Introduction à Maven 2

Date de publication : 27/01/2006

Par John Ferguson Smart (The Wakaleo Blog)
Traduction de Denis Cabasson (Developpez.com)

Est-ce que vous passez trop de temps à maintenir des scripts de build complexes pour vos projets Java ? Avez-vous à réinventer ou à réapprendre un nouvel ensemble de cible de build pour chaque nouveau projet ? Est-ce que vous finissez avec un grand nombre de dépendances inutiles et ne savez jamais réellement quels sont les JARs dont vous avez besoin ? Est-ce qu'un site interne à votre projet, généré automatiquement et toujours à jour vous intéresse ? Si c'est le cas, Maven 2 pourrait être la réponse à vos problèmes. La dernière version de cet outil populaire de build open-source est une réécriture complète de la version 1.x qui ajoute beaucoup de nouveautés très puissantes, comme la gestion des dépendances transitives, un cycle de vie bien défini, des builds personnalisés plus simples qu'en utilisant des tâches Ant, et une meilleure génération de site interne. N'importe quel nouveau projet Java a beaucoup à gagner à utiliser Maven 2.

Maven est un outil open-source de build pour les projets Java très populaire, conçu pour supprimer les tâches difficiles du processus de build. Maven utilise une approche déclarative, où le contenu et la structure du projet sont décrits, plutôt qu'une approche par tâche utilisée par exemple par Ant ou les fichiers make traditionnels. Cela aide à mettre en place des standards de développements au niveau d'une société et réduit le temps nécessaire pour écrire et maintenir les scripts de build.

L'approche déclarative, basée sur le cycle de vie du projet et utilisée par Maven 1, est pour beaucoup un changement radical par rapport aux techniques de build traditionnelles, et Maven 2 va encore plus loin dans cette optique. Dans cet article, je m'intéresse aux principes basiques derrières Maven 2 puis passe à un exemple concret. Commençons par voir les fondamentaux de Maven 2.

1. Le modèle objet projet

Le coeur d'un projet Maven 2 est le modèle objet projet (appelé POM pour project object

model). Il contient une description détaillée de votre projet, avec en particulier des informations concernant le versionnage et la gestion des configurations, les dépendances, les ressources de l'application, les tests, les membres de l'équipe, la structure et bien plus encore. Le POM prend la forme d'un fichier XML (*pom.xml*) qui est placé dans le répertoire de base de votre projet. Un fichier pom.xml est présenté ci-dessous :

```
<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/maven-v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.javaworld.hotels</groupId>
   <artifactId>HotelDatabase</artifactId>
   <packaging>war</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>Base de données des hotels</name>
   <url>http://www.hoteldatabase.com</url>
   <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>3.8.1</version>
         <scope>test</scope>
      </dependency>
   </dependencies>
</project>
```

2. La structure de répertoires Maven 2

Une partie de la puissance de Maven vient des pratiques standardisées qu'il encourage. Un développeur qui a déjà travaillé sur un projet Maven se sentira tout de suite familier avec la structure et l'organisation d'un autre projet Maven. Il n'y a pas besoin de gaspiller du temps à réinventer des structures de répertoires, des conventions, et à adapter des scripts de build Ant pour chaque projet. Même s'il est possible de redéfinir l'emplacement de chacun des répertoires vers une localisation spécifique, il est réellement intéressant de respecter la structure de répertoire standard de Maven 2 autant que possible, et ce pour plusieurs raisons :

- Cela rend le fichier POM plus court et plus simple
- Cela rend le projet plus simple à comprendre et rend la vie plus simple au pauvre développeur qui devra maintenir le projet quand vous partirez.
- Cela rend l'intégration de plug-ins plus simple

La structure standard de répertoire de Maven 2 est illustrée dans la figure 1. Dans le répertoire de base se trouve le POM (pom.xml) et deux sous répertoires : src pour tout le code source et target pour les éléments générés.

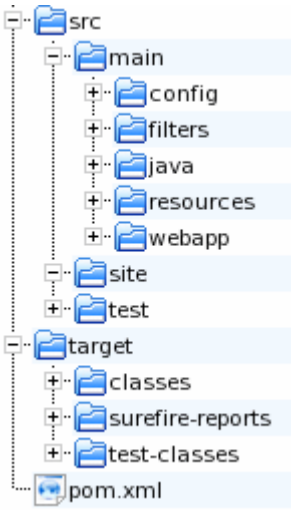

Figure 1. Le modèle de répertoire standard de Maven 2

Le répertoire src contient plusieurs sous-repertoires, chacun avec une utilité précise :

- **src/main/java:** Votre code java va ici (étonnamment)
- **src/main/resources:** Les autres ressources dont votre application a besoin
- **src/main/filters:** Les filtres de ressources, sous forme de fichier de propriétés, qui peuvent être utilisés pour définir des variables connues uniquement au moment du build.
- **src/main/config:** Les fichiers de configuration
- **src/main/webapp:** Le répertoire d'application web pour les projets WAR.
- **src/test/java:** Les tests unitaires
- **src/test/resources:** Les ressources nécessaires aux tests unitaires, qui ne seront pas déployées
- **src/test/filters:** Les filtres nécessaires aux tests unitaires, qui ne seront pas déployées
- **src/site**: Les fichiers utilisés pour générer le site web du projet Maven

3. Le cycle de vie du projet

Les cycles de vie des projets sont un concept central de Maven 2. La plupart des développeurs sont familiers avec les notions de phases du build comme compile, test et deploy. Ant a des cibles (targets) avec des noms semblables. Dans Maven 1, les plug-ins correspondant sont appelés directement. Pour compiler le code source java, par exemple, le plug-in *java* est utilisé :

```
$maven java:compile
```

Dans Maven 2, cette notion est standardisée dans un groupe de phases de cycle de vie bien définies et déterminées (voir figure 2). Au lieu de lancer des plug-ins, dans Maven 2, le développeur lance l'action liée à une phase du cycle de vie :

```
$mvn compile
```

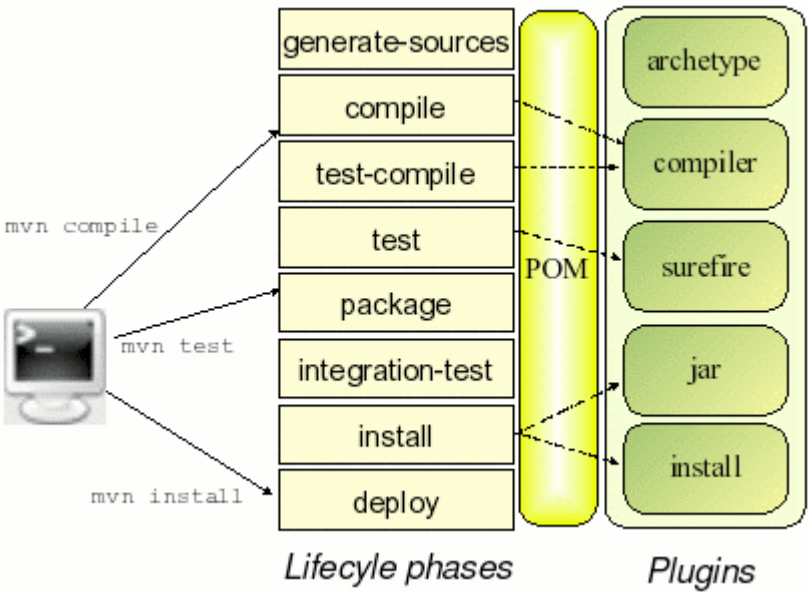

Figure 2. Les phases du cycle de vie Maven 2

Voilà quelques-unes unes des phases les plus utiles du cycle de vie Maven 2 :

- *generate-sources*: Génère le code source supplémentaire nécessité par l'application, ce qui est généralement accompli par les plug-ins appropriés.
- *compile*: Compile le code source du projet
- *test-compile*: Compile les tests unitaires du projet
- *test*: Exécute les tests unitaires (typiquement avec Junit) dans le répertoire src/test
- *package*: Mets en forme le code compilé dans son format de diffusion (JAR, WAR, etc.)
- *integration-test*: Réalise et déploie le package si nécessaire dans un environnement dans lequel les tests d'intégration peuvent être effectués.
- *install*: Installe les produits dans l'entrepôt local, pour être utilisé comme dépendance des autres projets sur votre machine locale.
- *deploy*: Réalisé dans un environnement d'intégration ou de production, copie le produit final dans un entrepôt distant pour être partagé avec d'autres développeurs ou projets.

Beaucoup d'autres phases du cycle de vie sont disponibles. Voir les références pour plus de détails.

Ces phases illustrent le bénéfice des pratiques recommandées, encouragées par Maven 2 : une fois que le développeur est familier avec les phases principales, il devrait se sentir à l'aise avec les phases du cycle de vie de n'importe quel projet Maven.

Les phases du cycle de vie invoquent les plug-ins nécessaires pour faire le travail. Appeler une phase du cycle de vie appelle automatiquement les phases précédentes du cycle de vie. Comme les phases du cycle de vie sont en nombre limité, faciles à comprendre et bien organisées, se familiariser avec le cycle de vie d'un nouveau projet Maven 2 est très

simple.

4. Dépendances transitives

Une des nouveautés de Maven 2 est la gestion des dépendances transitives. Si vous avez déjà utilisé un outil comme urpmi sur une Linux box, vous devez savoir ce que dépendance transitive veut dire. Avec Maven 1, vous aviez à déclarer chacun des JAR dont avait besoin, directement ou indirectement, votre application. Par exemple, pouvez vous lister les JARs nécessaires à une application Hibernate ? Avec Maven 2, vous n'avez pas besoin. Vous dites juste à Maven les bibliothèques dont vous avez besoin, et Maven se charge des bibliothèques dont vos bibliothèques ont besoin (et ainsi de suite).

Supposons que vous voulez utiliser Hibernate dans votre projet. Vous auriez simplement à ajouter une nouvelle dépendance à la section dependencies de votre pom.xml, comme suit :

```
<dependency>
   <groupId>hibernate</groupId>
   <artifactId>hibernate</artifactId>
   <version>3.0.3</version>
   <scope>compile</scope>
</dependency>
```

C'est tout! Vous n'avez pas besoin de courir après les autres JARs (et dans quelle version) dont a besoin Hibernate 3.0.3 pour fonctionner : Maven le fait pour vous !

La structure XML des dépendances dans Maven 2 est proche de celle utilisée dans Maven 1. La différence principale est l'attribut scope qui est expliqué dans la section suivante.

5. Portée des dépendances

Dans une application d'entreprise du monde réel, vous n'avez pas nécessairement besoin d'inclure toutes les dépendances dans l'application déployée. Certains des JARs sont nécessaires uniquement pour les tests unitaires, alors que d'autres seront fournis à l'exécution par le serveur d'application. En utilisant la technique de *la portée de dépendances*, Maven 2 vous permet d'utiliser certain JAR uniquement quand vous en avez besoin et de les exclure du classpath quand vous n'en avez pas besoin.

Maven mets à disposition quatre portés de dépendances :

- *compile*: Une dépendance de portée compile est disponible dans toutes les phases. C'est la valeur par défaut.
- *provided*: Une dépendance de portée provided est utilisée pour compiler l'application, mais ne sera pas déployée. Vous utiliserez cette portée quand vous

attendez du JDK ou du serveur d'application qu'il vous mette le JAR à disposition. L'API servlet est un bon exemple.
- *runtime*: Les dépendances de portées runtime ne sont pas nécessaires pour la compilation, uniquement pour l'exécution, comme les drivers JDBC (Java Database Connectivity).
- *test*: Les dépendances de portées test sont uniquement nécessaires pour compiler et exécuter les tests (par exemple Junit).

6. Communication de projet

Une partie importante de tout projet est la communication interne. Sans être une solution parfaite, un site web technique et centralisé permet de grandes avancées dans la vision commune d'une équipe. Avec un effort minimal, vous pouvez mettre en place un site web de qualité professionnelle en très peu de temps.

Cela prend encore une autre dimension quand la génération du site Maven est intégrée dans le processus de build en utilisant l'intégration continue ou les builds nocturnes automatisés. Un site Maven peut publier, sur une base quotidienne :

- Des informations générales sur le projet, comme les entrepôts de source, le suivi des anomalies, les membres de l'équipe, etc.
- Les tests unitaires et les rapports de couverture des tests unitaires
- Des revues de code automatisées avec Checkstyle ou PMD
- Des informations sur la configuration ou le versionnage
- Les dépendances
- La JavaDoc
- Le code source indexé et référençable, sous un format HTML
- La liste des membres de l'équipe
- Et beaucoup plus encore

Encore une fois, n'importe quel développeur calé sur Maven saura immédiatement ou regarder pour se familiariser avec un projet Maven 2.

7. Un exemple concret

Maintenant que nous avons vu quelques-unes unes des notions de base utilisées dans Maven 2, regardons comment cela se passe dans le monde réel. La fin de cet article examine comment nous pourrions utiliser Maven 2 pour un projet J2EE simple. L'application de demo modélise un système de base de données d'hôtels (simplifié). Pour illustrer comment Maven 2 traite les dépendances entre projets et composants, cette application sera construite autour de deux composants (voir figure 3) :

- Un composant métier : HotelDatabase.jar

- Un composant application web: HotelWebApp.war

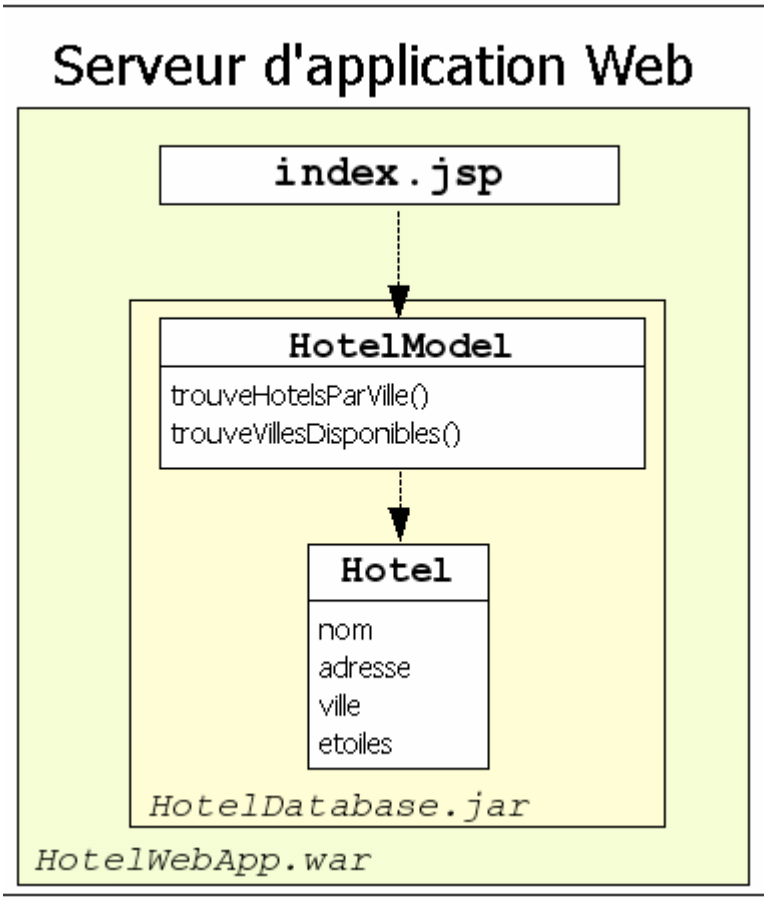

Figure 3. L'architecture de l'exemple d'application implique deux composants simples un JAR (HotelDatabase.jar) et un WAR (HotelWebapp.war)
Vous pouvez télécharger le code source pour suivre cet exemple dans les ressources.

8. Mise en place de l'environnement de projet

Nous commençons par configurer votre environnement de travail. Dans les projets réels, vous aurez souvent à définir et à configurer un environnement ou des paramètres spécifiques à un utilisateur, qui ne devront pas être communiqués à tous les utilisateurs. Si vous êtes derrière un proxy par exemple, vous aurez besoin de mettre en place la configuration de votre proxy, afin que Maven puisse télécharger des JARs issus d'entrepôts sur le web. Pour les utilisateurs de Maven 1, les fichiers build.properties et project.properties remplissaient ce travail. Avec Maven 2, ils ont été remplacés par un fichier settings.xml qui va dans le répertoire $HOME/.m2 (ndt : maintenant $MVN_HOME/conf). Voici un exemple :

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>
  <proxies>
    <proxy>
      <active/>
```

```
      <protocol>http</protocol>
      <username>dupont</username>
      <password>lupin</password>
      <port>8080</port>
      <host>my.proxy.url</host>
      <id/>
    </proxy>
  </proxies>
</settings>
```

9. Création d'un nouveau projet avec le plug-in archetype

L'étape suivante est de créer un nouveau squelette de projet Maven 2 pour le composant métier. Maven 2 propose le plug-in *archetype*, qui construit une structure de projet Maven 2. Ce plug-in est très utile pour mettre rapidement en place l'environnement d'un projet basique. Par défaut, archetype produit un modèle de projet pour une bibliothèque JAR. Plusieurs autres types d'artefact sont disponibles, pour d'autre type de projets, comme les applications web, les plug-ins Maven et bien d'autres.

Executez la commande suivante pour mettre en place votre projet HotelDatabase.jar :

```
mvn archetype:create -DgroupId=com.javaworld.hotels
        -DartifactId=HotelDatabase -Dpackagename=com.javaworld.hotels
```

Vous avez maintenant une structure de projet Maven 2 toute neuve. Allez dans le répertoire *HotelDatabase* pour continuer ce tutoriel.

10. Implémentation de la logique métier

Nous allons maintenant implémenter la logique métier. La classe Hotel est un simple JavaBean. La classe *HotelModel* implémente deux services: la méthode *trouveVillesDiponibles()*, qui liste toutes les villes disponibles et la méthode *trouveHotelsParVille()* qui liste les hôtels dans une ville donnée. Une implémentation basique de la classe *HotelModel* est présentée ici :

```
package com.javaworld.hotels.model;

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

import com.javaworld.hotels.businessobjects.Hotel;

public class HotelModel {

    /**
     * La liste de toutes les villes connues dans la base de données.
```

```
     */
    private static String[] villes = { "Paris", "Londres" };

    /**
     * La liste de tous les hotels de la base de données.
     */
    private static Hotel[] hotels = {
            new Hotel("Hotel Latin", "Quartier latin", "Paris", 3),
            new Hotel("Hotel Etoile", "Place de l'Etoile", "Paris", 4),
            new Hotel("Hotel Vendome", "Place Vendome", "Paris", 5),
            new Hotel("Hotel Hilton", "Trafalgar Square", "Londres", 4),
            new Hotel("Hotel Ibis", "The City", "Londres", 3), };

    /**
     * Retourne les hôtels dans une ville donnée.
     * @param ville le nom de la ville
     * @return une liste d'objets Hotel
     */
    public List <Hotel> trouveHotelsParVille(String ville) {
        List <Hotel> hotelsTrouves = new ArrayList <Hotel>();
        for (Hotel hotel : hotels) {
            if (hotel.getVille().equalsIgnoreCase(ville)) {
                hotelsTrouves.add(hotel);
            }
        }
        return hotelsTrouves;
    }

    /**
     * Retourne la liste des villes de la base de données qui ont un hôtel.
     * @return une liste des noms de villes
     */
    public String[] trouveVillesDiponibles() {
        return villes;
    }
}
```

11. Tests unitaires avec Maven 2

Maintenant, passons aux tests de l'application. Quelques classes de test simples peuvent être trouvées dans le code source. Les tests unitaires sont (ou devraient être !) une partie importante de toute application Java. Maven intègre complètement les tests unitaires dans le cycle de développement. Pour exécuter tous vos tests unitaires, vous appelez la phase *test* du cycle de vie :

```
mvn test
```

Une possibilité intéressante de Maven 2 est son utilisation des expressions régulières et du paramètre test pour contrôler les tests que vous voulez exécuter. Si vous voulez uniquement exécuter un test, vous pouvez utiliser le paramètre *test* :

```
mvn test -Dtest=HotelModelTest
```

Si vous voulez exécuter un sous-ensemble de vos tests unitaires, vous pouvez utiliser une expression régulière. Par exemple pour exécuter toutes les classes finissant par *ModelTest*

:

```
mvn test -Dtest=*ModelTest
```

12. Construire et déployer le JAR

Une fois que vous êtes satisfait de vos tests, vous pouvez construire et déployer votre nouveau JAR. La commande install compile, test et mets en paquet vos classes dans un fichier jar puis le deploy dans votre entrepôt Maven 2 local, d'où il peut être utilisé par d'autres projets.

```
mvn install
```

13. Créer l'application web

Maintenant nous voulons utiliser cette bibliothèque pour une application web. Pour simplifier, notre application web consistera uniquement en une JSP (JavaServer Page) qui utilisera directement la classe *HotelModel*. Tout d'abord nous créons un nouveau projet d'application web en utilisant le plug-in archetype :

```
mvn archetype:create -DgroupId=com.javaworld.hotels -DartifactId=HotelWebapp
        -Dpackagename=com.javaworld.hotels -DarchetypeArtifactId=maven-archetype-webapp
```

Ensuite, nous avons besoin d'inclure notre JAR métier dans cette application. Tout ce que nous avons à faire est de rajouter une dépendance au nouveau pom.xml pointant sur le composant HotelDatabase :

```
    <dependency>
      <groupId>com.javaworld.hotels</groupId>
      <artifactId>HotelDatabase</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
```

Maintenant, nous implémentons la JSP unique: Elle liste simplement les villes disponibles et, si une ville est choisie, liste les hôtels correspondants :

```
<html>
<body>
<h2>Application Tutoriel pour la base de données des hotels</h2>
<%@ page import="
   java.util.List,
    com.javaworld.hotels.businessobjects.Hotel,
    com.javaworld.hotels.model.HotelModel"
%>
<%
   HotelModel model = new HotelModel();
   String[] villes = model.trouveVillesDiponibles();
```

```
   String villeSelectionnee = request.getParameter("ville");
   List<Hotel> hotelList = model.trouveHotelsParVille(villeSelectionnee );
%>
   <h3>Choisissez une destination</h3>
   <form action="index.jsp" method="get">
     Merci de choisir une ville :
      <SELECT name="ville">
         <OPTION value="">---Toutes les villes ---</OPTION>
         <%
         for(String villeNom : villes ){
         %>
          <OPTION value="<%=villeNom %>"><%=villeNom %></OPTION>
         <%
         }
         %>
      </SELECT>
      <BUTTON type="submit">Chercher</BUTTON>
   </form>
   <% if (hotelList.size() > 0) { %>
      <h3>Hôtels disponibles à <%=villeSelectionnee %> </h3>
      <table border="1">
           <tr>
             <th>Nom</th>
             <th>Adresse</th>
             <th>Ville</th>
             <th>Catégorie</th>
            </tr>
         <%
         for(Hotel hotel : hotelList){
         %>
           <tr>
             <td><%=hotel.getNom()%></td>
             <td><%=hotel.getAdresse()%></td>
             <td><%=hotel.getVille()%></td>
             <td><%=hotel.getEtoiles()%> étoiles</td>
            </tr>
         <%
         }
         %>
       </table>
    <%}%>
</body>
</html>
```

Nous exécutons maintenant *mvn install* depuis le répertoire HotelWebapp ; cela va compiler, empaqueter et déployer le fichier HotelWebapp.war dans notre entrepôt local (vous pouvez également le trouver dans le répertoire *target* si nécessaire). Maintenant vous pouvez déployer ce fichier war dans votre serveur d'application favori et voir ce que vous avez (voir figure 4).

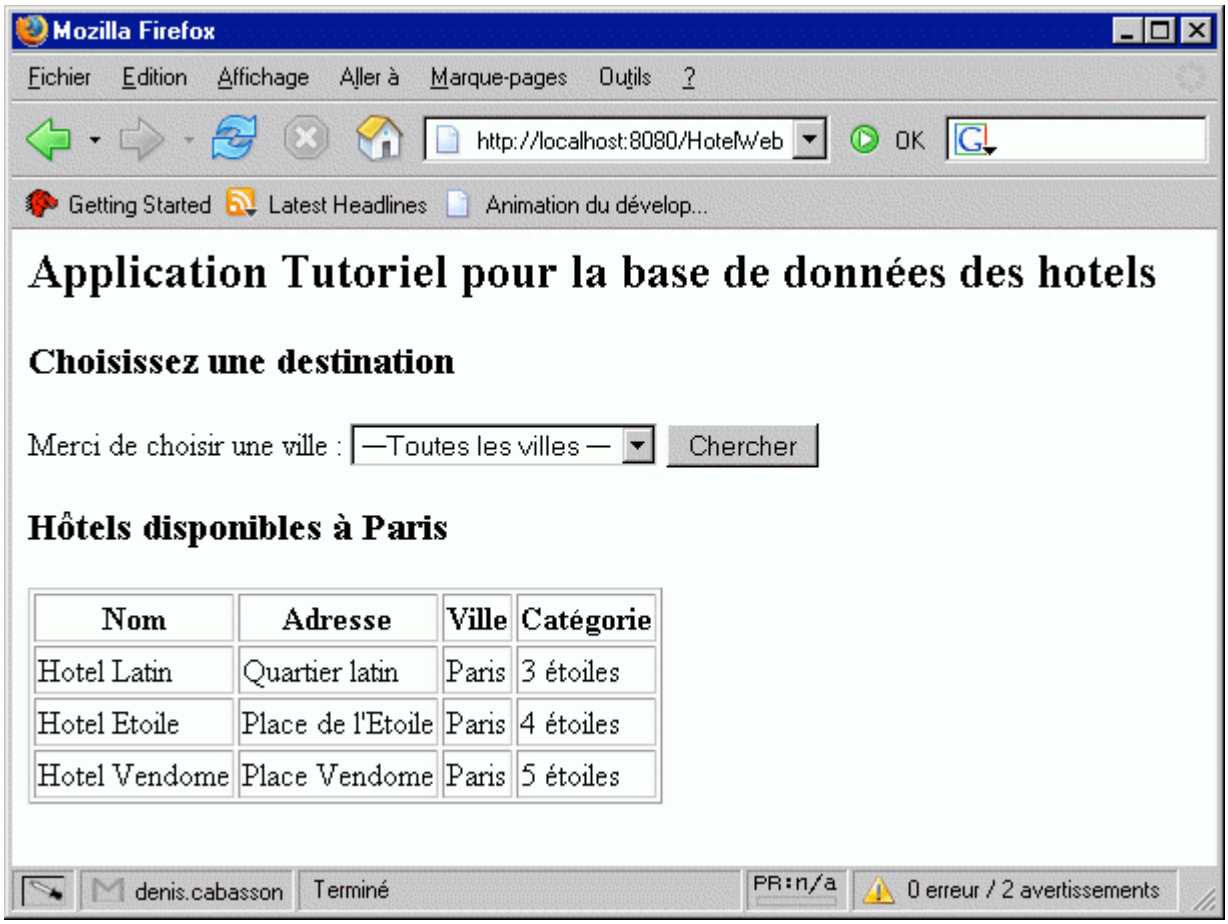

Figure 4. L'application exemple

14. Travailler avec les plugs-in

Maven 2 arrive avec un nombre toujours croissant de plug-ins qui ajoute des fonctionnalités supplémentaires à votre processus de build sans grand effort. Pour utiliser un plug-in, vous le liez à une phase du cycle de vie. Maven trouvera alors quand (et comment) l'utiliser. Certains plugs-in sont déjà utilisés par Maven par derrière, vous avez donc uniquement besoin de les déclarer dans la section *plugins* de votre fichier pom.xml. Le plug-in suivant, par exemple, est utilisé pour compiler du code avec le J2SE 1.5 :

```
...
<build>
   ...
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-compiler-plugin</artifactId>
         <configuration>
            <source>1.5</source>
            <target>1.5</target>
         </configuration>
      </plugin>
   </plugins>
</build>
```

Dans d'autres cas, vous liez le plug-in à une phase du cycle de vie de façon à ce que Maven sache quand l'utiliser. Dans l'exemple suivant, nous exécutons une tache Ant standard. Pour ce faire, nous lions le plug-in *maven-antrun-plugin* à la phase *generate-sources* et ajoutons la tache Ant entre les balises tag, comme montré ici :

```
  ...
<build>
      ...
    <plugins>
       <plugin>
          <artifactId>maven-antrun-plugin</artifactId>
          <executions>
            <execution>
              <phase>generate-sources</phase>
              <configuration>
                <tasks>
                  <!-- Les taches ant vont ici -->
                </tasks>
              </configuration>
              <goals>
                <goal>run</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
     </plugins>
  </build>
```

Conclusion

Maven 2.0 est un outil très puissant qui simplifie et standardise énormément le processus de build. En favorisant une organisation standard des projets et des bonnes pratiques reconnus, Maven ôte une bonne quantité de travail. Et des plugs-in standards, comme le générateur de site web, offrent des outils collaboratifs intéressants avec peu d'effort. Vérifiez par vous-même !

- Les fichiers source de l'application exemple
- Téléchargez Maven 2
- La page d'accueil de Maven 2
- Documentation sur le cycle de vie Maven 2
- Des tips sur Maven 2 sur le blog de John Smart

Traduction pour http://www.developpez.com par Denis Cabasson de l'article An introduction to Maven 2 de John Ferguson Smart paru le 05/12/2005 sur JavaWorld. JavaWorld ne peux en aucun cas être tenu pour responsable du contenu de cette traduction.