« isMaigre () » qui retourne un boolean représentant l'état de maigreur de la personne.
« isSurPoids () » qui retourne un boolean représentant l'état de surpoids de la personne.
« isObese () » qui retourne un boolean représentant l'état d'obésité de la personne.

Indications :

Vous trouverez sur le lien ci-dessous toutes les informations pour les calculs des IMC :

https://fr.wikipedia.org/wiki/Indice_de_masse_corporelle

2) Créer la classe **com.cours.dao.PersonneDao** et **com.cours.dao.IPersonneDao** son interface dans **maven-dao-first-project** avec :

- Un attribut de type java.sql.Connection Connection connection qui gèrera la connexion à la base de données (Cf. fichier base_personnes.sql).

- Les méthodes : findAll,findById,create,update,delete.

```
public List<Personne> findAll(),
public Personne findById(int id),
public Personne create(Personne person),
public Personne update(Personne person),
public boolean delete(Personne person)
```

Vous pourrez ajouter dans votre fichier **pom.xml** la dépendance suivante :

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version> 5.1.41</version>
</dependency>
```

3) Créer la classe **com.cours.service.ServiceFacade** et **com.cours.service.IServiceFacade** son interface dans **maven-service-first-project.**

4) Créer une interface web avec le projet **maven-web-first-project** dans lequel vous allez afficher un tableau HTML avec comme colonnes Prénom,Nom,Poids ect….

Vous ajouterez aussi les boutons Ajouter/Modifier/Supprimer pour ajouter une personne, modifier une personne et supprimer une personne.

5) Créer une interface web avec le projet **maven-web-first-project** dans lequel vous allez afficher un tableau HTML avec comme colonnes Prénom,Nom,Poids ect….

Vous ajouterez aussi les boutons Ajouter/Modifier/Supprimer pour ajouter une personne, modifier une personne et supprimer une personne.