Configuración de aplicaciones multientorno con Maven

Por **Ángel García Jerez** - 12 julio, 2011

Configuración de aplicaciones multientorno con Maven

1. Introducción

Maven es una de las herramientas más utilizadas para gestionar y generar la aplicaciones en el mundo Java. Las aplicaciones habitualmente tienen recursos que dependen del entorno donde se ejecutan. En este tutorial os enseñaremos un ejemplo de como configurar maven para poder generar la aplicación para diferentes entornos.

2. Entorno

Entorno utilizado para escribir este tutorial:

- **Hardware**: Mac Book Pro (Core 2 Duo 2,8 Ghz, 4 GB RAM, 500 GB)
- **Sistema Operativo**: Snow Leopard 10.6.8
- **Maven**: 3.0.3

3. Ejemplo práctico

Vamos a poner un ejemplo práctico: tendremos una aplicación web que utiliza la librería Liquibase. Para aquellos que no la conozcáis se encarga de llevar el control de cambios que se van produciendo en nuestro schema de base de datos a lo largo del ciclo de vida de nuestra aplicación. No vamos a entrar en detalle en ella pero para aquellos que tengáis curiosidad os recomiendo que os leáis el siguiente tutorial "Liquibase-Gestión De Cambios En Base De Datos".

Esta librería nos permite definir "contextos" similares a los perfiles de maven o spring. En este caso agrupan conjuntos de cambios (changeset) contra un schema. Estos contextos se activan a través de una propiedad en Liquibase. En este ejemplo configuraremos Liquibase mediante un Listener en el fichero web.xml, aunque existen otras forma de configuración/ejecución.

Habitualmente las aplicaciones utilizan juegos de datos específicos para el entorno de desarrollo que no deben propagarse al resto de entornos. Con Liquibase esto lo podemos

realizar fácilmente creando un contexto para desarrollo (development) y otro para producción (production).

Por tanto, dependiendo del entorno donde ejecutemos nuestra aplicación deberemos inicializar Liquibase con unos contextos u otros. Aunque podemos cambiar la configuración de Liquibase cada vez que tengamos que generar la aplicación para cada entorno, lo más óptimo y adecuado es automatizar este proceso utilizando filtros de maven.

4. Preparando el ejemplo

Antes de configurar nuestro proyecto con los filtros de maven vamos a ver el estado inicial de la aplicación antes de añadirlos.

En el fichero web.xml tendremos la configuración de liquibase:

```Shell
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/w
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/
    id="WebApp_ID" version="2.5">

    <display-name>Maven Filter</display-name>

    <listener>
        <listener-class>liquibase.integration.servlet.LiquibaseServletListener</listener-cla
    </listener>


    <context-param>
        <param-name>liquibase.changelog</param-name>
        <param-value>liquibase/changelogs.xml</param-value>
    </context-param>
    <context-param>
        <param-name>liquibase.datasource</param-name>
        <param-value>java:comp/env/jdbc/filterDS</param-value>
    </context-param>
    <context-param>
        <param-name>liquibase.onerror.fail</param-name>
        <param-value>true</param-value>
    </context-param>
    <context-param>
        <param-name>liquibase.contexts</param-name>
        <param-value>development,production</param-value>
    </context-param>


</web-app>
```

Como podéis ver hemos configurado liquibase mediante su listener y hemos añadido las propiedades necesarias para que este funcione. De todas las propiedades sólo destacaremos “liquibase.contexts” con la que definimos los contextos que van a ser ejecutados. En este caso “development” y “production”.

El fichero con las sentencias que se van a ejecutar contra nuestro schema:

```Shell
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
    xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog http://www.liquibase.org
    http://www.liquibase.org/xml/ns/dbchangelog-ext http://www.liquibase.org/xml/ns/dbchange
```

```
7
8       <changeSet id="1" author="autentia" context="production">
9           <createTable tableName="Client">
10              <column name="id" autoIncrement="true" type="INT">
11                  <constraints primaryKey="true" nullable="false"/>
12              </column>
13              <column name="name" type="VARCHAR(8)" >
14                  <constraints nullable="false"/>
15              </column>
16              <column name="password" type="VARCHAR(50)">
17                  <constraints nullable="false"/>
18              </column>
19              <column name="active" type="BIT(1)">
20                  <constraints nullable="false"/>
21              </column>
22          </createTable>
23          <addUniqueConstraint tableName="Client" columnNames="name"/>
24      </changeSet>
25
26      <changeSet id="2" author="autentia" context="development">
27          <sql>
28              INSERT INTO Client (name,password,active) values ('maven','abcb21LQTcIANtvYMT7Q\
29          </sql>
30      </changeSet>
31
32
33  </databaseChangeLog>
```

Prestad atención al valor de context de los changeset. El primero se ejecutará cuando activemos el contexto de “production” y el segundo en “development”.

Y el fichero pom.xml:

Shell

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSche
2      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0
3      <modelVersion>4.0.0</modelVersion>
4      <groupId>com.autentia</groupId>
5      <artifactId>mavenfilter</artifactId>
6      <packaging>war</packaging>
7      <version>1.0.0-SNAPSHOT</version>
8      <name>mavenfilter Maven Webapp</name>
9      <dependencies>
10         <!-- ========================= liquibase ========================= -->
11         <dependency>
12             <groupId>org.liquibase</groupId>
13             <artifactId>liquibase-core</artifactId>
14             <version>2.0.1</version>
15         </dependency>
16         <!-- ========================= HsqlDB ========================= -->
17         <dependency>
18             <groupId>org.hsqldb</groupId>
19             <artifactId>hsqldb</artifactId>
20             <version>2.2.4</version>
21         </dependency>
22     </dependencies>
23     <build>
24         <finalName>mavenfilter</finalName>
25     </build>
26 </project>
```

Ahora sólo nos queda configurar maven para que podamos generar nuestra aplicación para diferentes entornos.

5 Habilitando los filtros en nuestro proyecto

Lo primero es crear dos ficheros (uno para producción y otro para desarrollo) donde añadimos las propiedades con los valores multientorno guardándolos en el directorio

src/main/filters.

En nuestro caso los ficheros sólo tendrán una única propiedad : los valores de los contexto que liquibase ejecutará cuando se despliegue la aplicación.

filter-dev.properties

```Shell
mavenfiler.liquibase.contexts=development,production
```

filter-prod.properties

```Shell
mavenfiler.liquibase.contexts=production
```

Como os podéis dar cuenta el nombre de los ficheros es muy importante ya que cada uno contendrá los valores correspondientes para cada uno de los entornos.

Ahora debemos configurar Maven para que sustituya los valores de estos ficheros en nuestros recursos de la aplicación. Abrimos el fichero pom.xml y añadimos las siguientes líneas:

```Shell
<properties>
    <env>prod</env>
</properties>
<build>
    <finalName>mavenfilter</finalName>
    <filters>
        <filter>src/main/filters/filter-${env}.properties</filter>
    </filters>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-war-plugin</artifactId>
            <version>2.1.1</version>
            <configuration>
                <webResources>
                    <resource>
                        <filtering>true</filtering>
                        <directory>src/main/webapp</directory>
                        <includes>
                            <include>WEB-INF/web.xml</include>
                        </includes>
                    </resource>
                </webResources>
            </configuration>
        </plugin>
    </plugins>
</build>
```

En la línea 2 se define la propiedad “env” e indica el entorno para el que maven generará la aplicación web. Por defecto el valor es “prod”, en el caso de que queramos cambiar el valor podremos hacerlo mediante perfiles de maven o añadiendo un parámetro al ejecutar maven por línea de comandos (al final del tutorial configuraremos un perfil que cambia el valor de esta propiedad).

En la línea 7 indicamos el fichero de propiedades que utilizará maven para sustituir los valores multientorno. Como podéis ver tiene el patrón ${env}. Con esto conseguimos que

cuando se ejecute maven utilice el fichero de producción o de desarrollo en función del valor de la propiedad.

Y por último configuramos el plugin maven-war-plugin para que sustituya todos los patrones de los recursos de la aplicación por el valor definido en los ficheros de filtros. Hacer hincapié en la línea 17 donde se habilita la ejecución de los filtros, la línea 18 que indica sobre que directorio se aplica los filtros y la línea 20 con el recurso sobre el que se aplica finalmente los filtros.

Una vez que hemos finalizado toda la configuración de maven tendremos que añadir los patrones en aquellos recursos que dependen de entorno, en nuestro caso el fichero web.xml.

```Shell
<context-param>
    <param-name>liquibase.contexts</param-name>
    <param-value>${mavenfiler.liquibase.contexts}</param-value>
</context-param>
```

El cambio es mínimo, donde antes teníamos el valor “development,production” ahora debemos tener el nombre de la propiedad de nuestro fichero de filtro entre ${}.

Si ejecutamos “mvn clean package” veremos que el valor que contendrá el fichero web.mxl corresponderá con el valor de la propiedad de filter-prod.properties.

Si queréis descargaros el proyecto de ejemplo lo podréis hacer desde aquí.

6 Habilitando el perfil de desarrollo

Cuando un desarrollador trabaja con este tipo de proyectos es muy habitual que configure un perfil maven para redefinir el valor de la propiedad “env” y así ahorrar bastante tiempo a la hora de generar la aplicación para este entorno.

Únicamente tendrá que modificar su fichero settings.xml con:

```Shell
<profiles>
...
      <profile>
        <id>development</id>
        <properties>
            <env>dev</env>
        </properties>
    </profile>
...
 </profiles>
<activeProfiles>
    <activeProfile>development</activeProfile>
</activeProfiles>
```

Añadimos un perfil con el nombre “development” donde la propiedad “env” tendrá el valor “dev” y en la línea 12 activamos dicho perfil.

Esto hará que cuando generemos el proyecto se haga con las propiedades de desarrollo.

7 Conclusión

Las aplicaciones cuya configuración es dependiente de entorno son muy comunes y con este tutorial hemos querido enseñaros como gestionarlas utilizando maven. Como habéis podido observar es realmente sencillo.

Aunque en este tutorial hemos utilizado los filtros para las aplicaciones multientorno también podéis utilizarlos para otros menesteres según vuestras necesidades.

Ángel García Jerez

Consultor tecnológico de desarrollo de proyectos informáticos.
Co-autor del libro "Actualización y mantenimiento del PC (Edición de 2010) publicado por Anaya Multimedia.
Ingeniero Técnico en Informática de Sistemas e Ingeniero en Informática (premio al mejor expediente de su promoción).
Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación.
Somos expertos en Java/Java EE