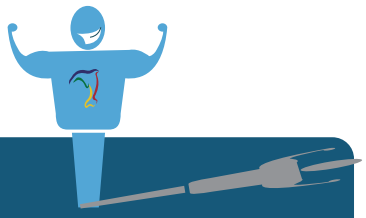
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

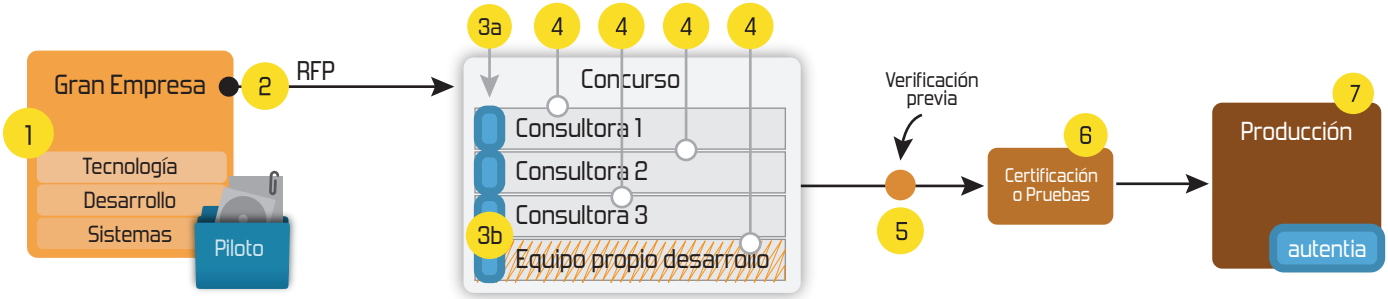
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com



autentia
real business solutions
Hosting patrocinado por
enredados



DESARROLLADO POR:
Francisco J. Arroyo

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/JEE

Anuncios Google Java J2EE Tutorial LOG4J Tutorial Sun Java XML

Fecha de publicación del tutorial: 2009-02-26

42

Share | Regístrate para votar

Consumir servicio web rest en Axis2 con Android

0. Índice de contenidos.

- 1. Introducción
- 2. Entorno
- 3. Instalación
- 4. Crear el servidor
- 5. Crear el cliente
- 6. Conclusiones

1. Introducción

En este tutorial se pretende mostrar, paso a paso, cómo consumir un servicio web REST con Axis2 desde una aplicación Android. Está dirigido a personas que apenas tienen experiencia previa en montar un servicio web con Java.

Más específicamente, vamos a construir una aplicación Android que nos va a mostrar varias listas de compras, cada una con sus productos, y toda la información de esas listas se van a cargar a través de un servicio web.

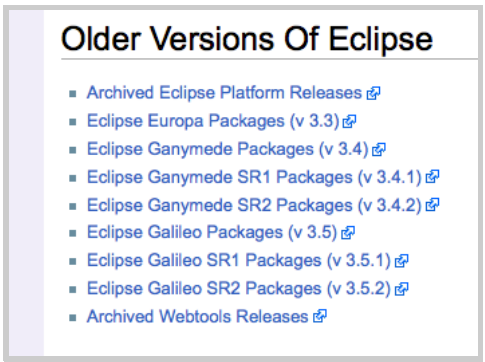
2. Entorno

- Hardware
  - Mackbook Pro
    - Intel Core i7 2Ghz
    - 8GB RAM
    - 500GB HD
    - Sistema Operativo: Mac OS X (10.6.6)
  - Nexus One
    - Android 2.3.3 (Gingerbread).
- Software
  - Eclipse Galileo
  - PostgresSQL 9.0.3
  - Hibernate 3
  - Spring 2.5
  - Axis 1.4.1
  - Apache Tomcat 6.0

3. Instalación

El gestor de base de datos que vamos a utilizar PostgresSQL, en concreto la versión 9.0.3. Para obtener el gestor nos dirigiremos hacia http://www.postgresql.org/download/ y descargaremos la versión que corresponda con nuestro sistema operativo.

La herramienta de desarrollo que vamos a utilizar es Eclipse, la versión Galileo. Si el lector no dispone de esta herramienta, puede ir a http://wiki.eclipse.org/Older_Versions_Of_Eclipse



y elegir el paquete "Eclipse IDE for Java EE Developers, (188MB)"

Catálogo de servicios Autentia

Últimas Noticias

XV Charla Autentia - web2py (y Google App Engine) - RECORDATORIO

XV Charla Autentia - web2py (y Google App Engine)

XIV Charla Autentia - ZK - Vídeos y Material

Hablando de coaching ágil, milagro nocturo y pruebas de vida

XIII Charla Autentia - AOS y TDD - Vídeos y Material

Histórico de NOTICIAS

Últimos Tutoriales

Autentia en la javacup

Listener del ciclo de vida de JSF, en JSF2 y con el soporte de FacesTrace (Primefaces)

Log4J: Modificar el nivel de log en tiempo de ejecución con el soporte de Spring

Lanzar test de Selenium 2 en un navegador remoto

CRUD con Spring MVC Portlet (IV): Realizando pruebas unitarias

Síguenos a través de:

Últimas ofertas de empleo

2011-03-02
T. Información - Analista / Programador - MALAGA.

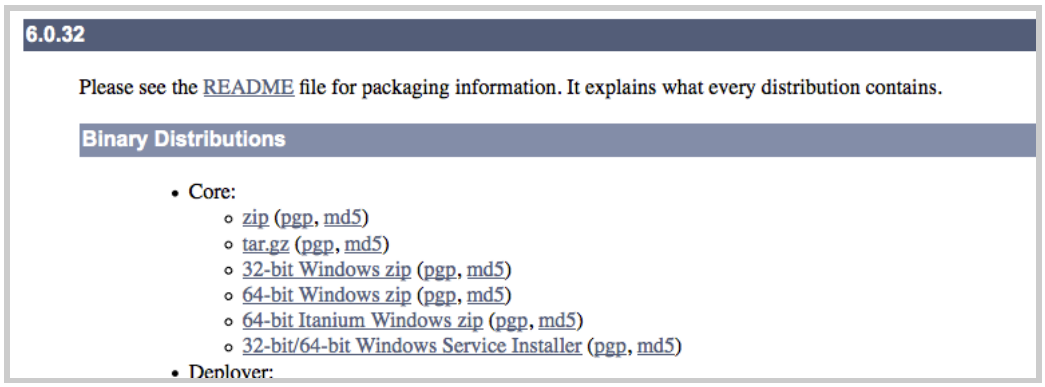
2011-02-24
T. Información - Especialista CRM - MADRID.

2011-02-16
Marketing - Experto en Marketing - CADIZ.

2011-02-08

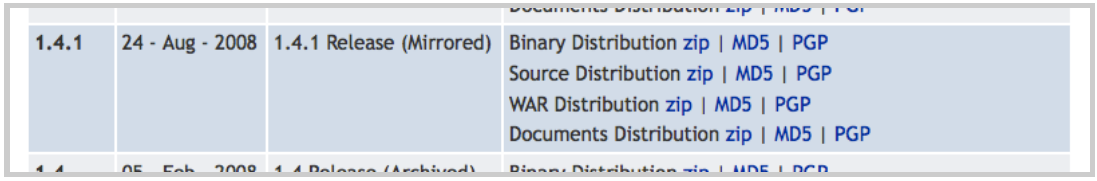


El servidor de aplicaciones que vamos a utilizar es Apache Tomcat 6.0, si no lo tenemos, necesitaremos descargarlo. Para obtenerlo, nos dirigimos a http://tomcat.apache.org/download-60.cgi y nos bajamos los binarios de "Core", en mi caso elijo el "tar.gz"

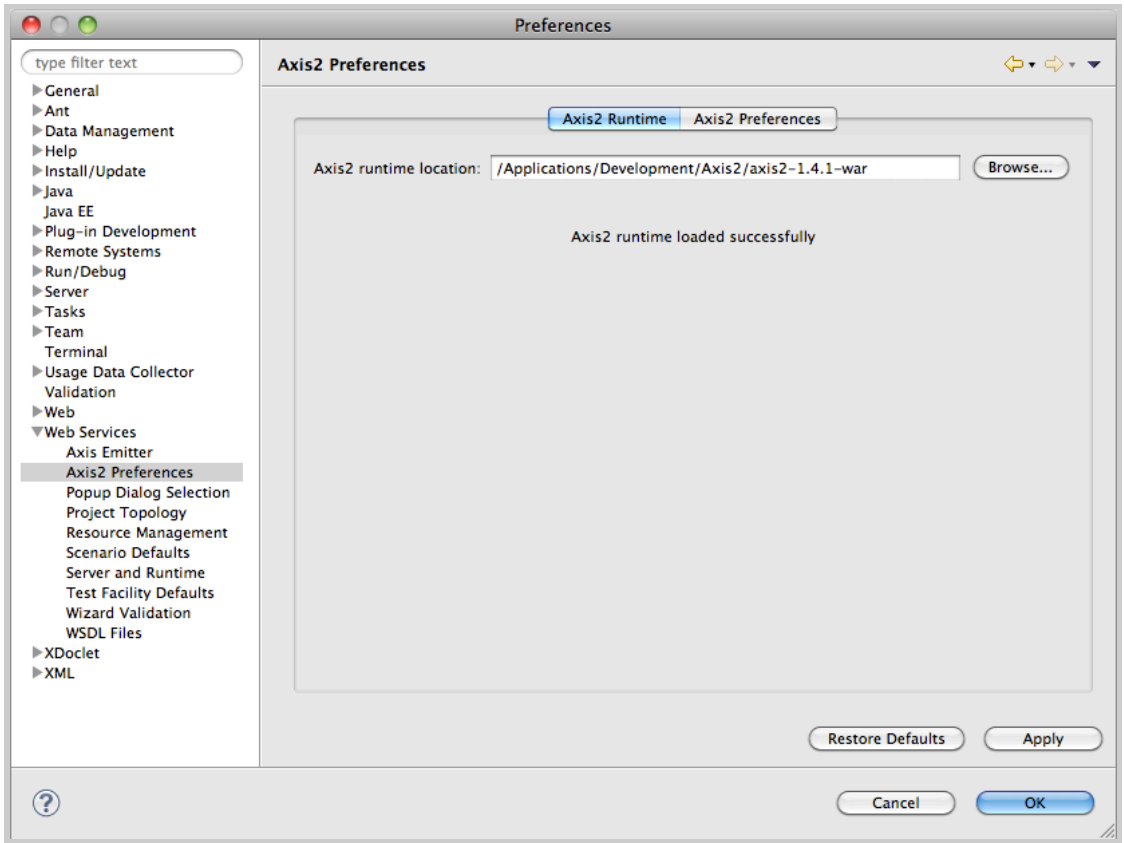


y lo descomprimimos en una ubicación que utilizaremos más tarde.

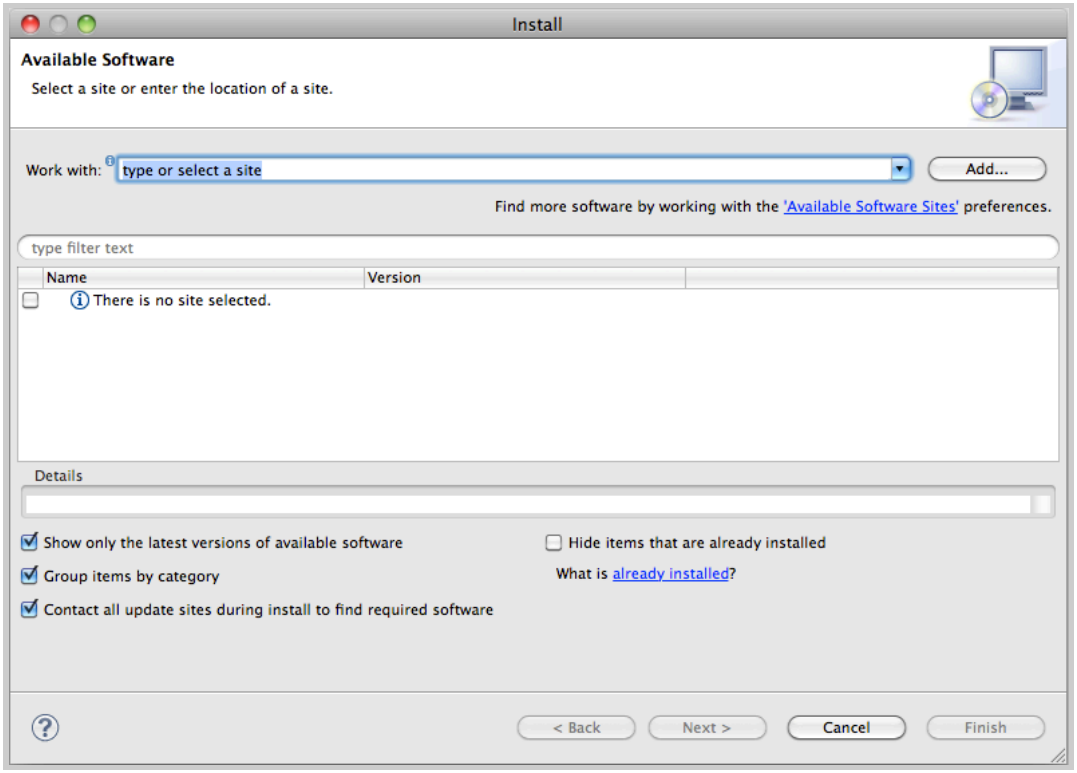
Además nos van a hacer falta las librerías de Axis2. Para descargarlas, vamos a http://axis.apache.org/axis2/java/core/download.cgi y elegimos de la versión 1.4.1, la distribución WAR.



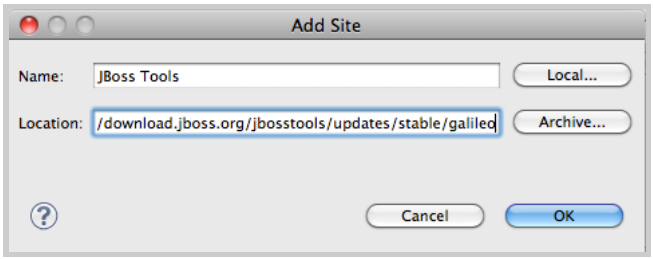
Una vez descargada la descomprimimos y configuramos Eclipse para que sepa donde se encuentra. Para configurar Eclipse, abrimos Eclipse - > Preferences, y dentro de Preferences, nos dirigimos a WebServices -> Axis2 Preferences y le indiciamos donde se encuentra el WAR que acabamos de descargar.



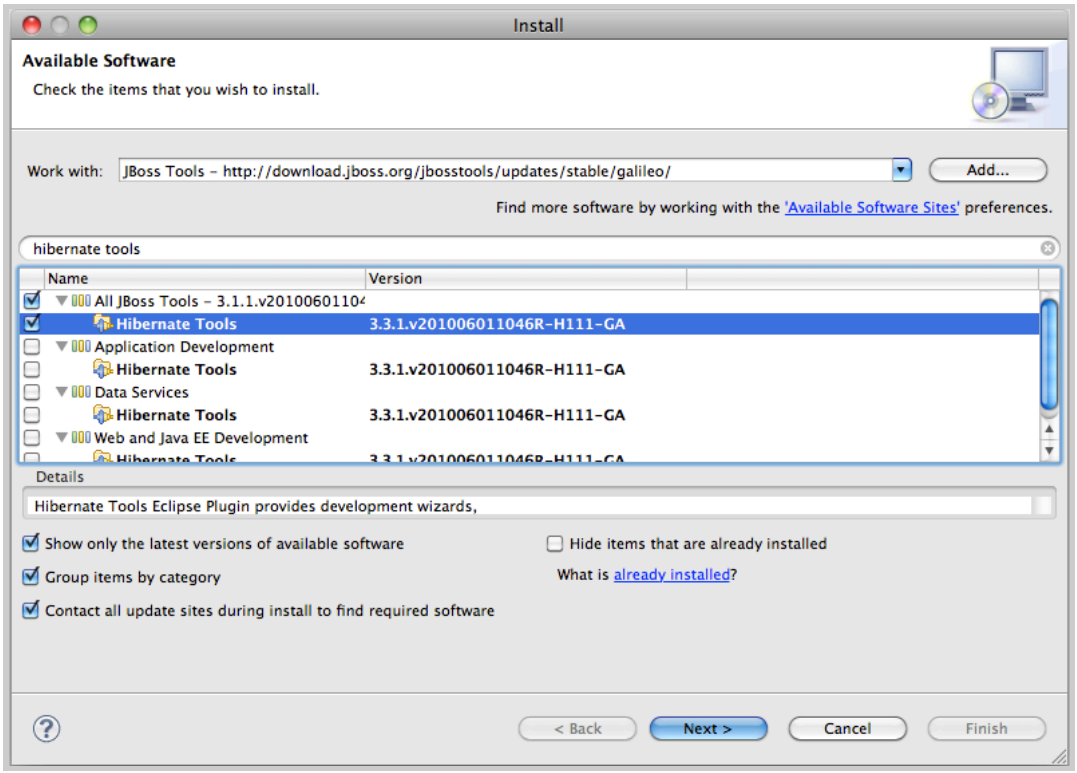
También vamos a instalar "Hibernate-tools" para que nos ayude con la configuración de Hibernate a la hora de crear los archivos de mapeo de nuestros Bean. Para ello, en Eclipse, abrimos "Help" -> "Install New Software",



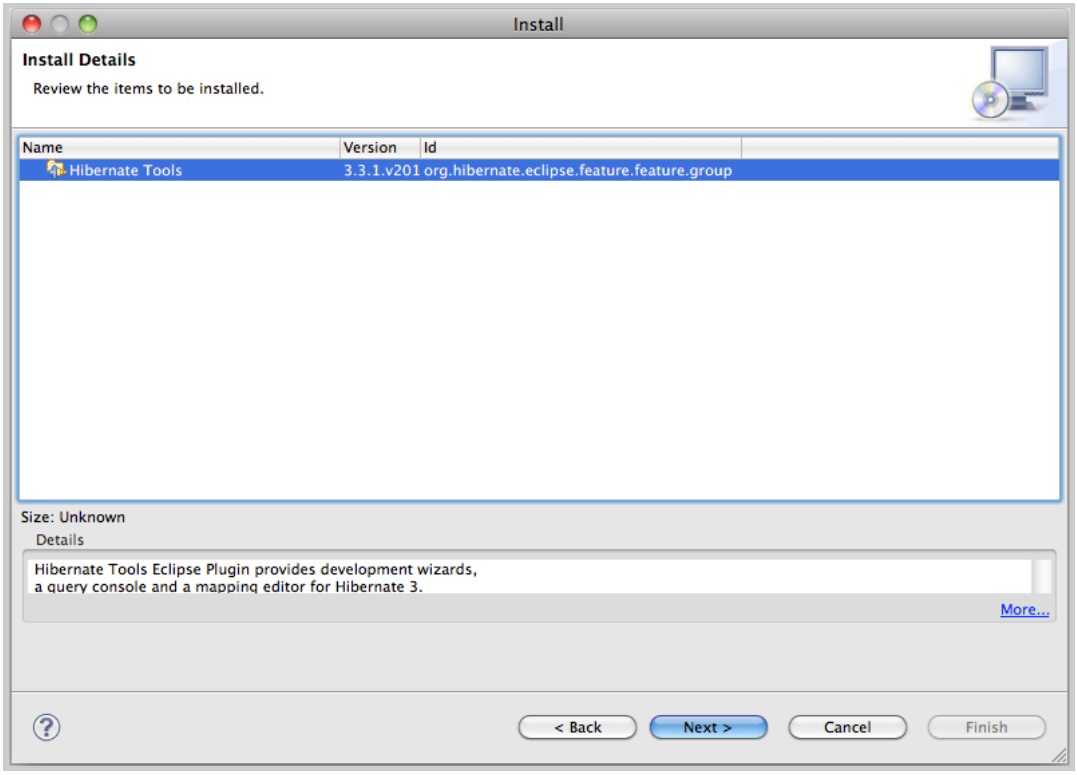
y añadimos el repositorio de Jboss en donde se van a encontrar las "Hibernate-tools". Para añadir el repositorio, pulsamos el botón "Add" y rellenamos la ventana. En "Name" le ponemos un nombre al repositorio, y en "Location", especificamos la ruta en donde se encuentran los archivos. En mi caso he puesto como "Name" Jboss Tools y en "Location" "http://download.jboss.org/jbosstools/updates/stable/galileo/"



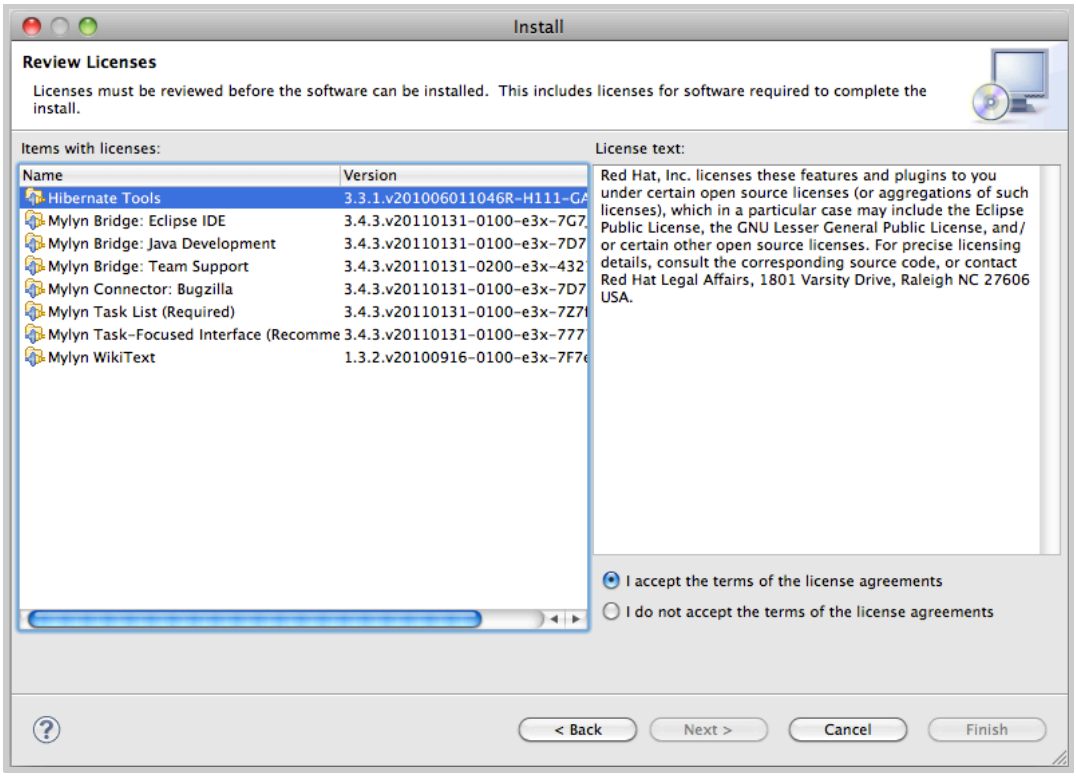
Pulsamos en ok, Entonces filtramos por "Hibernate tools", marcamos el check que corresponda con "Hibernate tools" y pulsamos en "Next".



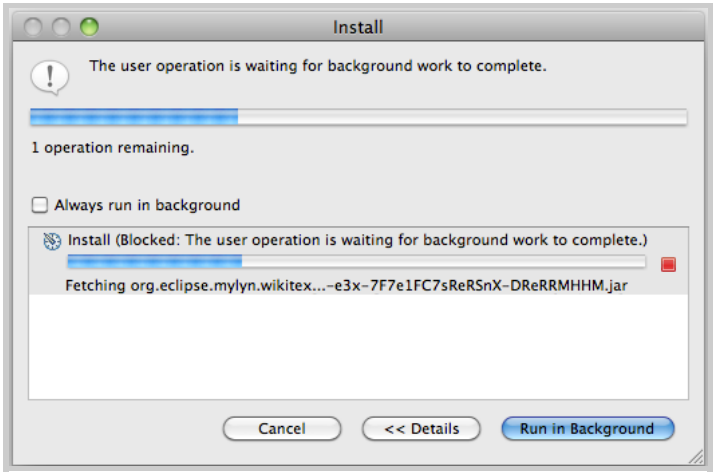
Entonces se pondrá a calcular dependencias y tras un rato, aparecerá una pantalla como esta:



Pulsamos en "Next"



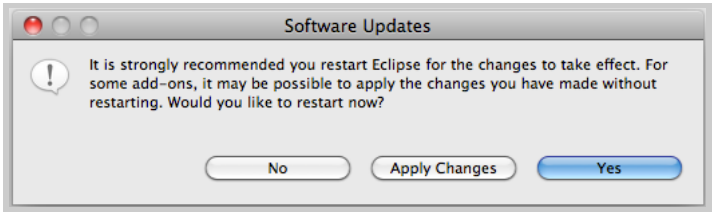
Aceptamos todas las licencias y le damos a "Finish" con lo que empezará a descargar las "Hibernate-tools"



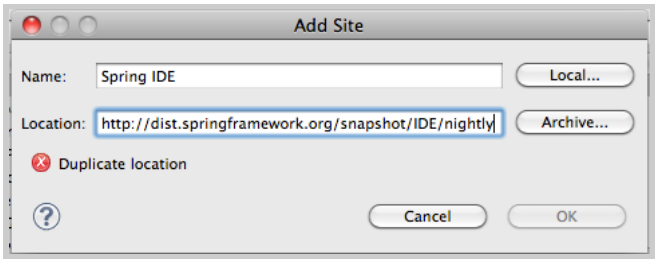
Cuando termine de adquirir los archivos necesarios, nos mostrará un aviso indicándonos que el contenido que se ha descargado no está firmado. Le damos a "Ok" y empezará con la instalación del plugin.



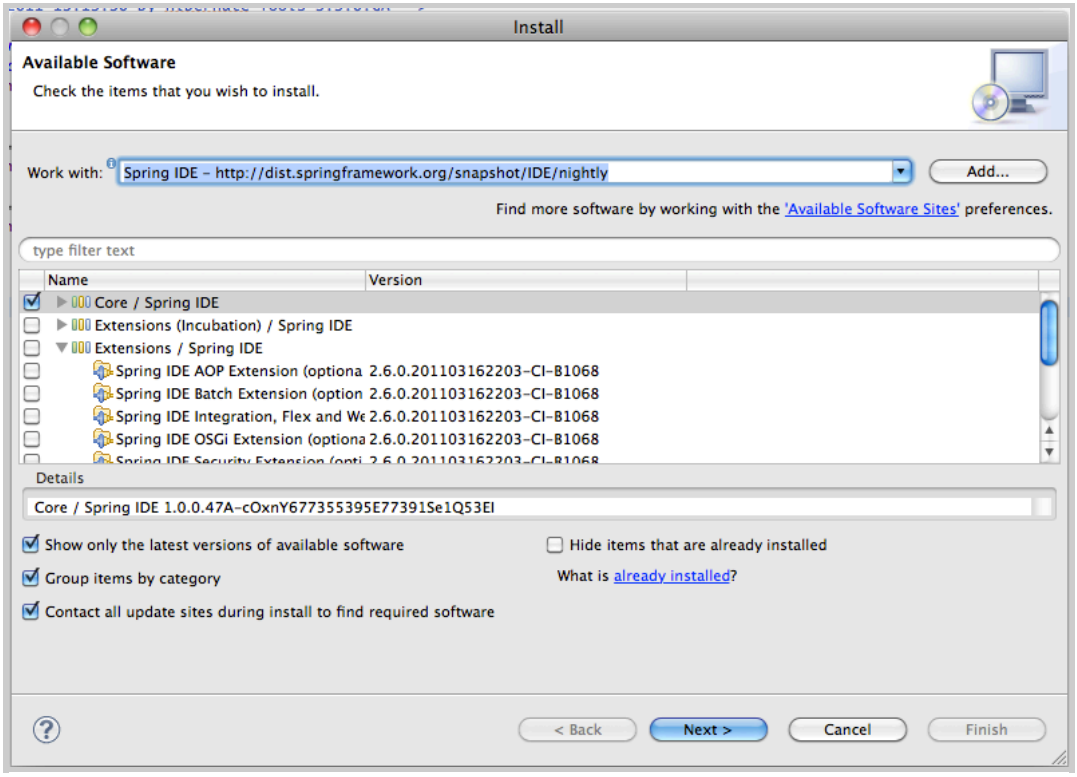
Una vez instalado nos mostrará un aviso indicándonos que necesita reiniciar Eclipse para que los cambios se hagan efectivos. Pulsamos en "Yes" y reiniciará Eclipse.



También nos va a facilitar la tarea si instalamos el plugin SpringIde para Eclipse. Para instalarlo, lo haremos, igual que en el caso anterior, a través de Eclipse. Pinchamos en "Help" -> "Install New Software" y añadimos un nuevo repositorio. En este caso tendremos que rellenar el campo "Location" con "http://dist.Springframework.org/snapshot/IDE/nightly" en "Name" pondremos algo del estilo "SpringIDE"



Una vez que ha creado el indice de los paquetes seleccionamos "Core / Spring IDE"



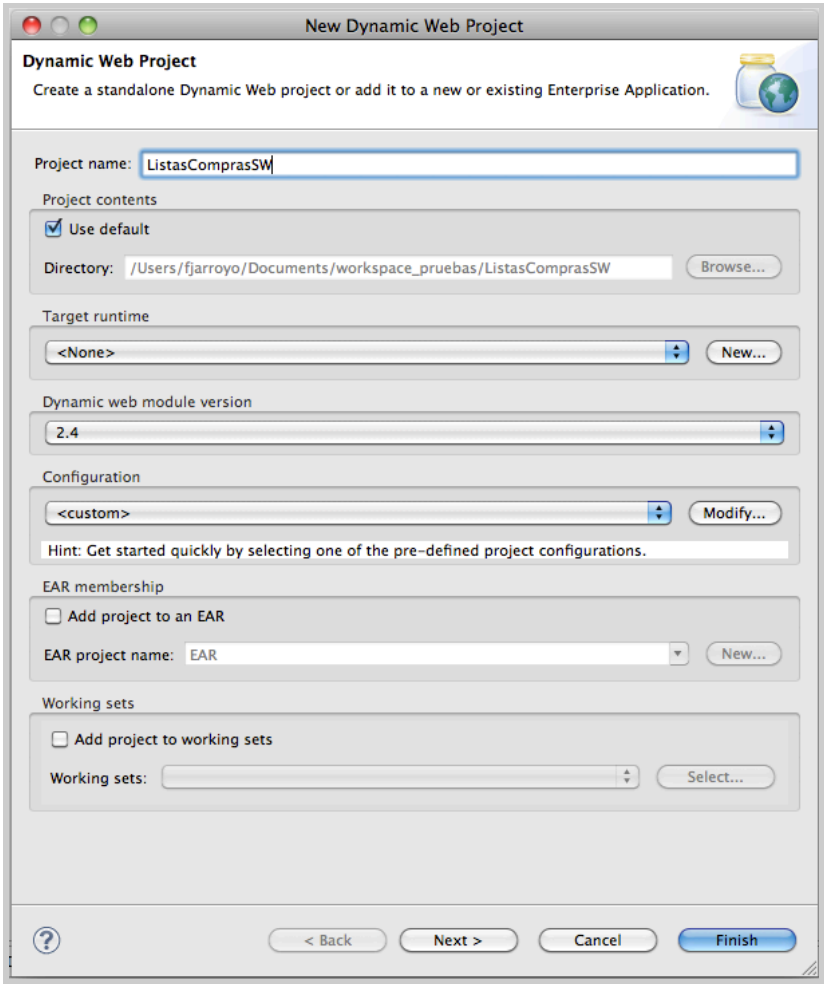
y al igual que en el caso anterior, le damos a "Next" y aceptamos tanto la licencia como el aviso de instalación de software sin firmar.

NOTA: Se obviará la instalación y configuración del sdk de android y de los plugins de Android para Eclipse

4. Crear el servidor

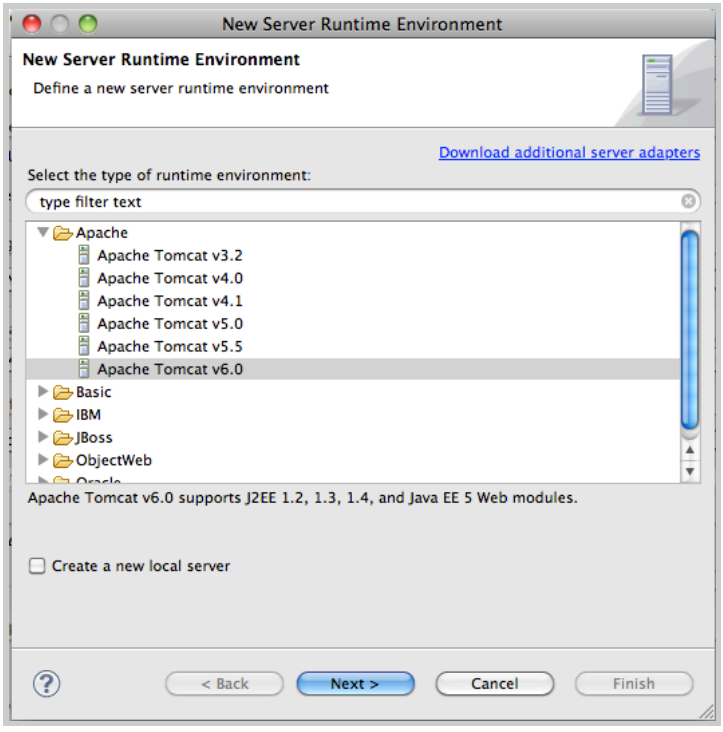
Una vez que tenemos configurado todo lo necesario para realizar el proyecto, empezamos con lo divertido, a codificar :).

Desde eclipse, creamos un nuevo proyecto Web, por lo que clickeamos en File -> New Project -> Dynamic Web Project y nos aparecerá una pantalla como esta

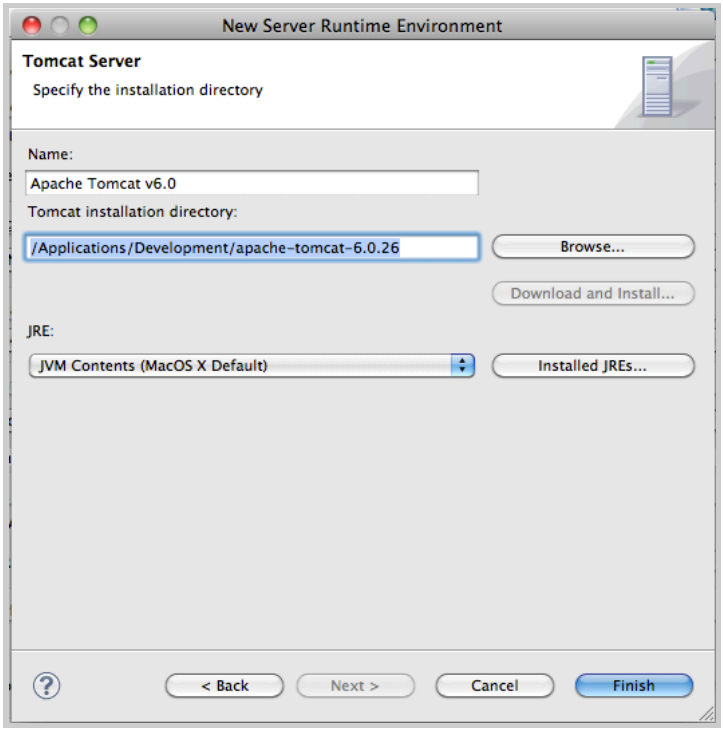


En nuestro caso el proyecto se va a llamar ListaComprasSW.

En esta misma ventana, vamos a especificar (aunque se puede hacer posteriormente) el "Target Runtime" haciendo click en el botón "New",

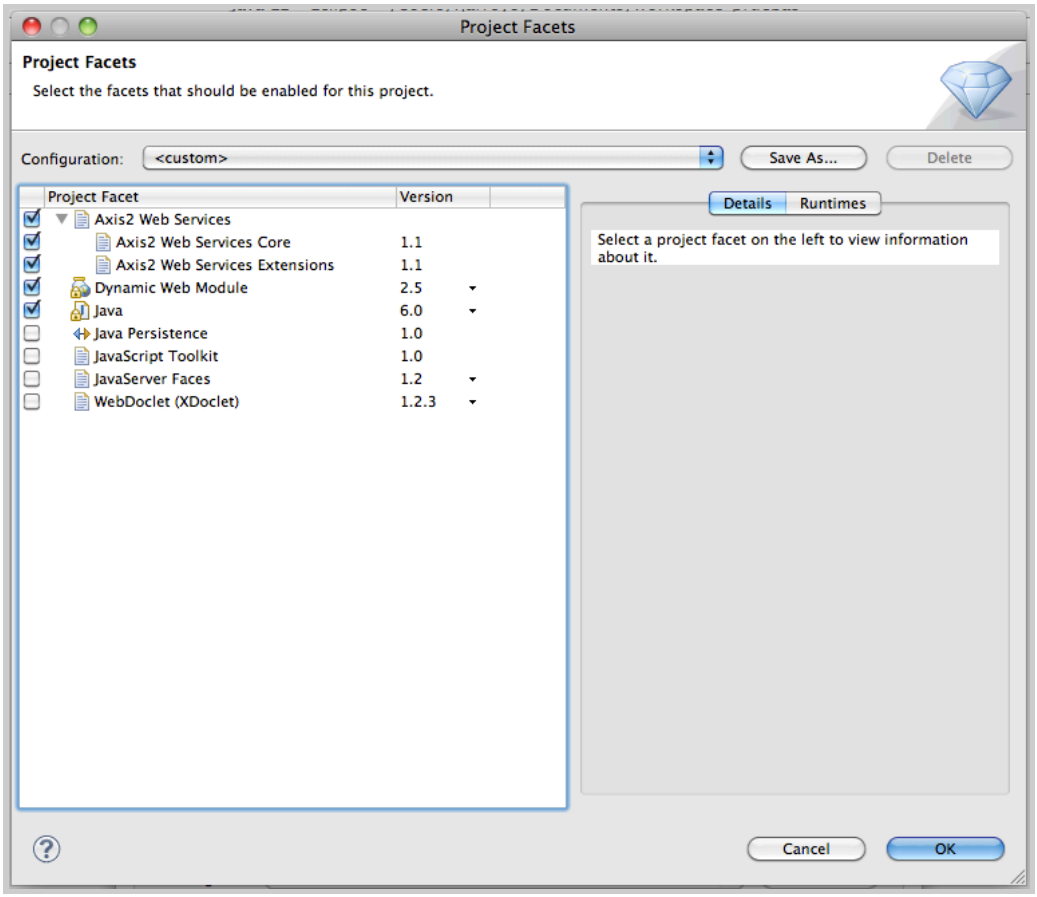


Escogemos, "Apache Tomcat v6.0" y pulsamos el botón "Next".

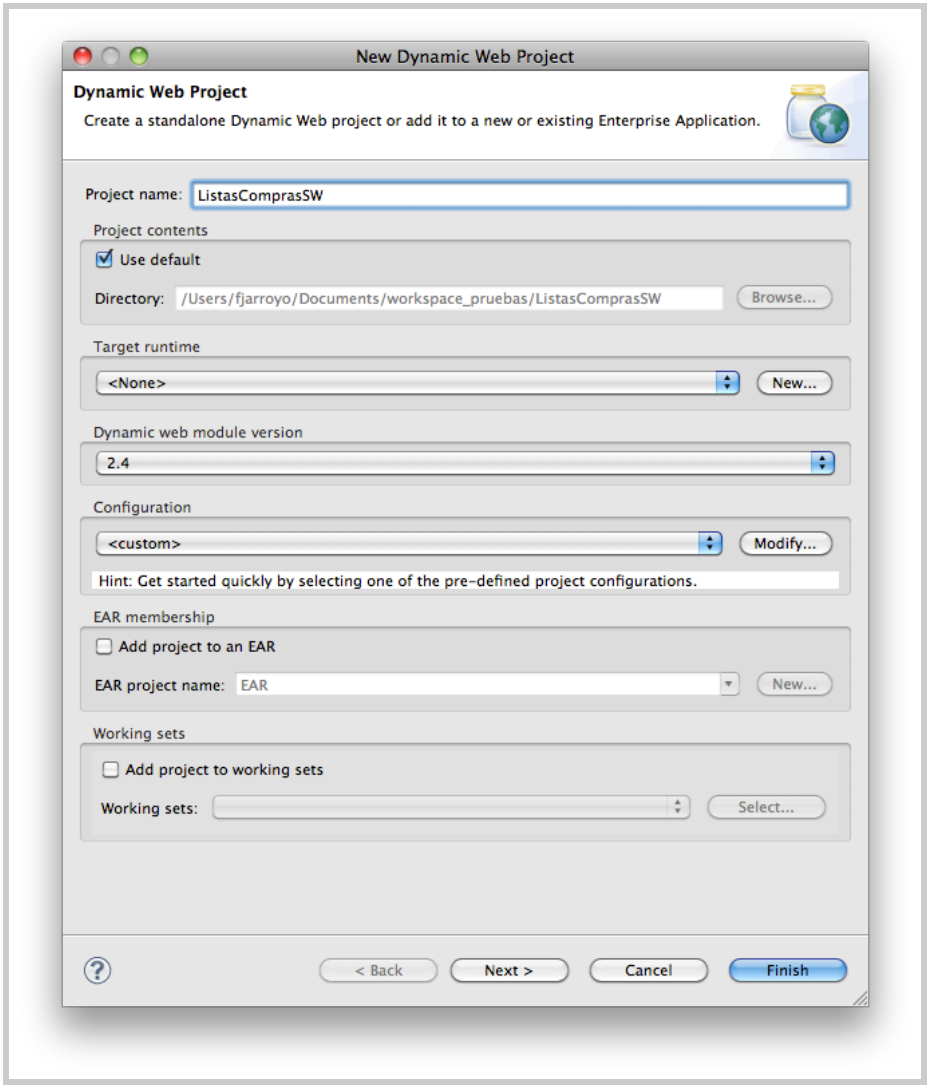


En la ventana que nos aparece, pulsamos en "Browse", y seleccionamos la ruta en la que hemos descomprimido anteriormente el "Apache Tomcat" que hemos descargado.

Pulsamos "Finish" y volvemos a la ventana que nos aparece cuando creamos un nuevo proyecto y pulsamos en el botón "Modify". Habilitamos "Axis2 Web Services Core" y "Axis2 Web Services Extensions" quedándonos como la pantalla que se muestra a continuación.



Con lo que finalmente, la ventana de configuración de un nuevo proyecto nos queda así:



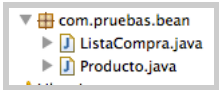
Pulsamos "Finish" y nos creará el esqueleto de la aplicación.

Lo primero que vamos a hacer es el modelo de datos. En nuestro caso, el modelo es muy sencillo, consta de dos tablas y de una relación 1 a N.

Una tabla en donde guardaremos las listas de la compra de nombre listacompra, que tendrá un idListaCompra que será su clave primaría y dos campos más en donde guardaremos la fecha en la que se creó y el nombre de la lista de la compra.

La segunda tabla será en la que almacenaremos los productos que queremos guardar en la lista de la compra. Esta tabla contiene un idCompra que es su clave principal, un campo para guardar el nombre del producto, otro campo para guardar si el producto se ha cogido ya y una clave foránea que obtendrá de la tabla listacompra para crear la relación.

Este modelo queda reflejado en dos beans que agregaremos al proyecto, dentro del paquete com.pruebas.bean.



El código correspondiente a ListaCompra.java es

```
view plain  print  ?
01. package com.pruebas.bean;
02.
03. import java.util.List;
04.
05. public class ListaCompra {
06.     int idListaCompra;
07.     String nombre;
08.     String fecha;
09.     List<Producto> productos;
10.
11.     public ListaCompra()
12.     {
13.
14.     }
15.
16.     public int getIdListaCompra()
17.     {
18.         return idListaCompra;
19.     }
20.
21.     public void setIdListaCompra(int idListaCompra)
22.     {
23.         this.idListaCompra = idListaCompra;
24.     }
25.
26.     public String getNombre()
27.     {
28.         return nombre;
29.     }
30.
31.     public void setNombre(String nombre)
32.     {
33.         this.nombre = nombre;
34.     }
35.
36.     public String getFecha()
37.     {
38.         return fecha;
39.     }
40.
41.     public void setFecha(String fecha)
42.     {
43.         this.fecha = fecha;
44.     }
45.
46.     public List<Producto> getProductos()
47.     {
48.         return productos;
49.     }
50.
51.     public void setProductos(List<Producto> productos)
52.     {
53.         this.productos = productos;
54.     }
55. }
56.
```

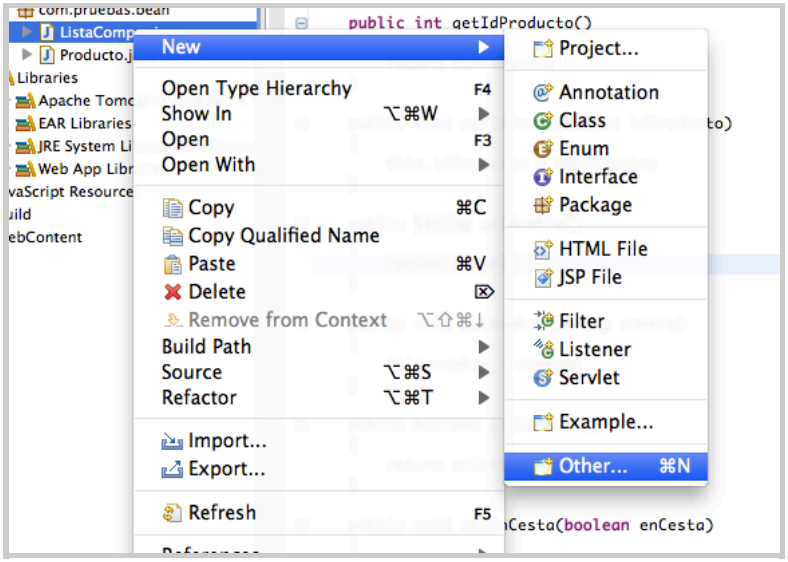
El código correspondiente a Producto.java es

```
view plain  print  ?
01. package com.pruebas.bean;
02.
03. public class Producto {
04.
05.     int idProducto;
06.     String nombre;
07.     boolean enCesta;
08.
09.     public int getIdProducto()
10.     {
11.         return idProducto;
12.     }
13.
14.     public void setIdProducto(int idProducto)
15.     {
16.         this.idProducto = idProducto;
17.     }
18.
19.     public String getNombre()
20.     {
21.         return nombre;
22.     }
23.
24.     public void setNombre(String nombre)
25.     {
26.         this.nombre = nombre;
27.     }
28.
29.     public boolean getEnCesta()
30.     {
31.         return enCesta;
32.     }
33.
34.     public void setEnCesta(boolean enCesta)
35.     {
36.         this.enCesta = enCesta;
37.     }
38.
39. }
```

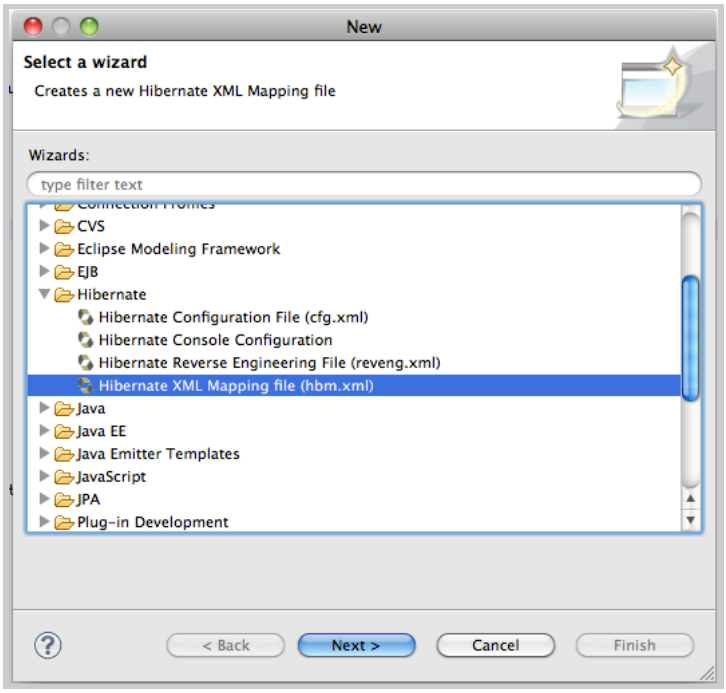
Vemos como la clase ListaCompra va a tener una propiedad "List" en donde va a almacenar todos los productos que pertenecen a una lista. Y si os fij´is, veréis que la clase Producto.java, no tiene ninguna propiedad para referirse al padre, aunque en el modelo de datos la tabla en la que mapearemos la clase Producto si debe tener una cláve foránea que obtiene de la tabla ListaCompra.

Para realizar el mapeo entre los objetos del proyecto y la base de datos en la que guardaremos la información tenemos que crear unos archivos para indicar de que forma queremos que se relaciones nuestra base de datos con nuestros objetos de la aplicación.

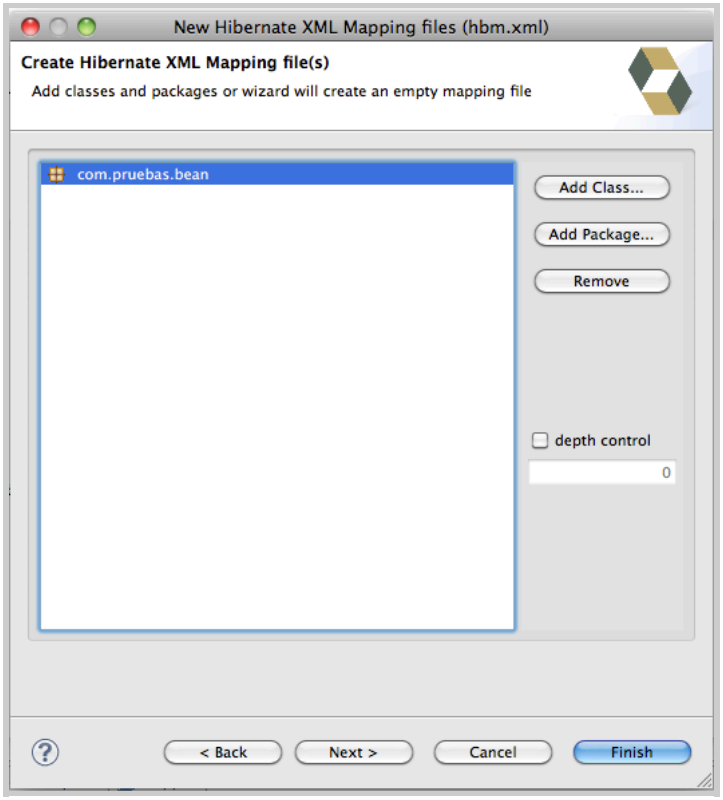
Para crear estos archivos lo que hacemos es clickear con el botón derecho en el proyecto y entonces en "New" -> "Other" como se muestra a continuación



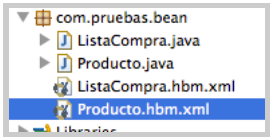
y seleccionamos Hibernate XML Mapping file (hbm.xml)



en la ventana que nos aparece, seleccionamos "Add package" e incluimos el paquete com.pruebas.bean que es donde se encuentran las dos clases que queremos mapear con Hibernate.



Pulsamos en "Finish" y nos creará los dos archivos de mapeado.



Vamos a modificar un poco estos archivos.

Archivo creado por el asistente.

```
view plain  print  ?
01.  <?xml version="1.0"?>
02.  <!DOCTYPE Hibernate-mapping PUBLIC "-
     //Hibernate/Hibernate Mapping DTD 3.0//EN"
03.  "http://Hibernate.sourceforge.net/Hibernate-
     mapping-3.0.dtd">
04.  <!-- Generated 17-mar-
     2011 15:13:50 by Hibernate Tools 3.3.0.GA --
     >
05.  <Hibernate-mapping>
06.      <class name="com.pruebas.bean.ListaCompra"
07.          <id name="idListaCompra" type="int">
08.              <column name="IDLISTACOMPRA" />
09.              <generator class="assigned" />
10.          </id>
11.          <property name="nombre" type="java.lanc
12.              <column name="NOMBRE" />
13.          </property>
14.          <property name="fecha" type="java.lang.
15.              <column name="FECHA" />
16.          </property>
17.          <list name="productos" inverse="false"
18.              <key>
19.                  <column name="IDLISTACOMPRA" />
20.              </key>
21.              <list-index></list-index>
22.              <one-to-
     many class="com.pruebas.bean.Producto" />
23.          </list>
24.      </class>
25.  </Hibernate-mapping>
```

Archivo final

```
view plain  print  ?
01.  <?xml version="1.0"?>
02.  <!DOCTYPE Hibernate-mapping PUBLIC "-
     //Hibernate/Hibernate Mapping DTD 3.0//EN"
03.  "http://Hibernate.sourceforge.net/Hibernate-
     mapping-3.0.dtd">
04.  <!-- Generated 25-feb-
     2011 18:12:04 by Hibernate Tools 3.3.0.GA --
     >
05.  <Hibernate-mapping>
06.     <class name="com.pruebas.bean.ListaCompra" t
07.          <id name="idListaCompra" type="int" >
08.              <column name="IDLISTACOMPRA" />
09.              <generator class="increment" />
10.          </id>
11.          <property name="nombre" type="java.lang.
12.              <column name="NOMBRE" />
13.          </property>
14.          <property name="fecha" type="java.lang.S
15.              <column name="FECHA" />
16.          </property>
17.
18.           <list name="productos" inverse="false"
19.               cascade="all">
20.              <key>
21.                  <column name="IDLISTACOMPRA" />
22.              </key>
23.              <list-index></list-index>
24.              <one-to-
     many class="com.pruebas.bean.Producto" />
25.          </list>
26.     </class>
27.  </Hibernate-mapping>
```

El primer cambio aparece al seleccionar un generador distinto al que nos pone el asistente. Al cambiar el generador de la clave de "assigned" a "increment" conseguimos que en vez de que tengamos que proporcionar nosotros una clave de forma manual, sea el gestor de la base de datos quien se encargue de gestionarla

El segundo cambio aparece en la línea 17, en donde hemos optado por cambiar el atributo lazy="true" por lazy="false". Este atributo se encarga de recuperar los objetos de la base de datos en demanda, esto quiere decir que se van a ir recuperando de la base de datos según vayan haciendo falta. En nuestro caso, teniendo en cuenta el contexto de la aplicación, un servicio web, nos interesa que cuando hagamos una petición, este nos devuelva todos los datos y podamos cerrar la conexión con el servidor.

En cuanto al segundo archivo de mapeado tenemos

Archivo creado por el asistente.

Archivo final

view plain print ?

```
01. <?xml version="1.0"?>
02. <!DOCTYPE Hibernate-mapping PUBLIC "-
    //Hibernate/Hibernate Mapping DTD 3.0//EN"
03. "http://Hibernate.sourceforge.net/Hibernate-
    mapping-3.0.dtd">
04. <!-- Generated 17-mar-
    2011 15:13:50 by Hibernate Tools 3.3.0.GA --
    >
05. <Hibernate-mapping>
06.     <class name="com.pruebas.bean.Producto" tab
07.         <id name="idProducto" type="int">
08.             <column name="IDPRODUCTO" />
09.             <generator class="assigned" />
10.         </id>
11.         <property name="nombre" type="java.lang
12.             <column name="NOMBRE" />
13.         </property>
14.         <property name="enCesta" type="boolean"
15.             <column name="ENCESTA" />
16.         </property>
17.     </class>
18. </Hibernate-mapping>
```

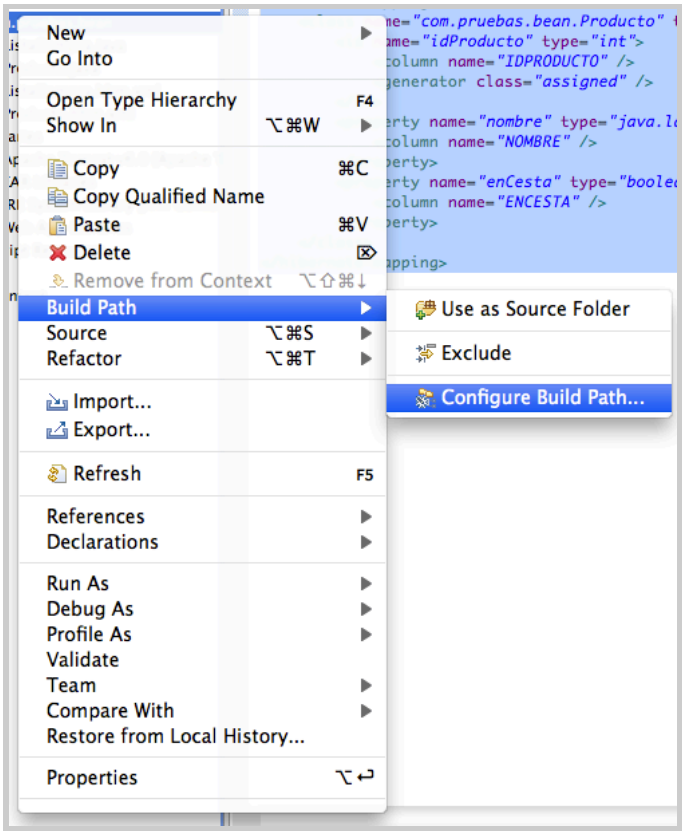
view plain print ?

```
01. <?xml version="1.0"?>
02. <!DOCTYPE Hibernate-mapping PUBLIC "-
    //Hibernate/Hibernate Mapping DTD 3.0//EN"
03. "http://Hibernate.sourceforge.net/Hibernate-
    mapping-3.0.dtd">
04. <!-- Generated 25-feb-
    2011 18:12:04 by Hibernate Tools 3.3.0.GA --
    >
05. <Hibernate-mapping>
06.     <class name="com.pruebas.bean.Producto" tab
07.         <id name="idProducto" type="int">
08.             <column name="IDPRODUCTO" />
09.             <generator class="increment" />
10.         </id>
11.         <property name="nombre" type="java.lang
12.             <column name="NOMBRE" />
13.         </property>
14.         <property name="enCesta" type="boolean"
15.             <column name="ENCESTA" />
16.         </property>
17.
18.     </class>
19. </Hibernate-mapping>
```

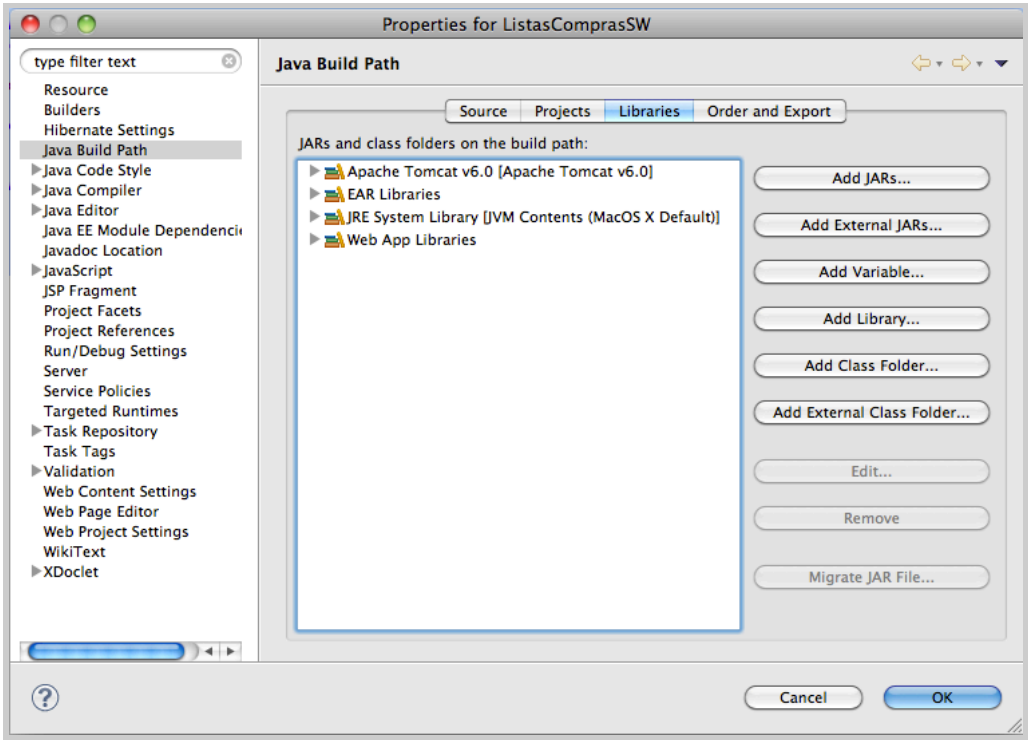
Como vemos solo hemos realizado un único cambio para cambiar el generator de la clave IDPRODUCTO igual que en el caso anterior.

Como vamos a utilizar Spring para inyectar las dependencias, vamos a aprovechar la configuración de Spring para configurar Hibernate y tener la configuración más centralizada.

Para esto, añadimos la librería de Spring a nuestro proyecto. Por lo tanto configuramos el Build Path de nuestro proyecto

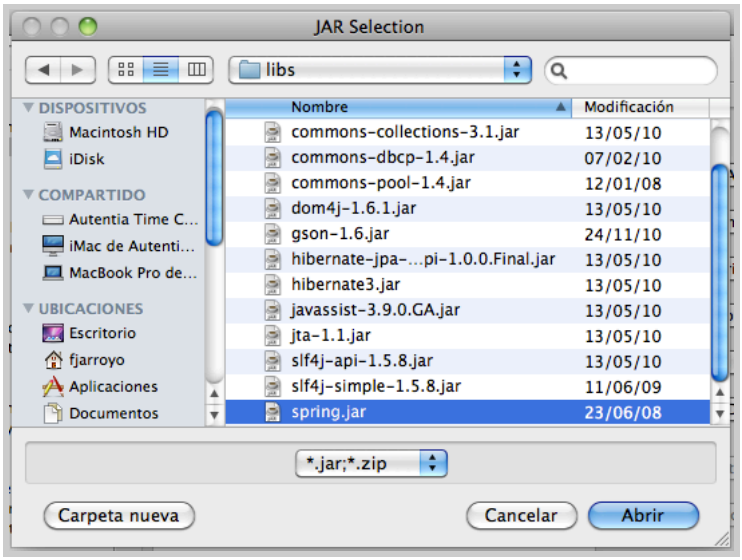


Y en "Libraries"

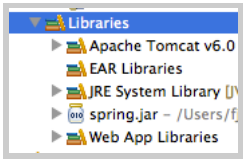


pulsamos en "Add external JAR"

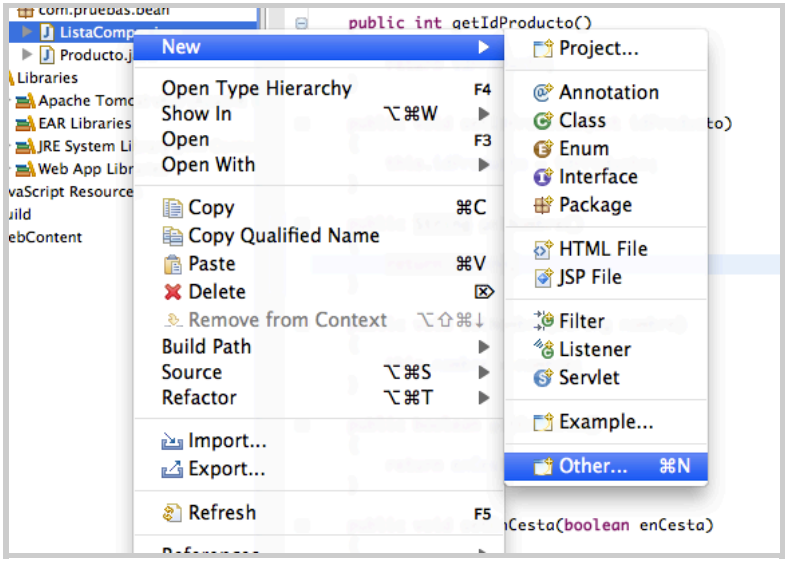
y seleccionamos el jar de Spring.



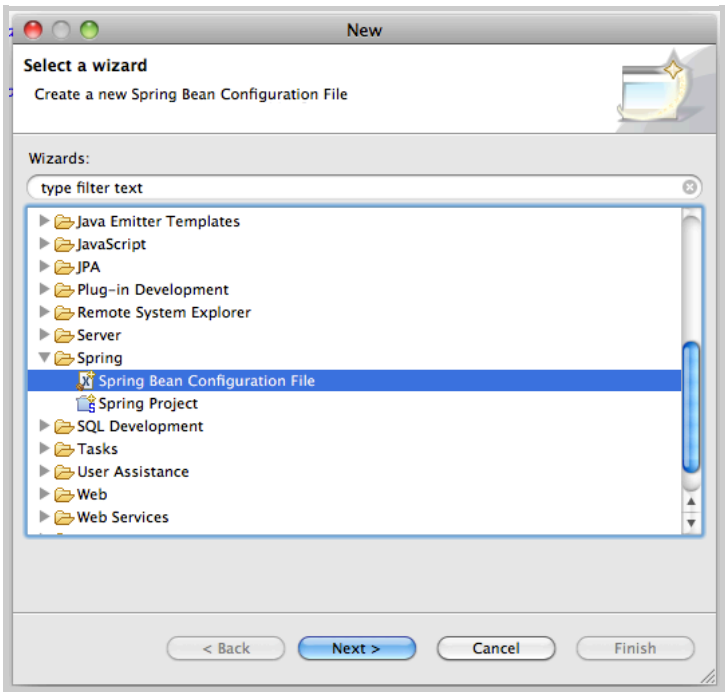
aceptamos y ya tendremos la librería en el classpath del proyecto



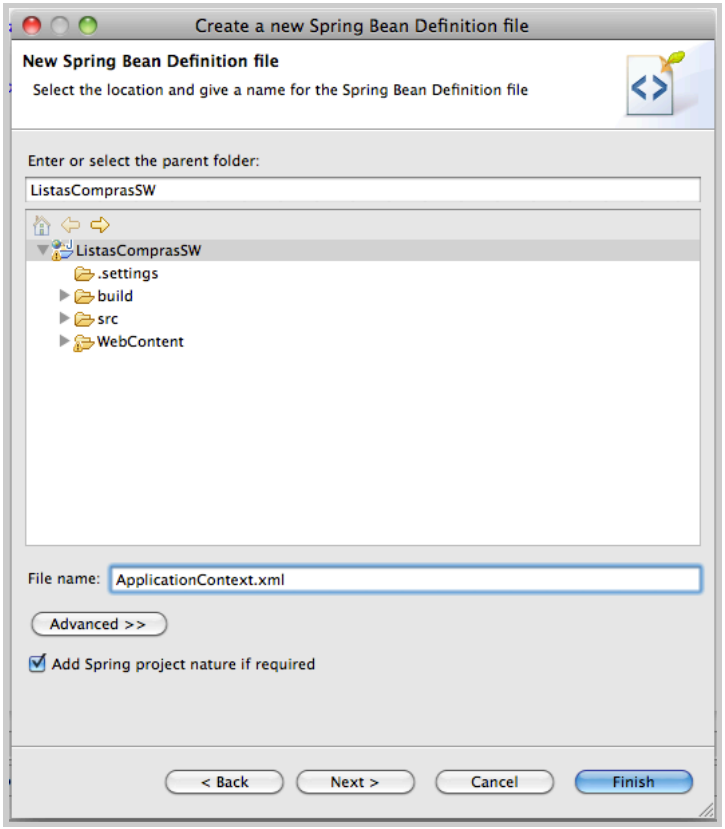
Ahora tenemos que crear el archivo de configuración de Spring. Para ello damos con e botón derecho del ratón y le damos a "New" -> "Other"



sólo que en esta ocasión debemos pinchar en "Spring Bean Configuration File"



Escogemos la carpeta del proyecto como "parent_folder" y le ponemos como nombre "ApplicationContext.xml"



Pinchamos en "Finish" y nos creará un archivo en la raíz del proyecto y nos añadirá otro apartado en nuestro proyecto con el nombre de "Spring Elements"



El contenido de ApplicationContext antes de que lo modifiquemos es:

Archivo creado por el asistente.

view plain print ?

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <beans xmlns="http://www.Springframework.org/sc
03.     xmlns:xsi="http://www.w3.org/2001/XMLSchema
    instance"
04.     xsi:schemaLocation="http://www.Springframew
    beans.xsd">
05.
06.
07. </beans>
```

Archivo con las modificaciones para Hibernate

view plain print ?

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <beans xmlns="http://www.Springframework.org/sc
03.     xmlns:xsi="http://www.w3.org/2001/XMLSchema
    instance"
04.     xsi:schemaLocation="http://www.Springframew
    beans-2.5.xsd">
05.
06.     <bean id="pgDataSource" class="org.Springfr
07.         <property name="driverClassName" value
08.         <property name="url" value="jdbc:postg
09.         <property name="username" value="postg
10.         <property name="password" value="prueb
11.      </bean>
12.
13.     <bean id="sessionFactory" class="org.Spring
14.         <property name="dataSource" ref="pgData
15.         <property name="mappingResources">
16.             <list>
17.                 <value>com/pruebas/bean/ListaCo
18.                 <value>com/pruebas/bean/Product
19.             </list>
20.         </property>
21.         <property name="HibernateProperties">
22.             <props>
23.                 <prop key="Hibernate.show_sql">
24.                 <prop key="Hibernate.dialect">c
25.             </props>
26.         </property>
27.     </bean>
28. </beans>
```

Como podemos ver ahora tenemos dos beans en el archivo de configuración.

En el primer bean, con id "pgDataSource", es donde configuramos nuestra conexión con la base de datos. Estamos indicando con la propiedad "driverClassName" que utilice el driver "org.postgresql.Driver" para realizar la conexión con nuestra base de datos. Le indicamos donde está nuestra base de datos con la propiedad url, que en este caso es una base de datos local a la que se accede por el puerto 5432. Y le indicamos las credenciales para el inicio de sesión.

El segundo bean, con id "sessionFactory", es quien nos va a proporcionar a través de "HibernateDaoSupport" el acceso de a la base de datos sin que tengamos que encargarnos nosotros de manejar las sesiones de Hibernate. Indicamos mediante la propiedad "dataSource" que utilice la conexión que hemos configurado en el bean anterior. Con la propiedad "mappingResources" indicamos donde se encuentran los archivos de mapeo que hemos creado anteriormente para que Hibernate pueda relacionar los objetos de la aplicación con las tablas de la base de datos. Y por último, configuramos la propiedad "Hibernate.show_sql" a true, para que nos vaya mostrando por consola las operaciones que va realizando (y así poder depurar con más facilidad).

A continuación nos creamos una interfaz para establecer las operaciones básicas que tiene que realizar un Dao para realizar la

persistencia de nuestros objetos. Tiene los métodos básicos de alta, baja y modificaciones. La interfaz es:

```
view plain  print  ?
01. package com.pruebas.dao;
02.
03. import java.util.List;
04.
05. import com.pruebas.bean.ListaCompra;
06.
07. public interface IDao
08. {
09.     public void save(ListaCompra listaCompra);
10.     public void saveOrUpdate(ListaCompra listaCompra);
11.     public void saveOrUpdateALl(ListaCompra[] listasCompra);
12.     public List<ListaCompra> All();
13.     public ListaCompra getListaCompra(int id);
14.     public void delete(int id);
15.
16. }
```

Vemos en la línea 5 que se está importando "com.pruebas.bean.ListaCompra". No es buena idea importar una clase en una interfaz de este estilo que trata de establecer las operaciones básicas de nuesto "Data Source Object", pero creemos que simplifica el entendimiento para este tutorial. Una buena práctica habría sido utilizar genéricos para eliminar el acoplamiento.

Y a continuación la clase que implementa esta interfaz

```
view plain  print  ?
01. package com.pruebas.dao;
02.
03. import java.util.List;
04.
05. import org.Springframework.orm.Hibernate3.support.HibernateDaoSupport;
06.
07. import com.pruebas.bean.ListaCompra;
08.
09. public class HibernateDaoImpl extends HibernateDaoSupport implements IDao{
10.
11.     public List<ListaCompra> All() {
12.         return (List<ListaCompra>)getHibernateTemplate().loadAll(ListaCompra.class);
13.     }
14.
15.     @Override
16.     public void delete(int id)
17.     {
18.         Object record = getHibernateTemplate().get(ListaCompra.class, id);
19.         getHibernateTemplate().delete(record);
20.     }
21.
22.     public ListaCompra getListaCompra(int id)
23.     {
24.         return (ListaCompra) getHibernateTemplate().get(ListaCompra.class, id);
25.     }
26.
27.     @Override
28.     public void saveOrUpdate(ListaCompra listaCompra)
29.     {
30.         // TODO Auto-generated method stub
31.         getHibernateTemplate().saveOrUpdate(listaCompra);
32.     }
33.
34.     @Override
35.     public void saveOrUpdateALl(ListaCompra[] listasCompra)
36.     {
37.         for(int i=0; i < listasCompra.length; i++)
38.         {
39.             getHibernateTemplate().saveOrUpdate(listasCompra[i]);
40.         }
41.     }
42.
43.     @Override
44.     public void save(ListaCompra listaCompra)
45.     {
46.         // TODO Auto-generated method stub
47.         getHibernateTemplate().save(listaCompra);
48.     }
49.
50. }
```

Vemos que además de implementar la interfaz "Dao", además hereda de la clase "HibernateDaoSupport". Esta es la manera que tiene Spring de darnos soporte a Hibernate. Mediante la configuración que hemos hecho en "ApplicationContext.xml" y heredando de "HibernateDaoSupport" podemos obtener una sesión de Hibernate para realizar la persistencia de nuestros objetos llamando al método "getHibernateTemplate()".

Además vamos a necesitar poder serializar y deserializar los objetos que tengamos persistidos, ya que estos van a viajar a través de la Internet. Para lograr este objetivo, vamos a utilizar una librería llamada Gson que va a realizar el trabajo por nosotros. Esta librería la podéis descargar de http://code.google.com/p/google-gson/downloads/detail?name=google-gson-1.6-release.zip. Una vez descargada, la tenéis que descomprimir y añadir al classpath del proyecto.

Incluimos en el proyecto dos archivos más, ISerializador.java y SerializadorImp.java,

ISerializador.java

```
view plain  print  ?
01. package com.pruebas.util;
02.
03. public interface ISerializador
04. {
05.
06.     public String Serializar(Object object);
07.     public <T> T Deserializar(String object, Class<T> clase);
08.
09. }
```

SerializadorImp.java

```
view plain  print  ?
01. package com.pruebas.util;
02.
03. import com.google.gson.Gson;
04.
05. public class SerializadorImp <T> implements ISerializador{
06.
07.     private static Gson gson = null;
08.
09.     public SerializadorImp()
10.     {
11.         gson = new Gson();
12.     }
13.
14.     @Override
15.     public String Serializar(Object objecto)
16.     {
17.         // TODO Auto-generated method stub
18.         return gson.toJson(objecto);
19.     }
20.
21.     @Override
22.     public <T> T Deserializar(String object,Class<T> clase)
23.     {
24.         // TODO Auto-generated method stub
25.         return (T)gson.fromJson(object, clase);
26.     }
27.
28. }
```

Como vemos, lo único que hace es recubrir al objeto gson para realizar las serializaciones y deserializaciones

A continuación, creamos un controlador que se encargue de gestionar los parámetros que recibimos de la vista y la persistencia de los datos.

```
view plain  print  ?
01. package com.pruebas.controller;
02.
03. import java.util.List;
04.
05. import com.pruebas.bean.ListaCompra;
06. import com.pruebas.dao.IDao;
07. import com.pruebas.util.ISerializador;
08.
09. public class ListaComprasController {
10.
11.     public IDao dao = null;
12.     public ISerializador serializador = null;
13.
14.     public ListaComprasController()
15.     {
16.     }
17.
18.     public void setIDao(IDao dao)
19.     {
20.         this.dao = dao;
21.     }
22.
23.     public void setISerializador(ISerializador serializador)
24.     {
25.         this.serializador = serializador;
26.     }
27.
28.     public String getListasCompras()
29.     {
30.         List<ListaCompra> listasCompras = dao.All();
31.         return serializador.Serializar(listasCompras);
32.     }
33.
34.     public String getLista(int idLista)
35.     {
36.
37.         ListaCompra listaCompra = dao.getListaCompra(idLista);
38.         return serializador.Serializar(listaCompra);
39.     }
40.
41.     public void nuevaLista(String jsonListaCompra)
42.     {
43.         ListaCompra listaCompra = serializador.Deserializar(jsonListaCompra, ListaCompra.class);
44.         dao.save(listaCompra);
45.     }
46.
47.     public void guardarLista(String jsonListaCompra)
48.     {
49.         ListaCompra listaCompra = serializador.Deserializar(jsonListaCompra, ListaCompra.class);
50.         dao.saveOrUpdate(listaCompra);
51.     }
52.
53.     public void borrarLista(String jsonListaCompra)
54.     {
55.         ListaCompra listaCompra = serializador.Deserializar(jsonListaCompra, ListaCompra.class);
56.         dao.delete(listaCompra.getIdListaCompra());
57.     }
58. }
```

Lo importante de esta clase es que como vemos en la línea 11 y 12, tenemos dos propiedades, "dao" y "serializador" que utilizamos varias veces en los métodos de la clase sin instanciarlos antes. Esto es posible porque dejamos que Spring haga la inyección de dependencias. Para conseguir esto, modificamos el archivo "ApplicationContext.xml" e incluimos estas líneas.

```
view plain  print  ?
01.  <bean id="serializador" class="com.pruebas.util.SerializadorImp">
02.  </bean>
03.
04.  <bean id="dao" class="com.pruebas.dao.HibernateDaoImpl">
05.      <property name="sessionFactory"><ref bean="sessionFactory"/></property>
06.  </bean>
07.
08.  <bean id="listaComprasController" class="com.pruebas.controller.ListaComprasController">
09.      <property name="IDao" ref="dao"/>
10.      <property name="ISerializador" ref="serializador"/>
11.  </bean>
```

Como vemos, hemos creado tres "beans". El primero, "serializador", es en el que indicamos que implementación queremos para el serializador, en este caso va a pertenecer a la clase "com.pruebas.util.SerializadorImp". El segundo, "dao", es el objeto donde realizaremos la persistencia. Y el tercero, "listaComprasController", es el controlador de la vista, y vemos como inicializamos dos propiedades "IDao" y "ISerializador", con esto conseguimos que en el momento de instanciar el controlador, Spring se encarge de inicializar esos objetos.

A continuación creamos una interfaz que exponga los métodos que va a publicar el servicio web.

```
view plain  print  ?
01.  package com.pruebas.service;
02.
03.  public interface IListaComprasSW {
04.
05.      public void nuevaLista(String jsonListaCompra);
06.      public void guardarLista(String jsonListaCompra);
07.      public void borrarLista(String jsonListaCompra);
08.      public String getListas();
09.      public String getLista(int idLista);
10.  }
```

Como vemos, vamos a publicar 5 métodos con los que podremos crear nuevas listas de la compra, guardar, y obtener listas. Las típicas funcionalidades de un gestor

Ahora pasamriamos a crear la clase que implemente está interfaz. Ademas de implementar los métodos de la interfaz, añadimos una propiedad "ListaComprasController" que más adelante vemos como será inicializada a través del constructor gracias a Spring.

```
package com.pruebas.service;

import com.pruebas.controller.ListaComprasController;

public class ListaCompraSW implements IListaComprasSW {

    private ListaComprasController controller;

    public ListaCompraSW(ListaComprasController controller)
    {
        this.controller = controller;
    }

    public ListaComprasController getListaComprasController()

    public void setListaComprasController(ListaComprasController listaComprasController)

    public void borrarLista(String jsonListaCompra) {

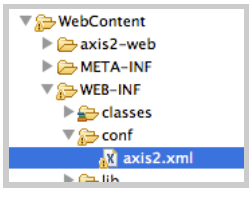
    public String getLista(int idLista) {

    public String getListas() {

    public void guardarLista(String jsonListaCompra)

    public void nuevaLista(String jsonListaCompra) {
}
```

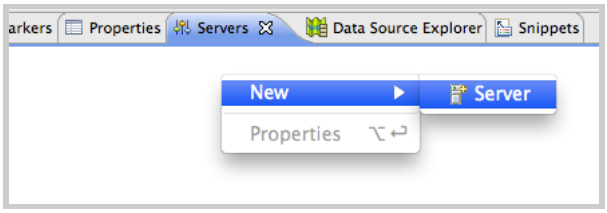
Antes de desplegar nuestro servicio web, debemos añadir un pará metro a la configuración de Axis2 para permitir servicios rest. El archivo que debemos modificar está en "WebContent\WEB-INF\conf\axis2.xml.



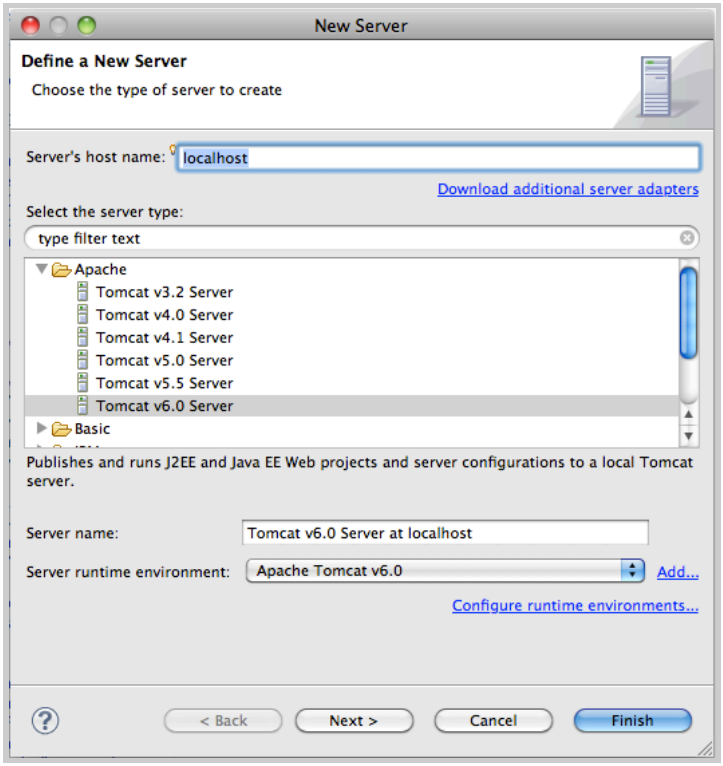
y la línea que hay que añadir es

```
view plain  print  ?
01.  <parameter name="enableREST" locked="false">true</parameter>
```

Una vez que tenemos hecho todo esto, vamos a publicar nuestro servicio web. Para esto tenemos que configurar Eclipse para que nos despliegue el servicio web en el servidor TomCat que nos hemos bajado. Entonces pinchamos en la pestaña "Server" en Eclipse y clickeamos con el botón derecho y le damos a "New".

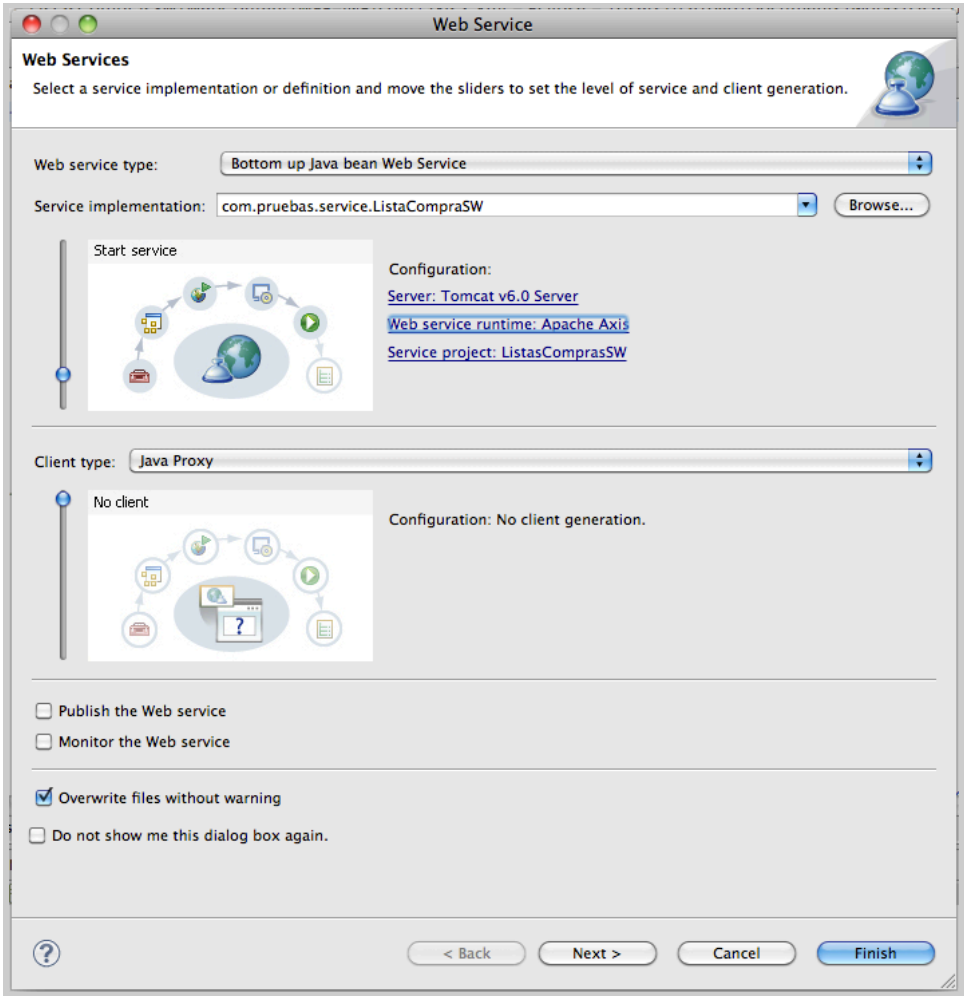


Dejamos las opciones como nos aparecen y le damos a "Finish"

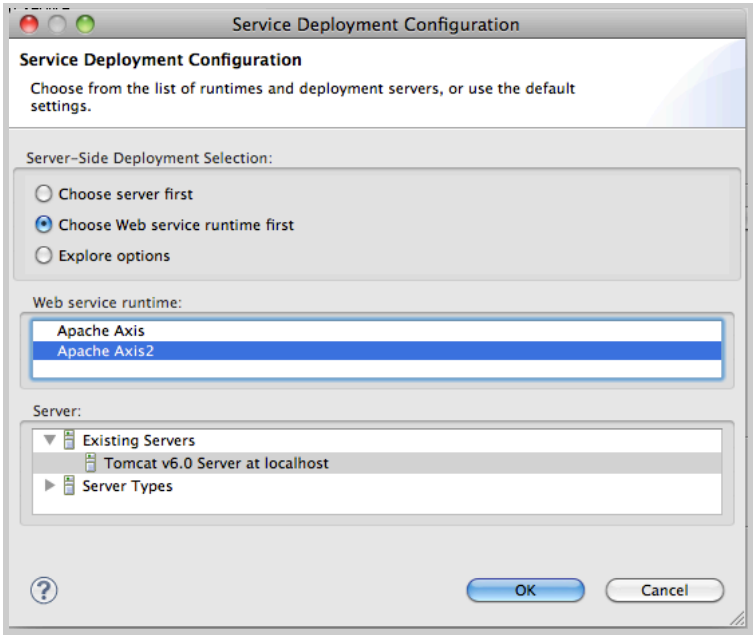


y ya tendremos un servidor en donde desplegar nuestro servicio web

Ahora lo que tenemos que hacer es indicar que clase es la que expone los métodos web. Para hacerlo, nos ponemos encima de la clase que contiene los métodos y pulsamos con el botón derecho y clickeamos en "WebServices" -> "Create Web Services". Nos aparece una pantalla como la siguiente:

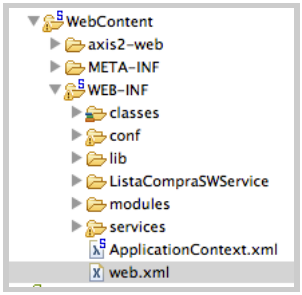


Si nos fijamos, por defecto nos ha configurado para ejecutar el servicio con Axis, pero a nosotros nos interesa que sea con Axis2, por lo que clickeamos en "Web Service Runtime: Apache Axis" y en la ventana que aparece seleccionamos Axis2, le damos a "Ok" y a "Finish" con lo que nos levantará Tomcat y nos desplegará nuestro servicio web.



Como estamos utilizando Spring dentro de Axis2, Spring no es capaz de cargar su contexto, con lo que necesitamos modificar un par de archivos más.

Primero modificaremos el archivo web.xml para agregar un listener y poder pasar el contexto de Spring, además de indicarle donde se va a encontrar nuestro contexto (el archivo applicationContext.xml).



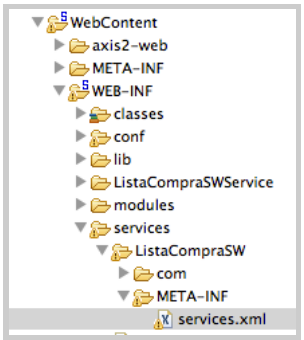
Para ello añadimos las líneas que hay a continuación en web.xml

```
view plain  print  ?
01. <listener>
02.     <listener-class>org.Springframework.web.context.ContextLoaderListener</listener-class>
03.     </listener>
04.     <context-param>
05.       <param-name>contextConfigLocation</param-name>
06.       <param-value>/WEB-INF/ApplicationContext.xml</param-value>
07.     </context-param>
```

y para que Spring nos inyecte la dependencia de nuestra clase que se encarga de recibir las peticiones Http, añadimos a "ApplicationContext.xml" las siguientes líneas.

```
view plain  print  ?
01. <bean id="listaCompraSW" class="com.pruebas.service.ListaCompraSW">
02.         <constructor-arg><ref bean="listaComprasController"/></constructor-arg>
03.     </bean>
```

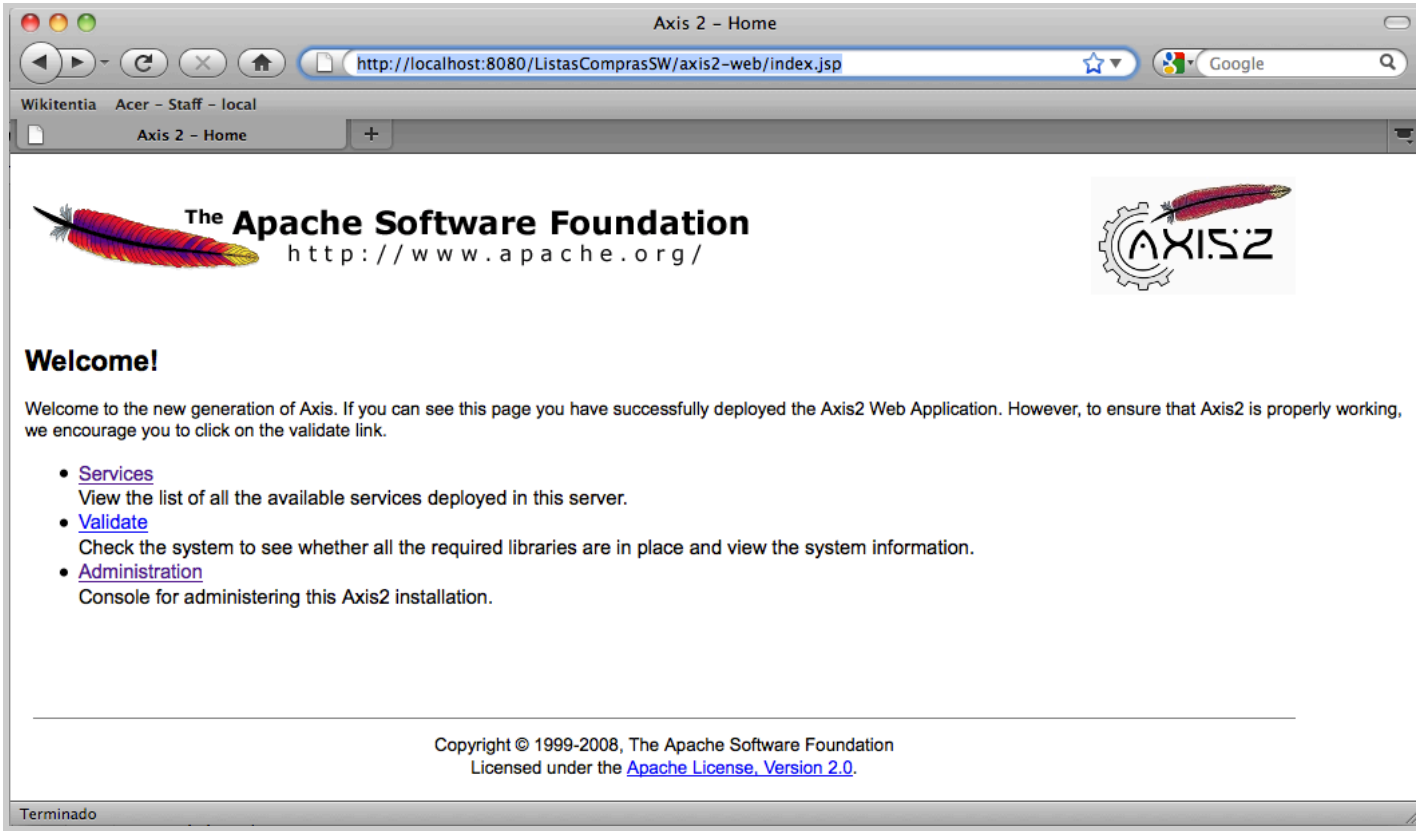
Por último debemos modificar un archivo más. Para ello, abrimos el archivo que se encuentra en "WebContent" -> "WEB-INF" -> "services" -> "ListaCompraSW" -> "META-INF" -> "services.xml"



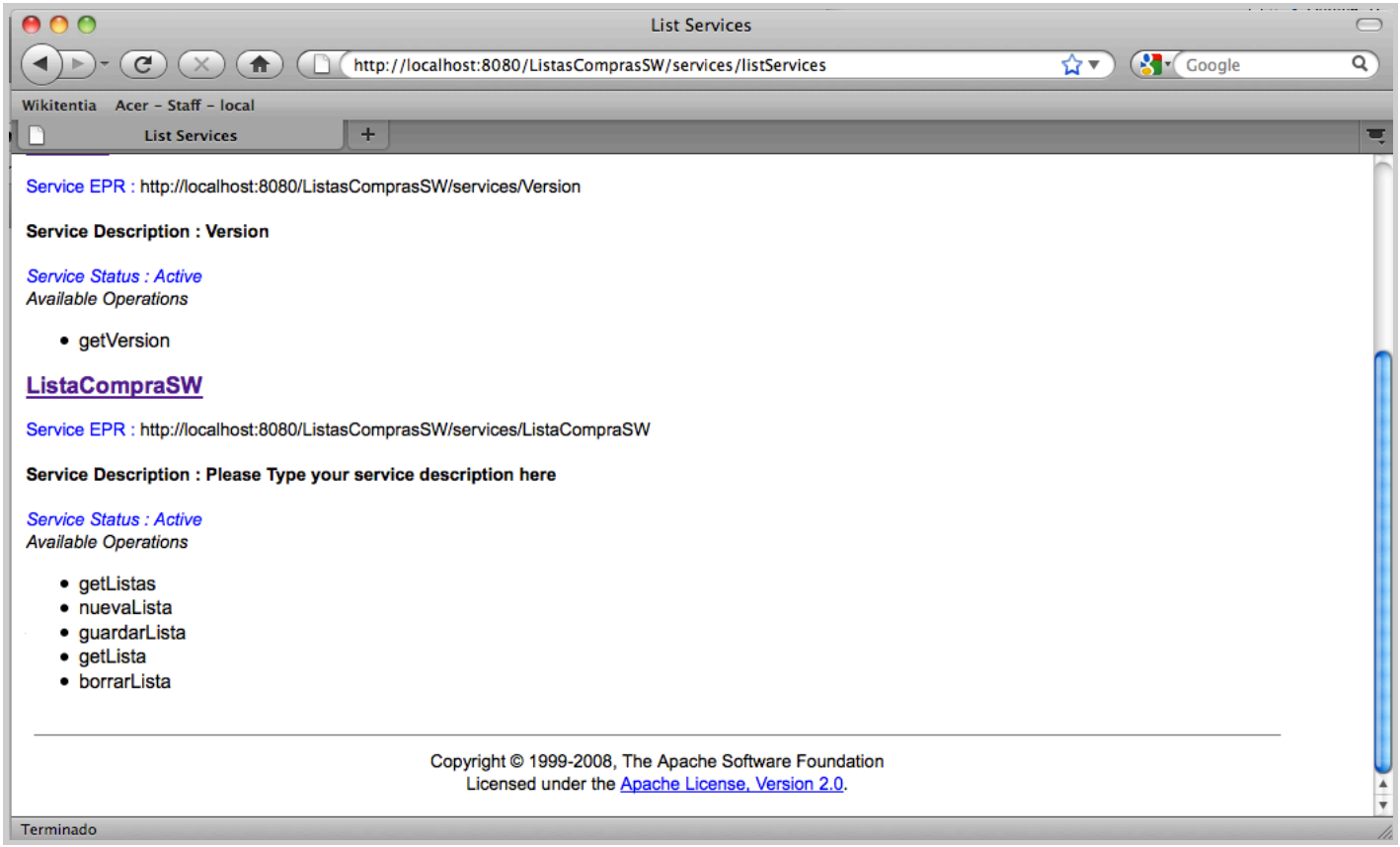
y añadimos las líneas

```
view plain  print  ?
01.  <parameter name="ServiceObjectSupplier">org.apache.axis2.extensions.Spring.receivers.SpringServletCont
02.     <parameter name="SpringBeanName">listaCompraSW</parameter>
03.     <operation name="getValue">
04.         <messageReceiver class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver"/>
05.     </operation>
```

Si ahora abrimos un navegador y nos dirigimos a http://localhost:8080/ListasComprasSW/axis2-web/index.jsp deberíamos ver algo como esto:



y si pinchamos en Services deberemos ver listado nuestro servicio



Con lo que ya tenemos nuestro servicio web ;)

5. Crear el cliente

Ahora vamos a crear un cliente sencillo en Android que va a consumir nuestro servicio web. La aplicación va a constar de dos activities, cada uno con un listView.

El primer activity es la vista principal en donde podemos ver las listas de la compra recuperadas por la aplicación. Además de poder añadir nuevas listas y especificar la dirección IP en donde se encuentra nuestro servidor.

El segundo activiy contiene un listado de los productos que tiene una lista de la compra.

Main.java

```
package com.pruebas;

import java.util.List;

public class Main extends Activity implements OnItemClickListener, OnDismissListener  {

    List<ListaCompra> listasCompras = null;
    ListaComprasController controller = null;
    public static Context context;

    /** Called when the activity is first created. */
    public void onCreate(Bundle savedInstanceState) {

    public void onResume(){

    public void onRestart(){

    public void onItemClick(AdapterView<?> arg0, View arg1, int arg2, long arg3) {

    public void onDismiss(DialogInterface arg0) {

    public boolean onCreateOptionsMenu(Menu menu) {

    public boolean onOptionsItemSelected(MenuItem item) {

    private void mostrarDialogDireccionIP(){

    private void actualizarListas(){


}
```

ListaProductos.java

```
package com.pruebas;

import com.pruebas.bean.ListaCompra;

public class ListaProductos extends Activity {

    ListaCompra listaCompra = null;

    public void onCreate(Bundle savedInstanceState) {

    public boolean onCreateOptionsMenu(Menu menu) {

    public boolean onOptionsItemSelected(MenuItem item) {

}
```

Para rellenar estos los listView de los activities hemos creado dos adapters que se encargaran de rellenar los listview

ListaComprasAdapter.java

```java
package com.pruebas;

import java.util.List;

public class ListaComprasAdapter extends BaseAdapter{

    private Context context = null;
    private List<ListaCompra> listasCompras = null;

    public ListaComprasAdapter(Context context, List<ListaCompra> listasCompras)


    public int getCount() {

    public Object getItem(int arg0) {

    public long getItemId(int arg0) {

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {

        RelativeLayout rowLayout;

        // TODO Auto-generated method stub
        if(convertView == null)
        {
            ListaCompra listaCompra = listasCompras.get(position);
            rowLayout = (RelativeLayout)LayoutInflater.from(context).inflate(R.layout.rowlistacompra,
            TextView tvTituloListaCompra = (TextView) rowLayout.findViewById(R.id.tvTituloListaCompra)
            tvTituloListaCompra.setText(listaCompra.getNombre());
            TextView tvFechaListaCompra = (TextView) rowLayout.findViewById(R.id.tvFechaListaCompra);
            tvFechaListaCompra.setText(listaCompra.getFecha());
            convertView = rowLayout;
        }
        return convertView;
    }

}
```

ListaProductosAdapter.java

```java
package com.pruebas;

import android.content.Context;

public class ListaProductosAdapter extends BaseAdapter implements OnClickListener{

    private Context context = null;
    private ListaCompra listaCompra = null;

    public ListaProductosAdapter(Context context, ListaCompra listaCompra)

    public int getCount() {

    public Object getItem(int arg0) {

    public long getItemId(int arg0) {

    @Override
    public View getView(int position, View convertView, ViewGroup parent)
    {
        RelativeLayout rowLayout;

        // TODO Auto-generated method stub
        if(convertView == null)
        {
            rowLayout = (RelativeLayout)LayoutInflater.from(context).inflate(R.layout.rowlistaproducto
            Producto producto = (Producto)getItem(position);

            TextView tvNombreProducto = (TextView)rowLayout.findViewById(R.id.tvNombreProducto);
            tvNombreProducto.setText(producto.getNombre());

            CheckBox cbEnCesta = (CheckBox)rowLayout.findViewById(R.id.cbEnCesta);
            cbEnCesta.setId(position);
            cbEnCesta.setChecked(producto.getEnCesta());
            cbEnCesta.setOnClickListener(this);


            if((position % 2) == 0)
                rowLayout.setBackgroundColor(Color.parseColor("#ffffcd"));

            convertView = rowLayout;
        }

        return convertView;
    }

    public void onClick(View v) {
```

Para acceder al servicio web, que de lo que se trataba el tema en cuestión ;), Creamos nuestro "dao", pero esta vez en vez de usar Hibernate para acceder a una base de datos, utilizamos una conexión HTTP contra el servicio que hemos creado anteriormente

```java
package com.pruebas.dao;

import java.io.IOException;

public class Dao implements IListaComprasSW {

    Context context = null;
    String urlBase;

    public Dao(Context context)

    public String getUrlBase()

    public void setUrlBase(String url)

    public void borrarLista(String jsonListaCompra) {

    @Override
    public String getLista(int idLista) {

        String urlRequest = getUrlBase() + "getLista?idLista=" + idLista;
        String jsonListaCompra = "";

        try {
            jsonListaCompra = PeticionHttp.realizarPeticion(urlRequest);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            Log.e("ListaCompras", e.getMessage());
        }

        return jsonListaCompra;

    }

    public String getListas() {

    public void guardarLista(String jsonListaCompra) {

    public Long nuevaLista(String jsonListaCompra) {

}
```

La razón por la que implementa IListaCompraSW, es para asegurarnos que vamos a tener los mismos métodos que el servicio web que hemos implementado anteriormente.

También tendremos que modificar "AndroidManifest.xml" para pedir acceso a internet desde nuestra aplicación e indicarle que la aplicación consta de dos activities.

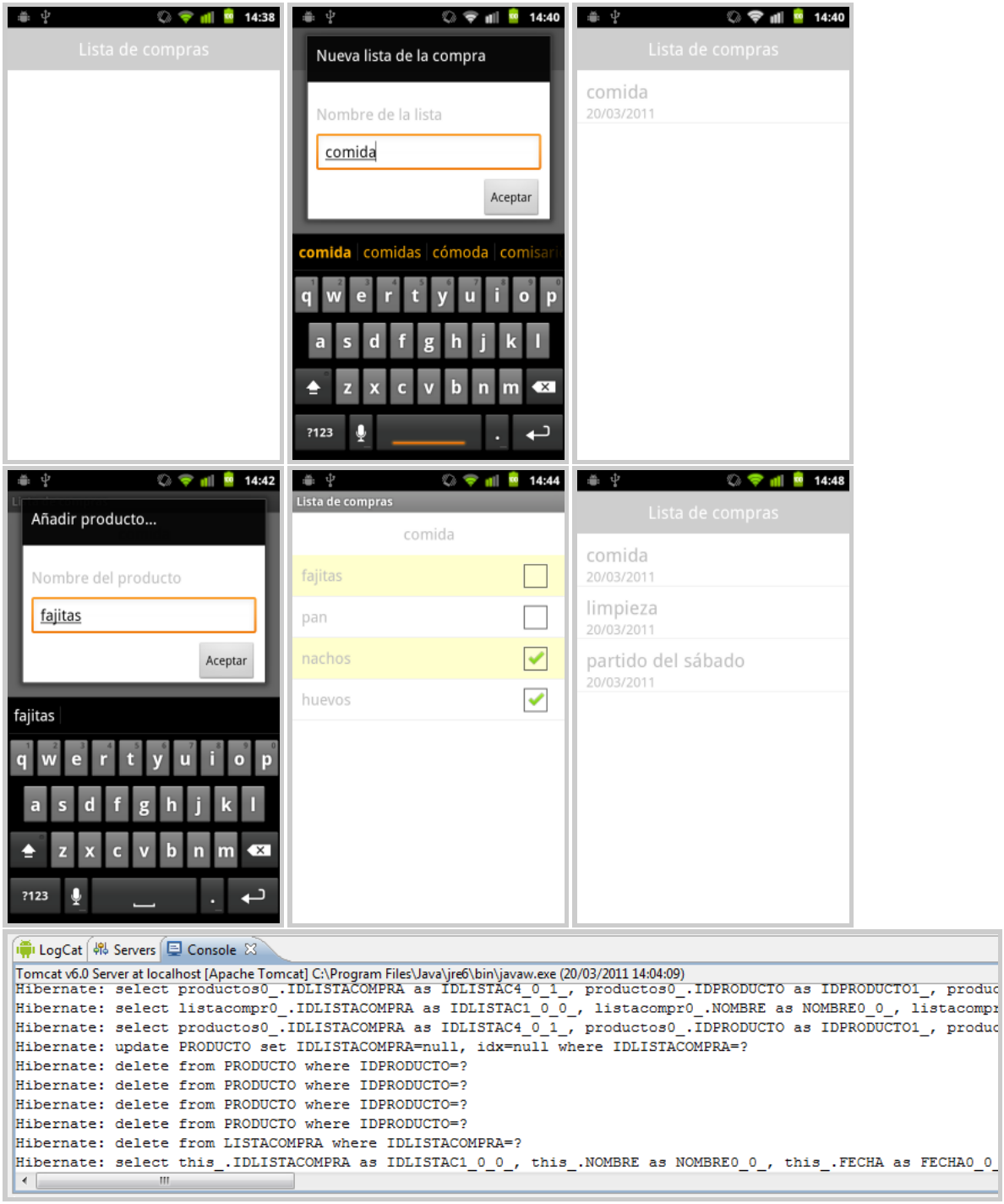
view plain print ?

```
01.     <?xml version="1.0" encoding="utf-8"?>
02.         <manifest xmlns:android="http://schemas.android.com/apk/res/android"
03.       package="com.pruebas"
04.       android:versionCode="1"
05.       android:versionName="0.1">
06.     <uses-sdk android:minSdkVersion="7" />
07.
08.     <application android:icon="@drawable/icon" android:label="@string/app_name">
09.         <activity android:name="Main"
10.                   android:label="@string/app_name"
11.                   android:theme="@android:style/Theme.NoTitleBar">
12.             <intent-filter>
13.                 <action android:name="android.intent.action.MAIN" />
14.                 <category android:name="android.intent.category.LAUNCHER" />
15.             </intent-filter>
16.         </activity>
17.
18.         <activity android:name=".ListaProductos"
19.                   android:label="@string/app_name">
20.         </activity>
21.     </application>
22.
23.     <uses-permission android:name="android.permission.INTERNET" />
24.
25. </manifest>
```

6. Conclusiones

Al usar servicios web REST, vemos que podemos acceder a esa información de una manera muy sencilla y ligera (a través de una petición HTTP), aspecto que nos viene muy bien si el cliente que va a consumir esos datos es un cliente ligero (un móvil en este caso).

A continuación os muestro unas capturas de pantalla de como funciona el invento :) y el código fuente de los dos proyectos :P



Si queréis descargaros el código fuente de los dos proyectos lo tenéis aquí

Anímate y coméntanos lo que pienses sobre este **TUTORIAL:**

Puedes opinar o comentar cualquier sugerencia que quieras comunicarnos sobre este tutorial; con tu ayuda, podemos ofrecerte un mejor servicio.

Enviar comentario

(Sólo para usuarios registrados)

» **Registrate** y accede a esta y otras ventajas «

COMENTARIOS

Esta obra está licenciada bajo licencia Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5

Copyright 2003-2011 © All Rights Reserved | Texto legal y condiciones de uso | Banners | Powered by Autentia | Contacto