Servidores Web

Sesión 3: Construcción, despliegue y prueba de aplicaciones web con Maven

Puntos a tratar

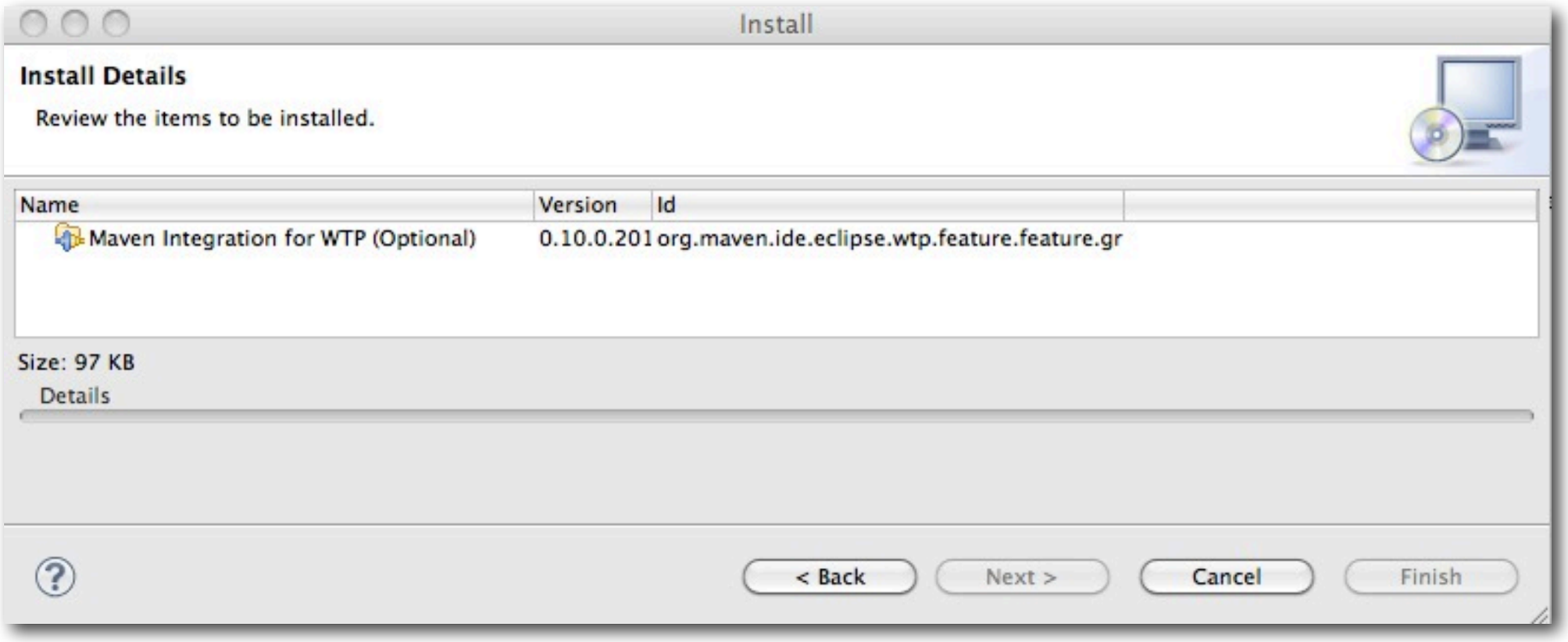
- **Proyectos web con Maven**
  - Peculiaridades
  - Trabajar con Maven y Eclipse
- Despliegue en el servidor
- Pruebas de aplicaciones web
  - Pruebas unitarias
  - Pruebas de integración y funcionales

Proyectos web con Maven

- Tienen una estructura de directorios distinta a la habitual. El empaquetado final debe respetar la estructura estándar que vimos en sesiones anteriores
- Se empaquetan en un .war, no un .jar
- Dependen (como mínimo) del API de servlets
- Se despliegan en un servidor
  - Necesitamos gestionar el arranque/parada del servidor para poder probar la aplicación

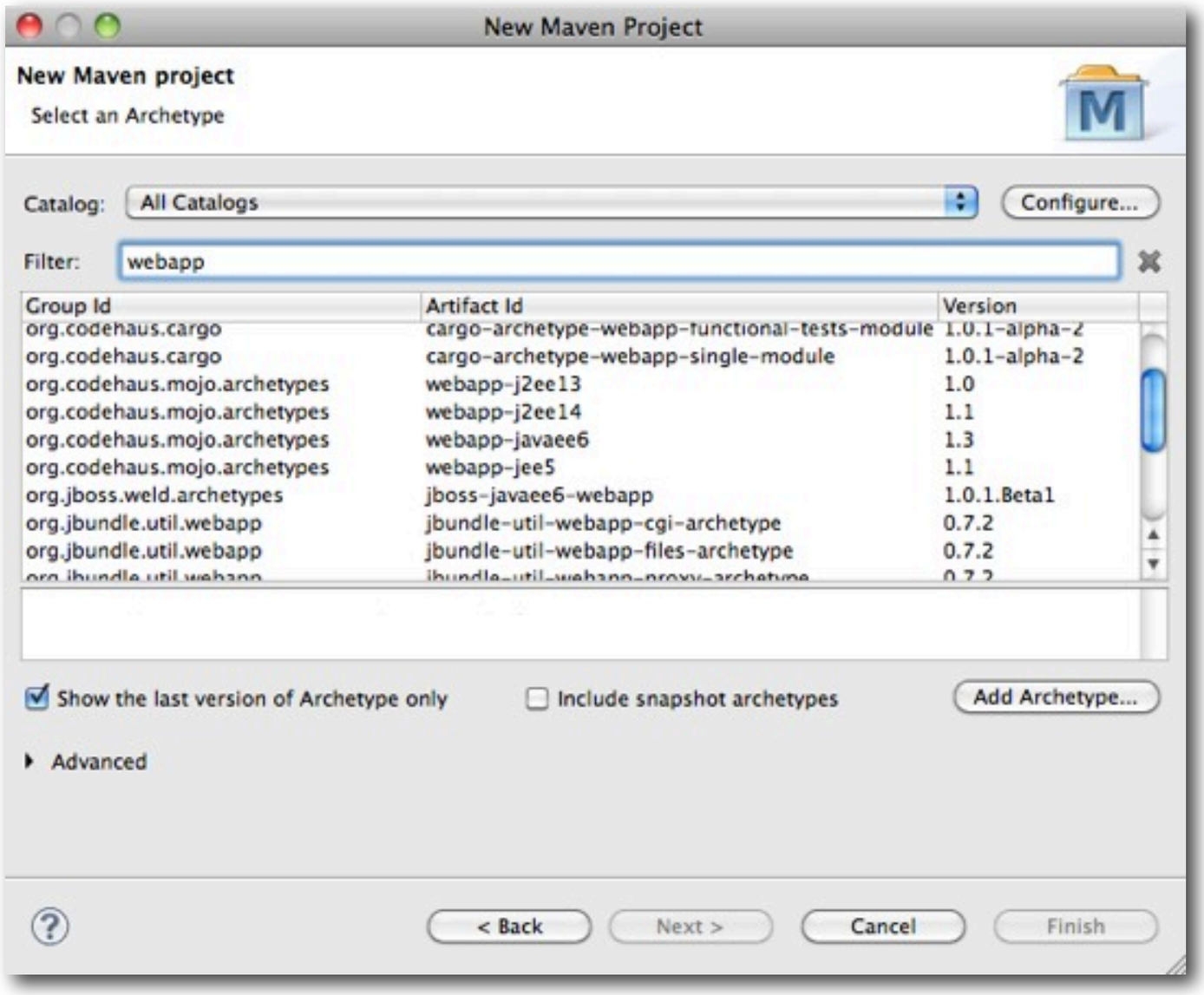
Proyectos web Maven con Eclipse

- Es necesario instalar el "**Maven Integration for WTP**" para poder desplegar en el servidor tanto desde Maven como desde Eclipse



Crear el proyecto: arquetipos

- Hay multitud de arquetipos para crear proyectos web de modo sencillo. El más básico es **`maven-archetype-webapp`**
  - Existen muchos otros para tecnologías específicas (Struts, Spring, AppFuse, EJB,...)



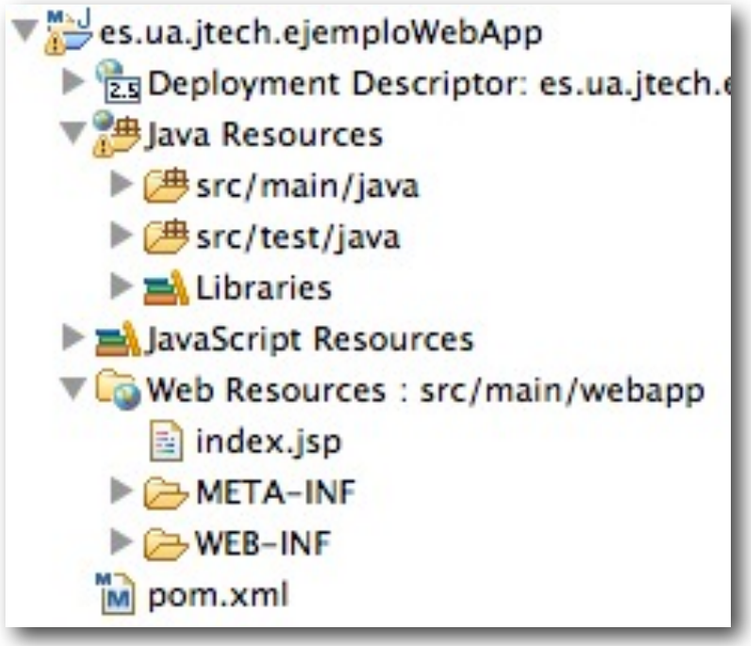
Particularidades del proyecto

- El empaquetado se hace en un **war**

```
<project xmlns=[...]>
  <modelVersion>4.0.0</modelVersion>
  <groupId>es.ua.jtech</groupId>
  <artifactId>EjemploWebMaven</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>EjemploWebMaven Maven Webapp</name>
  [...]
```

- Añadido a la **estructura de directorios**
  - aplic. web : `src/main/webapp`

    (toda la aplicación menos los fuentes Java)



Configurar compilador para Java 1.5

- Igual que en aplicaciones no web
  - Recordar que por defecto el plugin del compilador supone la 1.4

```
<plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
</plugins>
```

Dependencias

- Cualquier aplicación web **como mínimo dependerá del API de servlets** (si no es un “hola mundo”, claro)
- Si vamos a desarrollar tags de JSP propias también tendremos la dependencia del API JSP
  - Ambas son “**provided**”, ya que los JAR son necesarios para compilar pero ya vienen en el servidor

```
<dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
      <scope>provided</scope>
</dependency>
<dependency>
     <groupId>javax.servlet.jsp</groupId>
     <artifactId>jsp-api</artifactId>
     <version>2.0</version>
     <scope>provided</scope>
</dependency>
```

Maven y Eclipse simultáneamente

- **Eclipse** es más cómodo para editar, compilar, depurar o probar de modo informal, pero **Maven** es más apropiado para empaquetar o realizar baterías de pruebas
- En Maven podemos compilar y generar el war con el "goal" **install**

```
mvn clean install
```

- En Eclipse podemos seguir compilando automáticamente y desplegando con Run As > Run on Server como es habitual

Puntos a tratar

- Proyectos web con Maven
  - Peculiaridades
  - Trabajar con Maven y Eclipse
- **Despliegue en el servidor**
- Pruebas de aplicaciones web
  - Pruebas unitarias
  - Pruebas de integración y funcionales

Despliegue en el servidor

- Ya sabemos cómo desplegar aplicaciones desde Eclipse (Run on server)
- Desde Maven podemos hacer lo mismo, usando plugins adicionales
  - Cargo: **http://cargo.codehaus.org/**
  - Tomcat: **http://mojo.codehaus.org/tomcat-maven-plugin/**
  - Jetty: **http://docs.codehaus.org/display/JETTY/Maven+Jetty+Plugin**
- En general estos plugins permiten
  - Arrancar/parar el servidor
  - Cambiar su configuración
  - Desplegar/replegar aplicaciones

Uso del plugin “Cargo”

- Originalmente es un API Java al que se le ha añadido un plugin Maven
- El más potente
  - Permite controlar multitud de servidores distintos (Tomcat, Glassfish, JBoss, Weblogic,...)
  - Se pueden controlar servidores remotos o en local
  - Se puede especificar la configuración del servidor (una propia para Cargo independiente de la que tenga el servidor por defecto)
  - Incluso se puede bajar e instalar el servidor sobre la marcha

Configuración básica del plugin

- Y tan básica: si solo especificamos el nombre del plugin usará el servidor web por defecto (jetty)
  - Jetty es un servidor ligero que se puede usar empotrado (sin conexión “real” TCP/IP, útil para probar aplicaciones)

```
<plugin>
   <groupId>org.codehaus.cargo</groupId>
   <artifactId>cargo-maven2-plugin</artifactId>
</plugin>
```

Configuración para Tomcat

- Se especifica qué tipo de servidor es (*tomcat6x*) y dónde está instalado
- También un directorio temporal donde Cargo creará una copia de la estructura de dirs de Tomcat

```
<plugin>
   <groupId>org.codehaus.cargo</groupId>
   <artifactId>cargo-maven2-plugin</artifactId>
   <!-- configuración del plugin de cargo -->
   <configuration>
      <!-- configuración del contenedor -->
      <container>
         <containerId>tomcat6x</containerId>
         <home>/opt/apache-tomcat-6.0.29</home>
      </container>
      <!-- configuración del despliegue -->
      <configuration>
         <home>${project.build.directory}/tomcat6x</home>
      </configuration>
   </configuration>
</plugin>
```

Desplegar la aplicación

- Podemos arrancar el servidor con cargo:start.
  - Este objetivo no está vinculado a ninguna fase del ciclo de vida, por lo que hay que ejecutarlo manualmente, por ejemplo tras generar el war

```
mvn install cargo:start
```

  - Podemos parar el servidor con Ctrl-C o desde otra terminal con cargo:stop
- Por defecto el artefacto creado por el proyecto se despliega automáticamente al arrancar el servidor
  - Podemos desplegar un artefacto con cargo:deploy

Librerías comunes a varias aplicaciones

- Muy típico de frameworks o drivers JDBC
- Cada servidor tiene un directorio específico donde colocar los JAR. Cargo se encarga de los detalles
- Ejemplo: driver de MySQL
  - **Parte I**: especificar la dependencia de nuestra aplicación del JAR en cuestión

```
<dependencies>
[...]
   <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.13</version>
      <scope>provided</scope>
   </dependency>
[...]
</dependencies>
```

Librerías comunes a varias aplicaciones (II)

- **Parte II:** especificar la misma dependencia para el contenedor web (no hace falta la versión ya que referenciamos la anterior)

```
<plugin>
     <groupId>org.codehaus.cargo</groupId>
     <artifactId>cargo-maven2-plugin</artifactId>
     <configuration>
        <container>
           <containerId>tomcat6x</containerId>
           <home>/opt/apache-tomcat-6.0.29</home>
           <dependencies>
              <dependency>
                 <groupId>mysql</groupId>
                 <artifactId>mysql-connector-java</artifactId>
              </dependency>
           </dependencies>
        </container>
        <configuration>
            ${project.build.directory}/tomcat6x</home>
        </configuration>
     </configuration>
</plugin>
```

Puntos a tratar

- Proyectos web con Maven
  - Peculiaridades
  - Trabajar con Maven y Eclipse
- Despliegue en el servidor
- **Pruebas de aplicaciones web**
  - Pruebas unitarias
  - Pruebas de integración y funcionales

Pruebas en aplicaciones web

- Pruebas de la capa web (servlets y JSPs)
  - **Unitarias**: funcionan mejor en un contexto simulado (*mockups*). Herramientas: easymockups, servletunit
  - **Integración**: se prueban Servlets y JSPs en un contenedor web. Herramientas: Cactus
  - **Funcionales:** se prueba el funcionamiento de la aplicación (un caso de uso). Herramientas: HttpUnit, HtmlUnit, JWebUnit, Selenium, Canoo Web Test,...

Pruebas unitarias en aplicaciones web

- No hay mucha diferencia con otras pruebas unitarias, salvo por el hecho de que los servlets asumen un contexto que se deberá emular a base de *mockups* o bien desplegar en un servidor
- Estas pruebas se realizan durante la fase test, que es estándar del ciclo de vida de Maven.
  - La mayor parte de herramientas de *testing* implementan casos de prueba que heredan del `TestCase` de JUnit, así que funcionarán con el plugin “surefire” de Maven
  - Simplemente se deben colocar las pruebas en `src/test/java`

Ejemplo

- Queremos probar el siguiente servlet

```
public class SaludoServlet extends HttpServlet {
  public SaludoServlet() {
       super();
  }

  protected void doPost(HttpServletRequest request,
              HttpServletResponse response)
                throws ServletException, IOException {
       PrintWriter out = response.getWriter();
       String nom = request.getParameter("nombre");
       request.getSession().setAttribute("usuario", nom);
       out.println("<html> <head> <title>Saludo</title></head>");
       out.println("<body> <h1> Hola " + nom + "</h1>");
       out.println("</body> </html>");
  }
}
```

Prueba unitaria con ServletUnit

```
public class SaludoTest extends TestCase {
  ServletRunner sr;

  protected void setUp() throws Exception {
        super.setUp();
        sr = new ServletRunner();
        //registramos el servlet para poder llamarlo
        sr.registerServlet("SaludoServlet", SaludoServlet.class.getName());
  }

  public void testSaludo() throws IOException, SAXException, ServletException {
        //Llamamos al servlet por el nombre registrado, no por su URL del web.xml
        WebRequest pet = new GetMethodWebRequest("http://localhost:8080/SaludoServlet");
        pet.setParameter("nombre", "Juan");
        //Queremos obtener el "invocation context", necesario para
        //poder llamar a los métodos del servlet
        ServletUnitClient sc = sr.newClient();
        InvocationContext ic = sc.newInvocation(pet);
        //Del "invocation context" obtenemos el servlet
        SaludoServlet ss = (SaludoServlet) ic.getServlet();
        //Llamamos manualmente al método "doPost"
        ss.doPost(ic.getRequest(), ic.getResponse());
        //comprobamos que en la sesión se guarda el valor correcto
        assertEquals("nombre en sesión", "Juan",
                     ic.getRequest().getSession().getAttribute("usuario"));
}
```

Pruebas de integración y funcionales

- Para estos tests está la fase estándar del ciclo de vida de Maven integration-test
- **Es necesario que el servidor esté arrancado** al ejecutar estos test, pero Maven no lo hará automáticamente si no lo configuramos
  - Arrancaremos el servidor en la fase pre-integration-test
  - Ejecutaremos las pruebas con la herramienta adecuada
  - Pararemos el servidor en la fase post-integration-test
  - Si todo ha ido bien Maven continuará empaquetando, instalando en repositorio, etc,...

Arrancar/parar el servidor

```
<plugin>
    <groupId>org.codehaus.cargo</groupId>
    <artifactId>cargo-maven2-plugin</artifactId>
    <configuration>
      <container>
          [...]
      </container>
      <wait>false</wait>
    </configuration>
    <executions>
      <execution>
        <id>start</id>
        <phase>pre-integration-test</phase>
        <goals> <goal>start</goal> </goals>
      </execution>
      <execution>
        <id>stop</id>
        <phase>post-integration-test</phase>
        <goals> <goal>stop</goal> </goals>
      </execution>
    </executions>
</plugin>
```

para que Maven no se espere, tras arrancar el servidor

Separar físicamente los test

- Por desgracia Maven 2 no permite separar físicamente los test de integración de las pruebas unitarias
- Posibilidades:
  - Configurar el plugin de pruebas (surefire) para que en cada fase ignore/tenga en cuenta ciertas clases con test (ejemplo: arquetipo cargo-archetype-webapp-single-module)
  - Crear un proyecto multimódulo y configurar un módulo en exclusiva para test de integración (ejemplo: arquetipo cargo-archetype-webapp-functional-tests-module)

Configurar manualmente plugin test (surefire)

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-surefire-plugin</artifactId>
 <configuration>
  <excludes>
     <exclude>**/it/**</exclude>
  </excludes>
 </configuration>
 <executions>
   <execution>
     <phase>integration-test</phase>
       <goals><goal>test</goal></goals>
       <configuration>
          <excludes>
             <exclude>none</exclude>
          </excludes>
          <includes>
             <include>**/it/**</include>
          </includes>
       </configuration>
   </execution>
 </executions>
</plugin>
```

Las pruebas de integración estarán en un directorio llamado "it"

Ejemplo de prueba funcional

- Queremos probar el funcionamiento de este HTML y del servlet SaludoServlet, llamado por él

```
<html>
  <head> <title>Página principal</title> </head>
  <body>
    <h2>Esto es la página principal</h2>
    <form action="SaludoServlet" method="get">
      Escribe tu nombre: <input type="text" name="nombre"><br>
      <input type="submit" value="Enviar">
    </form>
  </body>
</html>
```

Pruebas con JWebUnit

- **JWebUnit** permite probar la aplicación web como una “caja negra”, comprobando lo que devuelven servlets y/o JSPs al cliente

```
public class WebAppTest extends WebTestCase {
   public void setUp() throws Exception {
      super.setUp();
      //especificar la URL base de la aplicación
      setBaseUrl("http://localhost:8080/EjemploTestsIntWebApp");
   }

   //aquí vendrán los métodos testXXXX
   ...
}
```

Pruebas con JWebUnit (II)

- **JWebUnit** permite interactuar con la aplicación como lo haría un usuario y comprobar la respuesta

```
public void testIndex() {
      //página por la que empezamos a navegar
      beginAt("/");
      //comprobar el <title> de la página
      assertTitleEquals("Página principal");
   }

public void testSaludo() {
      beginAt("/");
      //rellenar el formulario
      setTextField("nombre", "Pepe");
      //enviarlo
      submit();
      assertTitleEquals("Saludo");
      //comprobar que la página contiene un determinado texto
      assertTextPresent("Hola Pepe");
}
```

¿Preguntas...?