Desarrollo, despliegue y pruebas de aplicaciones web con Maven

Índice

1. Creación y empaquetado de un proyecto web con Maven

Vamos a ver cómo crear y empaquetar en un WAR una aplicación web usando Maven.

1.1. Creación del proyecto

Aunque se puede crear el POM partiendo de cero, es aconsejable usar uno de los arquetipos que nos ofrece Maven. Si no vamos a emplear ningún *framework* de desarrollo web, podemos usar el arquetipo para aplicaciones web más sencillo que ofrece Maven: `maven-archetype-webapp`. Este arquetipo usa la estructura de directorios estándar de aplicaciones JavaEE y empaquetará el resultado de la compilación en un WAR en lugar de un JAR.

Podemos crear el proyecto usando el plugin de Eclipse para Maven o bien en línea de comandos. Vamos a ver aquí el proceso en línea de comandos. Supongamos que nuestra aplicación se va a llamar `WebAppEjemplo`

Primero ejecutamos `mvn archetype:generate` pasándole el `artifactId` *WebAppEjemplo* y el `groupId` *es.ua.jtech*. Elegiremos el arquetipo 85, "maven-archetype-webapp".

Elegir el arquetipo web más apropiado

Los repositorios de Maven tienen un gran número de arquetipos para aplicaciones web. En realidad el arquetipo denominado "webapp-jee5" crearía un pom.xml exactamente con la versión de JavaEE que estamos usando durante el curso. Pero aunque solo sea la primera vez, es mucho más instructivo examinar qué necesitamos añadir al arquetipo básico y ver por qué, que usar una plantilla hecha sin saber cómo funciona ni cómo adaptarla a la versión de JavaEE que podamos necesitar. Una vez entendidas las ideas básicas, para aumentar la productividad es mejor ir al arquetipo más similar a nuestras necesidades finales.

```
mvn archetype:generate -DgroupId=es.ua.jtech
-DartifactId=WebAppEjemplo
```

Se nos pedirá el número del arquetipo (85) y los parámetros que no hemos especificado en línea de comandos, como el package o la versión.

Si abrimos el POM.xml generado veremos que es bastante sencillo y muy similar a los proyectos Java de escritorio. Únicamente se ha cambiado el tipo de empaquetado (`<packaging>`)a WAR. La única dependencia del proyecto por el momento es JUnit (modificaremos esto más tarde).

Por otro lado, la estructura de directorios es ligeramente más compleja que en un proyecto estándar. Dentro de la carpeta `src/main` tendremos una subcarpeta `webapp` donde se colocan los recursos web (páginas HTML o JSP, imágenes, etc) y el directorio `WEB-INF`

con el descriptor de despliegue. El arquetipo seleccionado habrá creado un descriptor de despliegue mínimo y una página index.jsp tipo "hola, mundo".

Podemos probar el empaquetado e instalación de la aplicación en el repositorio local ejecutando

```
mvn install
```

Esto generará dentro del directorio `target` el WAR con la aplicación empaquetada. El nombre del .war sigue el formato `${artifactId}-${version}.war`. Además instalará el mismo .war en el respositorio local de maven. Podemos desplegar la aplicación en Tomcat sin más que dejar dicho .war en su directorio `webapps`. Posteriormente veremos cómo hacer el despliegue de forma automatizada con maven

1.2. Configuración de la fase de compilación

El arquetipo generado por Maven no tiene ningún fichero fuente Java de ejemplo. Evidentemente un proyecto web real contendrá clases java, que se colocarían en `src/main/java`.

La configuración más habitual en la fase de compilación es la versión de Java que debe usar el compilador a la hora de leer el fuente y generar el *bytecode*. Por defecto se supone la versión 1.4, así que si nuestro proyecto usa elementos de la 1.5 como genéricos, anotaciones, etc, tendremos que cambiarlo a esta versión. Para ello debemos configurar el `maven-compiler-plugin`. Hacia el final del POM.xml añadiremos el código necesario, quedando el fichero como se muestra:

```
<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>es.ua.jtech</groupId>
  <artifactId>WebAppEjemplo</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>WebAppEjemplo Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>WebAppEjemplo</finalName>
    <plugins>
      <plugin>
```

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

Por otro lado, lo habitual es que nuestro proyecto dependa de librerías propias o de terceros además de JUnit. Como mínimo si es una aplicación web y tenemos algún servlet habrá dependencias del API de servlets que será necesario resolver para poder compilar. Los JAR con el API de servlets se suelen incluir en el propio servidor de aplicaciones, por lo que a maven habrá que decirle que es una dependencia únicamente en tiempo de compilación, pero que no debe empaquetarla en el .war resultante. Esto se especifica diciendo que la dependencia tiene ámbito (scope) *provided*.

```
<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">
  [...]
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  [...]
</project>
```

Como se verá en el módulo de servlets y JSP, si nuestra aplicación incluye alguna librería de etiquetas (*taglib*) JSP propia, tendremos que especificar la dependencia del API de JSP. Al igual que ocurre con los servlets el JAR se suele tomar del servidor, por lo que es de ámbito *provided*.

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

1.3. Trabajar con proyectos web Maven desde Eclipse

Si queremos editar el código, compilarlo y probar el proyecto desde dentro de Eclipse podemos usar el plugin *m2eclipse* que ya hemos empleado en el curso. Simplemente tenemos que importar el proyecto de Maven a Eclipse, pero antes tenemos que asegurarnos de que el plugin opcional para trabajar con proyectos web está instalado. Si lo está, al seleccionar la opción de menú `Help > About Eclipse` y pulsar en `Installation details` debería aparecer entre los plugins instalados "Maven integration for WTP (optional)".

Para instalar este extra, en Eclipse ir a `Help > Install new software...`. Tenemos que añadir un nuevo *site* de donde descargar el extra. Pulsar sobre el botón `Add...`. Como nombre del sitio podemos poner por ejemplo "m2eclipse extras". La URL debe ser `http://m2eclipse.sonatype.org/sites/m2e-extras`. Tras un tiempo nos aparecerán los plugins disponibles. Seleccionamos "Maven integration for Eclipse Extras" y dentro de él debemos instalar al menos el "Maven integration for WTP (optional)".

Falta importar el proyecto que se ha creado con Maven a Eclipse. Se seleccionaría la opción de menú `File > Import...` y dentro de la categoría `Maven` se elegiría `Existing Maven projects`. A partir de este momento podemos trabajar con el proyecto como es habitual en Eclipse, con la única diferencia de que también podemos gestionar su ciclo de vida a través de Maven.

2. Despliegue de aplicaciones web con maven

Aunque ya conocemos diversos modos de desplegar una aplicación web en Tomcat, es interesante integrar el proceso de despliegue o de arranque/parada del contenedor con Maven. Para ello existen diversos plugins, de los cuales el más conocido es el plugin de Cargo para Maven. Cargo es un *framework* java para controlar servidores de aplicaciones (parar/arrancar, controlar el estado actual, desplegar/replegar aplicaciones,...). Se pueden controlar tanto servidores locales como remotos, y se acepta un gran número de servidores JavaEE: no solo Tomcat, sino también JBoss, Glassfish, Weblogic, Jetty y otros.

Plugins y más plugins...

Existe un plugin Maven para Apache Tomcat, aunque aquí veremos Cargo porque está más extendido y además lo podremos usar también con el servidor Glassfish en aplicaciones enterprise. Otro plugin muy conocido es el plugin de Jetty, ya que Jetty es un servidor ligero y de tamaño reducido. Por ello es muy usado en demos y en proyectos web distribuidos como código de ejemplo de alguna tecnología. Así ejecutar el ejemplo es muy sencillo porque Maven se encarga de resolver todas las dependencias incluyendo instalar y ejecutar el propio servidor web.

Cargo ofrece una gran versatilidad: se puede configurar cómo desplegar el artefacto creado por Maven (su localización física, la ruta de su contexto,...), todos los aspectos del servidor (si es local o remoto, de dónde tomar sus ficheros de configuración,...). Incluso se pueden hacer cosas como instalar el servidor sobre la marcha sin más que dar una URL

de donde bajárselo en formato .zip. Es de esperar por tanto que su configuración detallada sea compleja. Nos limitaremos aquí a dar una configuración apropiada para nuestros propósitos.

Como nosotros usamos un servidor instalado en local, nos basta con decirle a Cargo qué servidor es y qué versión, en qué directorio está instalado y dónde queremos hacer el despliegue físico de la aplicación (se usará un directorio temporal). El plugin se incluiría en el POM.xml y se configuraría del siguiente modo:

```
[...]
<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>
[...]
```

En el caso de Tomcat, en el directorio donde se va a hacer el despliegue se hace una copia temporal de la estructura de directorios del servidor ("conf", "logs", "webapps", etc.). El despliegue se hace copiando el war generado por maven en esta carpeta "webapps" (no en la original de la instalación local de Tomcat). Esto nos permitiría usar una configuración distinta para el servidor y específica para nuestro proyecto (algo parecido a lo que hace el plugin WTP de Eclipse).

2.1. Despliegue de aplicaciones

Para arrancar el servidor, hay que ejecutar el objetivo `cargo:start`. El despliegue del .war generado por Maven se hará automáticamente, pero antes tendríamos que asegurarnos de que se genere con `mvn install`. Por tanto podemos hacer:

```
mvn install cargo:start
```

Por supuesto también podríamos enlazar el objetivo start con alguna de las fases del ciclo de vida estándar, para que se ejecute automáticamente. Posteriormente veremos cómo hacer esto en el contexto de pruebas de integración. El shell desde el que se arranca el servidor se queda parado por defecto con el servidor ejecutándose (salvo que especifiquemos `<wait>false</wait>` en la configuración del plugin). La parada del

servidor se puede hacer con Ctrl-C desde el shell con el que se ha arrancado o bien ejecutando el objetivo `cargo:stop`

Si lo único que queremos hacer es el despliegue, ejecutamos `cargo:deploy`. Para replegar, `cargo:undeploy`

2.2. Inclusión de librerías comunes

Es muy típico tener un mismo JAR que está compartido por varias aplicaciones. Normalmente todos los servidores tienen un directorio donde dejar estos JAR de manera que son accesibles a todas las aplicaciones desplegadas, o un mecanismo equivalente (recordemos que en Tomcat este directorio es "lib"). Así no es necesario incluir el JAR por separado en cada aplicación.

Cargo nos permite incluir el JAR en el servidor y hacerlo accesible a todas las aplicaciones sin necesidad de preocuparnos de en qué directorio físico debe residir con el servidor actual. Para ello hay que incluir en el POM la dependencia del JAR del modo estándar en Maven y luego añadir un `<dependencies>` con la lista de dependencias dentro de la configuración del contenedor de Cargo. Por ejemplo, para incluir en el servidor el JAR con el driver de MySQL haríamos algo como:

```
<!-- sección estándar de dependencias  -->
<dependencies>
[...]
   <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.13</version>
      <scope>provided</scope>
   </dependency>
</dependencies>

<build>
   <plugins>
      <plugin>
         [...]
      </plugin>
      <plugin>
         <groupId>org.codehaus.cargo</groupId>
         <artifactId>cargo-maven2-plugin</artifactId>
         <configuration>
            <container>
               <containerId>tomcat6x</containerId>
               <home>/opt/apache-tomcat-6.0.29</home>
               <!-- dependencias para el contenedor >
               <dependencies>
                  <dependency>
                     <groupId>mysql</groupId>
                     <artifactId>mysql-connector-java</artifactId>
                     <!-- nótese que aquí no ponemos la versión ya
que solo
                     estamos referenciando la dependencia
```

```
                    que está en la sección "estándar" de Maven -->
                  </dependency>
               </dependencies>
            </container>
            <configuration>
               ${project.build.directory}/tomcat6x</home>
            </configuration>
                </configuration>
      </plugin>
      [...]
     </plugins>
[...]
</build>
```

3. Pruebas en aplicaciones web

En la capa web puede ser necesario hacer diversos tipos de pruebas:

- **Pruebas unitarias:**, que comprueben el funcionamiento de los servlets propiamente dichos. Se usarían *mocks* para modelar el resto de elementos con los que va a interactuar el servlet, sean clases de la capa de negocio o bien elementos del contexto de los servlets (la petición, la respuesta, *cookies*, la sesión,...). Un ejemplo de herramienta que nos permite hacer pruebas de este tipo es ServletUnit, herramienta que forma parte de HttpUnit.
- **Pruebas de integración** de los servlets con el resto de elementos que forman parte de la aplicación. En este caso es apropiado probar los servlets dentro de un contenedor web real. Esto podemos hacerlo por ejemplo con la herramienta Cactus.
- **Pruebas funcionales:** un tercer tipo de pruebas serían aquellas que se hacen acercándose más al punto de vista del usuario final. Es decir, seguir la secuencia de un caso de uso y comprobar que el sistema devuelve los resultados esperados. Para hacer este tipo de pruebas disponemos de herramientas como HtmlUnit, Selenium o JWebUnit (esta última "envuelve" a las dos primeras para poder realizar las pruebas con un API común).

3.1. Pruebas unitarias en la capa web

Gestionar las pruebas unitarias con Maven sería muy similar a gestionar el tipo de pruebas unitarias que hemos hecho hasta ahora en el curso, con la única diferencia de que nos ayudaría disponer de una herramienta que incorpore los *mocks* de las clases del contenedor con las que debe interactuar el servlet. Como hemos dicho, ServletUnit es una de dichas herramientas. Como forma parte de HttpUnit habrá que incluir la dependencia de esta última en el POM.

```
<dependency>
   <groupId>httpunit</groupId>
   <artifactId>httpunit</artifactId>
   <version>1.7</version>
</dependency>
```

ServletUnit nos permite hacer pruebas de "caja negra", comprobando el valor de la respuesta que devuelve el servlet. También podemos ejecutar métodos individuales del servlet y comprobar los valores que devuelve el método o los que coloca el servlet en los distintos ámbitos (petición, sesión, ...).

Por ejemplo, supongamos el código del siguiente servlet:

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

  protected void doGet(HttpServletRequest request,
                HttpServletResponse response) throws
ServletException, IOException {
        doPost(request, response);
  }

  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>");
        }
  }
```

Podemos comprobar por ejemplo que al llamar al doPost se guarda en la sesión el valor esperado:

```
public class SaludoTest extends TestCase {
  //esto es lo que nos permite lanzar el servlet en el entorno
simulado
  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"));
}
```

También podemos probar el resultado devuelto por el servlet sin entrar a ejecutar los métodos manualmente. Esto es más sencillo (aunque para eso hay otras herramientas más sofisticadas)

```
public void testSaludoCajaNegra()  {
        ServletUnitClient sc = sr.newClient();
        WebRequest pet = new
GetMethodWebRequest("http://localhost:8080/SaludoServlet");
        pet.setParameter("nombre", "Juan");
        WebResponse resp = sc.getResponse(pet);
        assertEquals("Titulo de la página","Saludo",
resp.getTitle());
}
```

3.2. Pruebas de integración y funcionales

No obstante, el asunto de las pruebas funcionales y de integración es distinto, ya que se deben ejecutar con el contenedor web arrancado, y por tanto en una fase distinta a las pruebas unitarias. En el ciclo de vida estándar de Maven hay tres fases correspondientes a pruebas de integración: `pre-integration-test`, `integration-test` y `post-integration-test`. Estas fases se ejecutan después de `package`, cuando se empaqueta el artefacto generado en un JAR, WAR o EAR. Vamos a ver cómo usar estas fases para hacer las pruebas.

3.2.1. Adaptar la estructura del proyecto

Por desgracia, Maven no incluye por defecto la posibilidad de ejecutar un conjunto de prueba en la fase `test` (pruebas unitarias) y usar otro distinto en la fase `integration-test`. Tendremos que conseguirlo configurando manualmente el POM. Caben dos posibilidades:

- Configurar el plugin `surefire` de modo que ejecute algunos ficheros de prueba en la fase de pruebas unitarias y otros en la fase de integración. La distinción entre unos y otros se puede hacer por el directorio en que están guardados o por el nombre de las clases de prueba. Cargo incluye un arquetipo que nos creará un proyecto de esta clase: cargo-archetype-webapp-single-module (179). En el POM podremos ver que los test

de integración se meten en un directorio llamado `it` dentro de `src/test/java` y que se configura el plugin surefire para que ignore este directorio durante la fase de pruebas unitarias y ejecute los tests durante la fase de pruebas de integración.

```
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-surefire-plugin</artifactId>
 <configuration>
  <!--
        Excluir los test de integración de la fase de test
        (fase de pruebas unitarias)
  -->
  <excludes>
     <exclude>**/it/**</exclude>
  </excludes>
 </configuration>
 <executions>
   <execution>
      <phase>integration-test</phase>
         <goals>
            <goal>test</goal>
         </goals>
         <configuration>
            <!--
                Incluir los test de integración en la fase
                integration-test
            -->
            <excludes>
               <exclude>none</exclude>
            </excludes>
            <includes>
               <include>**/it/**</include>
            </includes>
         </configuration>
   </execution>
 </executions>
</plugin>
```

- Construir un proyecto multimódulo. En uno de los módulos irá la aplicación web junto con las pruebas unitarias, y en otro irán las pruebas de integración. Así tenemos por separado este código, lo que hace la estructura más limpia y más flexible. Podemos, por ejemplo, ejecutar las pruebas de integración o funcionales solo en determinadas ocasiones y no en cada compilación del proyecto. Cargo nos ofrece un arquetipo que seguirá esta estructura:
  cargo-archetype-webapp-functional-tests-module (178)

3.2.2. Gestionar el servidor web en la fase de pruebas

En cualquier caso, hay que arrancar el servidor web antes de ejecutar las pruebas de integración (fase "pre-integration-test") y pararlo después de ejecutar dichas pruebas (fase "post-integration-test"). Simplemente tendremos que enlazar el objetivo cargo:start con la primera fase y cargo:stop con la otra. Podemos conseguirlo introduciendo en el POM un código como este:

```
[...]
<plugin>
   <groupId>org.codehaus.cargo</groupId>
   <artifactId>cargo-maven2-plugin</artifactId>
   <configuration>
     <container>
        <!-- Aquí faltaría la configuración del contenedor
         (qué servidor es, dónde está,...) -->
        [...]
     </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>
```

Nótese que en la configuración del contenedor ponemos `<wait>false</wait>` para que Maven no se quede "esperando" tras arrancar el contenedor.

3.2.3. Ejecutar las pruebas

Como ya hemos dicho, existen diversas herramientas para ejecutar pruebas de integración y funcionales. Vamos a ver aquí un ejemplo de JWebUnit, que nos ofrece un API sencillo para probar la aplicación web del mismo modo que lo haría un usuario final. Así podemos acceder a una página, rellenar los campos de un formulario y enviarlo o pinchar en un enlace. Podemos también comprobar si en la página aparece algún elemento que tendría que estar (un determinado enlace, un texto, una tabla) o si tiene las características deseadas (el título, el tamaño,...).

Para usar JWebUnit debemos incluir la siguiente dependencia en el POM

```
<dependency>
        <groupId>net.sourceforge.jwebunit</groupId>
        <artifactId>jwebunit-htmlunit-plugin</artifactId>
        <version>2.2</version>
        <scope>test</scope>
</dependency>
```

JWebUnit es una capa de abstracción sobre HtmlUnit (una herramienta bastante veterana para pruebas web), así que veremos que Maven descarga también a nuestro repositorio esta última.

Suponiendo que tuviéramos el servlet `SaludoServlet` del que hablábamos en el apartado de pruebas unitarias y además el siguiente JSP que lo llama a través de un formulario:

```
<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>
```

Podríamos comprobar que tanto el JSP como el servlet devuelven el resultado esperado:

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

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

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

Se recomienda consultar la referencia básica de JWebUnit para ver con más detalle las posibilidades que ofrece la herramienta.