Componentes Web

Sesión 1 - Introducción a las aplicaciones web

Índice

Aplicaciones web

- Una aplicación web es una aplicación a la que se accede mediante HTTP
  - Utilizando un navegador web
- A la hora de desarrollar una aplicación web suelen utilizarse diferentes tecnologías

- En el lado del SERVIDOR:
- Debe ser capaz de recoger la petición del cliente y enviarle la respuesta adecuada
- Puede valerse de herramientas externas para procesar la petición y generar la respuesta de forma dinámica
  - *Servlets, JSP, PHP, ASP, etc.*

- En el lado del CLIENTE:
- Al cliente se le ofrece una respuesta visible en forma de página web
- Podemos utilizar elementos estáticos (HTML) o bien valernos de herramientas que den cierto dinamismo también a lo que se envía al cliente
  - *Javascript, Applets, Flash, etc.*

Aplicaciones web Java EE

- Las aplicaciones web Java EE se componen de:
  - Recursos estáticos
    HTML, imágenes, etc.
  - Documentos dinámicos
    Páginas JSP
  - Clases Java
    Servlets, *beans* y otros objetos Java
    Deben ser compiladas
  - Configuración de la aplicación
    Descriptor de despliegue (fichero XML)

Estructura de una aplicación web Java EE

- Estructura de directorios

`/`	Recursos estáticos y JSP Parte pública accesible desde la web
`/WEB-INF`	Configuración y clases Java No accesible desde la web
`/WEB-INF/web.xml`	Fichero descriptor de despliegue Configuración de la aplicación
`/WEB-INF/classes`	Clases Java de nuestra aplicación Ficheros `.class` (en estructura de paquetes)
`/WEB-INF/lib`	Librerías que utiliza la aplicación Ficheros JAR

Contexto

- Cada Aplicación Web es un contexto
  - Se compone de la estructura de directorios anterior

- A cada contexto se le asigna una ruta dentro del servidor
  - Por ejemplo, si asignamos la ruta `aplic` al contexto correspondiente a la siguiente estructura:

```
/pagina.htm
/WEB-INF/web.xml
```

- Podremos acceder a nuestra página con

  http://localhost:8080/aplic/pagina.htm

Ficheros WAR

- Podemos empaquetar las Aplicaciones Web en ficheros WAR (Archivos de Aplicación Web)

- Se utiliza la misma herramienta JAR para crearlos (sólo utilizamos una extensión distinta)
  - Contendrá la estructura de directorios completa del contexto

- Es un estándar de los servidores de aplicaciones Java EE

- Se utiliza para distribuir aplicaciones web
  - Podremos copiar el fichero WAR directamente al servidor web para poner en marcha la aplicación

Creación de un WAR

- Dada la siguiente estructura de carpetas:

```
web/ejemplo/
    index.html
    WEB-INF/
        web.xml
        classes/
            ClaseServlet.class
```

- Entrar en el directorio `web/ejemplo` y teclear

```
jar cMvf ejemplo.war *
```

- El raíz del WAR deberá contener `index.html` y `WEB-INF`

Despliegue en WildFly

- Ponemos en marcha el servidor *standalone*

```
$WILDFLY_HOME/bin/standalone.sh
```

- Copiamos el fichero WAR a `$WILDFLY_HOME/standalone/desployments`

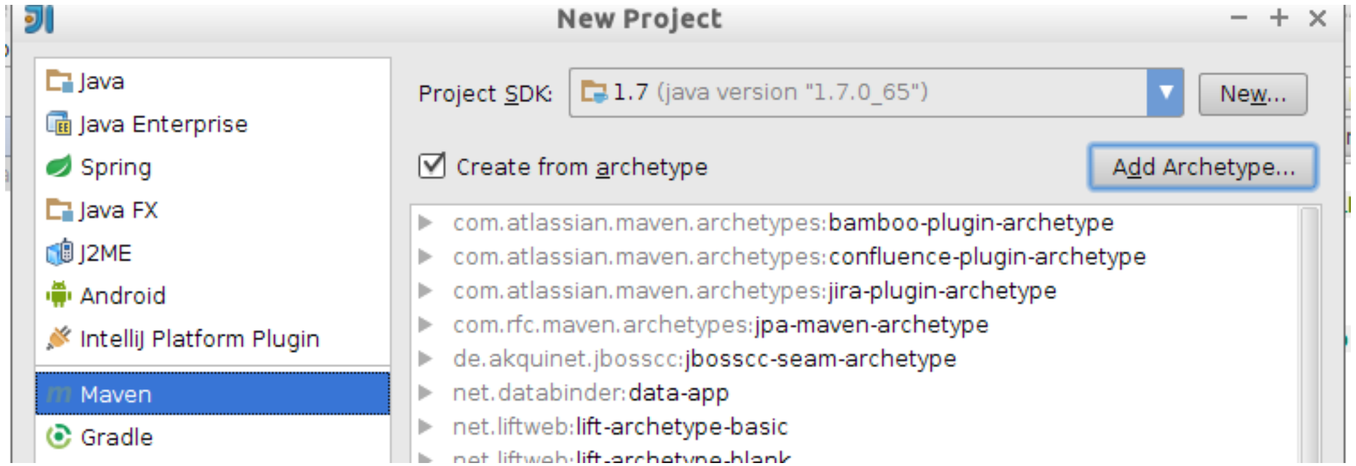
- WildFly descomprimirá y desplegará automáticamente el fichero WAR

- Podremos acceder a la aplicación mediante el navegador

http://localhost:8080/miaplicacion/

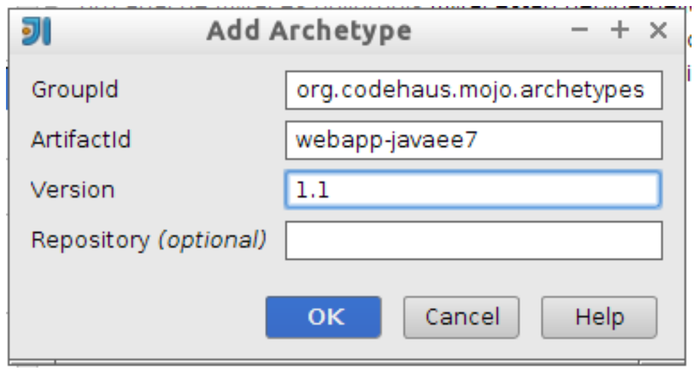
Creación de un proyecto web con IntelliJ

- Podemos crearlo como proyecto web Maven con ***New Project > Maven***



- Seleccionamos el arquetipo **`webapp-javaee7`**

```
<groupId>org.codehaus.mojo.archetypes</groupId>
<artifactId>webapp-javaee7</artifactId>
<version>1.1</version>
```



Nombre de la aplicación

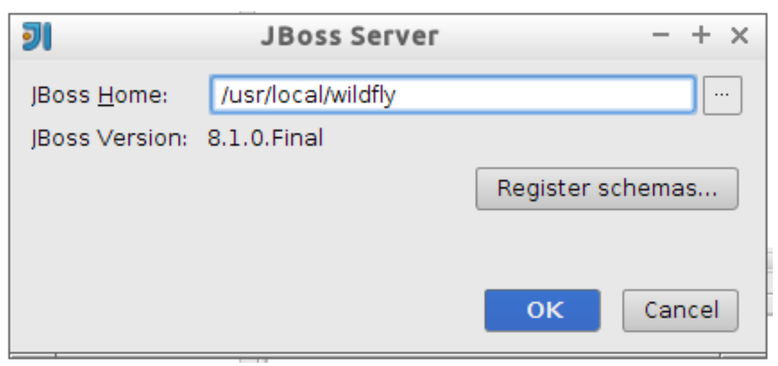
- Podemos especificar el nombre con el que se desplegará la aplicación en el `pom.xml`
  - Utilizamos la etiqueta `<finalName>`
  - Este será el nombre que se le dé al fichero WAR generado

```
<project>
    ...
    <build>
        <finalName>MiAplicacionWeb</finalName>
        ...
    </build>
</project>
```

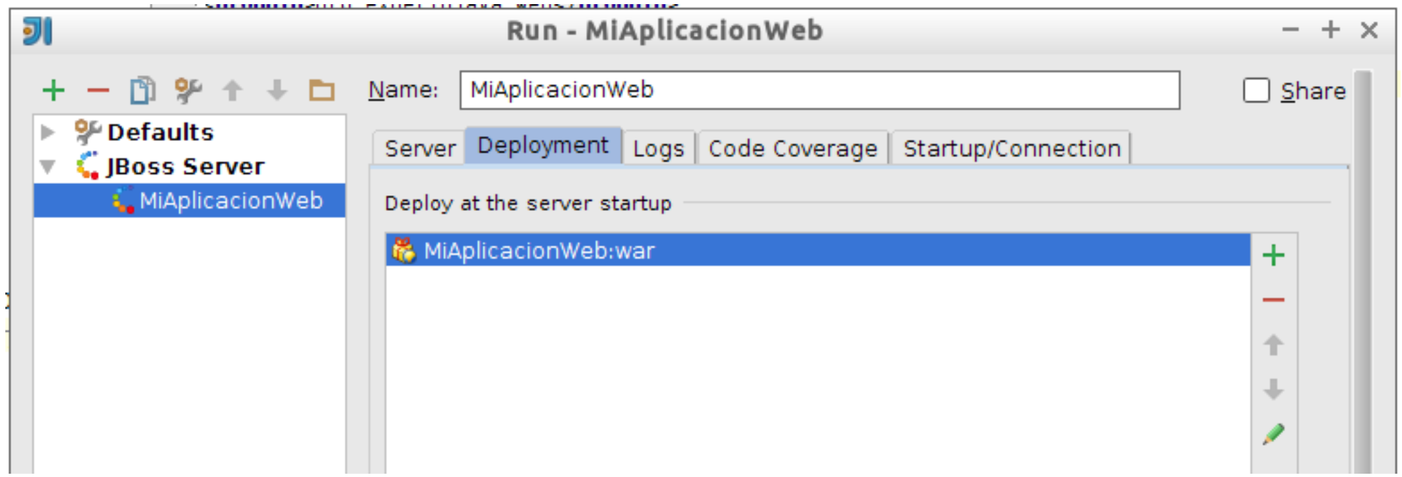
Despliegue desde IntelliJ

- Debemos crear un perfil de ejecución con *Run > Edit configurations …*

- Configuramos un servidor **JBoss Local**

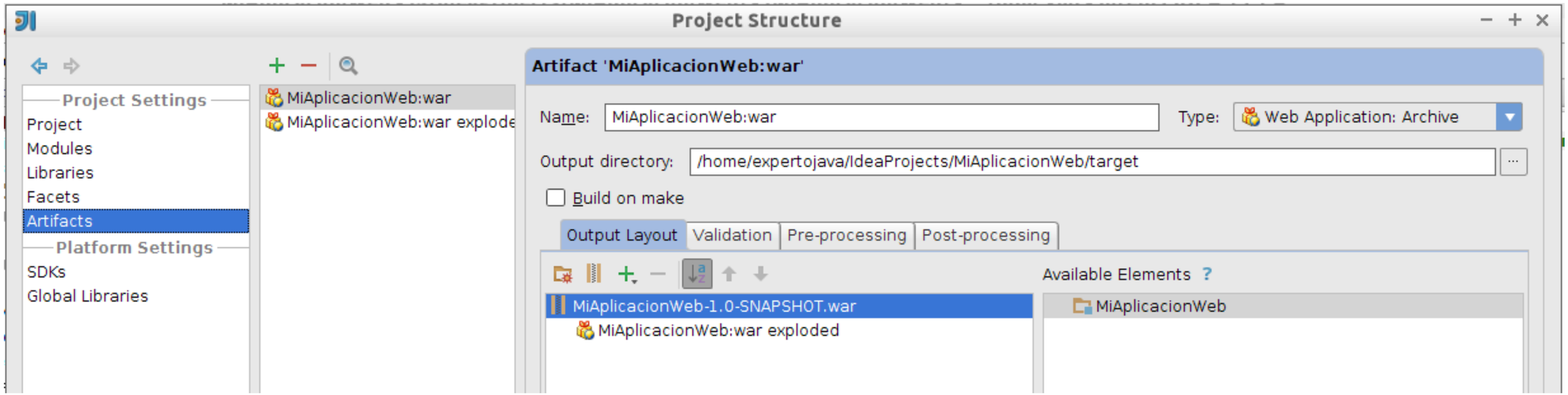


- Añadimos artefacto WAR en despliegue (*Deployment*)



Configuración de artefactos

- En *File* > *Project Structure …* > *Artifacts* podemos configurar los artefactos del proyecto
  - Por defecto tenemos *war* y *war exploded*



Descriptor de despliegue

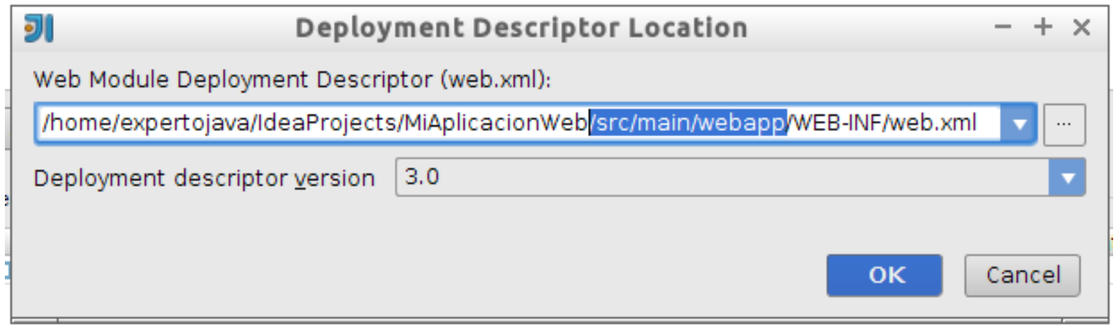
- Fichero `WEB-INF/web.xml`
  - Contiene configuración relativa a la aplicación
  - Opcional a partir de la API de *servlets* 3.0

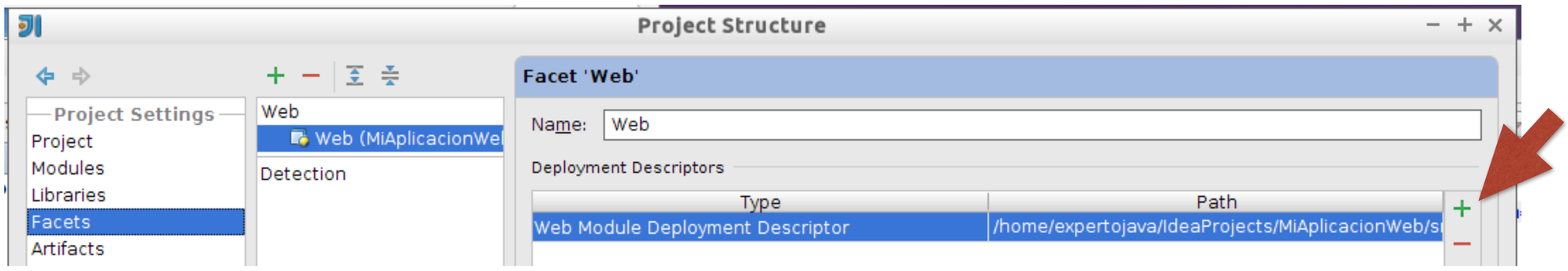
```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">

    <display-name>
       Mi Aplicacion Web
    </display-name>
    <description>
       Esta es una aplicacion web sencilla a modo de ejemplo
    </description>
</web-app>
```

Creación del descriptor de despliegue con IntelliJ

- Entramos en *File* > *Project Structure* … _ > *Facets*
- Seleccionamos el *facet Web*
- Añadimos nuevo descriptor de despliegue en `src/main/webapp/WEB-INF`





Páginas de inicio

- Podemos configurar en el descriptor de despliegue las páginas de inicio por defecto

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

- Al entrar en la siguiente dirección buscará una de las páginas indicadas

http://localhost:8080/miaplicacion/

Definición de servlet

- Un *servlet* es un programa Java que se ejecuta en un servidor web y construye o sirve páginas web.

- Permite la construcción dinámica de páginas, en función de determinados parámetros de entrada

- Más *sencillo* que un CGI, más *eficiente* (se arranca un hilo por petición, y no un proceso entero), más *potente* y más *portable*.

Recursos de servlets y JSP

- Servlets y JSP son dos conceptos muy interrelacionados
- Para trabajar con ellos se necesita:
  - Un *servidor Web* con soporte para servlets / JSP (*contenedor* de servlets y JSP: Tomcat, JBoss, …)
  - Las *librerías* o clases necesarias (proporcionadas por el servidor)
  - Recomendable también la *documentación de la API* de servlets / JSP

Arquitectura del paquete servlet

- En el paquete `javax.servlet` tenemos toda la infraestructura para trabajar con servlets
- El elemento central es la interfaz `Servlet`
- La clase `GenericServlet` es una clase abstracta que la implementa para un servlet genérico independiente del protocolo
- La clase `HttpServlet` en el paquete `javax.servlet.http` hereda de la anterior para definir un servlet vía web utilizando HTTP

Arquitectura del paquete servlet

- Cuando un servlet recibe una petición de un cliente, se reciben dos objetos:

  - Un objeto `ServletRequest` con los datos de la petición (información entrante: parámetros, protocolo, etc)

    Se puede obtener un `ServletInputStream` para leer los datos como un stream de entrada

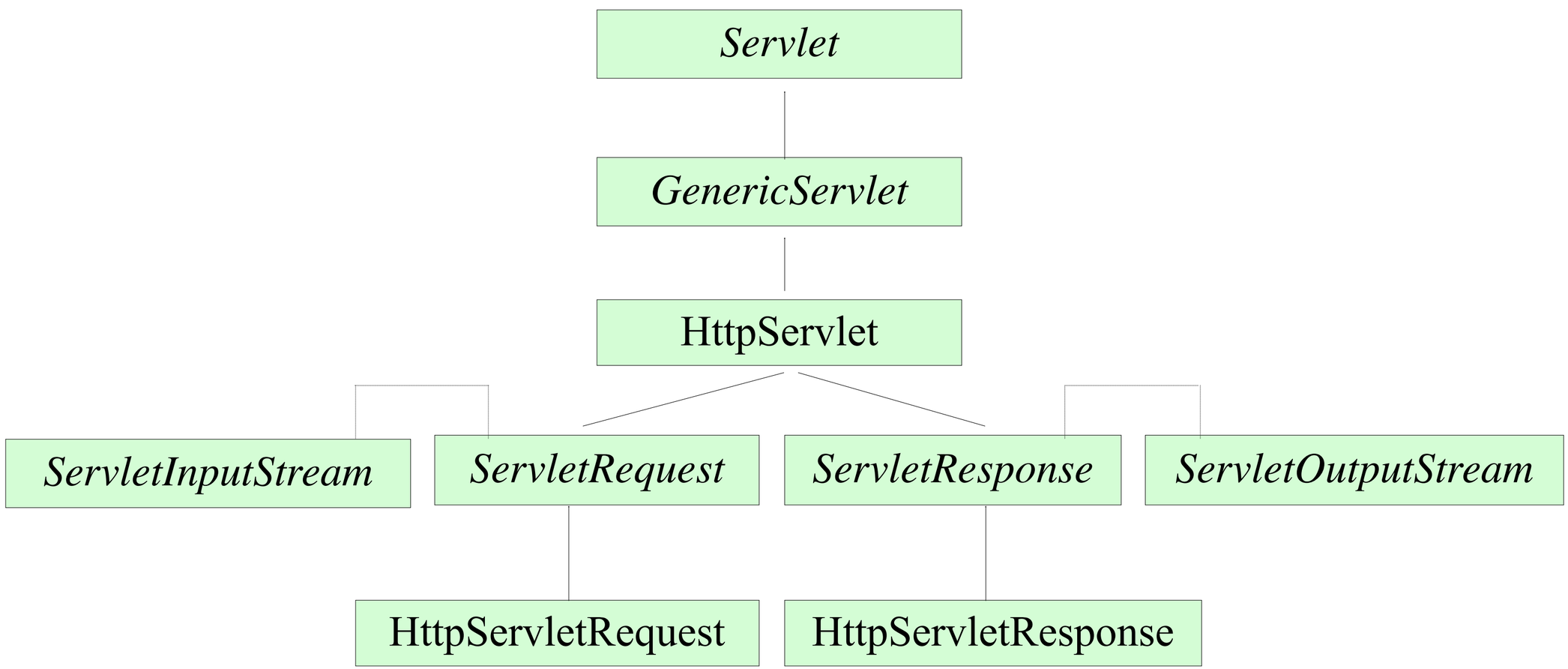
    La subclase `HttpServletRequest` procesa peticiones HTTP

  - Un objeto `ServletResponse` donde se colocarán los datos de respuesta del servlet ante la petición

    Se puede obtener un `ServletOutputStream` o un `Writer` para escribir esos datos en la salida

    La subclase `HttpServletResponse` trata respuestas HTTP

Arquitectura del paquete servlet



Ciclo de vida de un servlet

- Todos los servlets tienen el mismo ciclo de vida:
  - El servidor carga e inicializa el servlet
  - El servlet procesa N peticiones
  - El servidor destruye el servlet
- *Inicialización*: para tareas que se hagan una sola vez al iniciar el servlet

```
public void init() throws ServletException
{
    ...
}

public void init(ServletConfig conf) throws ServletException
{
    super.init(conf);
    ...
}
```

Ciclo de vida de un servlet

- *Procesamiento de peticiones*: cada petición llama al método `service( )`

```
public void service(HttpServletRequest request,
                    HttpServletResponse response)
throws ServletException, IOException
```

- Según el tipo de petición, llama a uno de los métodos (todos con los mismos parámetros y excepciones que `service( )`):

```
public void doGet(...)
public void doPost(...)
public void doPut(...)
public void doDelete(...)
public void doOptions(...)
public void doTrace(...)
```

Ciclo de vida de un servlet

- *Destrucción*: método `destroy( )`

```
public void destroy() throws ServletException
```

- Se debe deshacer todo lo construido en `init( )`
- Se llama a este método cuando todas las peticiones han concluido, o cuando ha pasado un determinado tiempo (en este caso, se debe controlar por código que se destruya cuando debe)

Estructura básica de un servlet

```
import javax.servlet.*;
import javax.servlet.http.*;

public class ClaseServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                throws ServletException, IOException {
        // ... codigo para una peticion GET
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                 throws ServletException, IOException {
        // ... codigo para una peticion POST
    }
}
```

NOTA: esta es la estructura más común de un servlet. Además, se puede incluir cualquiera de los métodos (`init()`, `destroy()`, `doPut()`, etc) vistos antes

Publicación de servlets

- Para utilizar un servlet en una aplicación web, la clase con el servlet debe estar localizable
  - Fichero `.class` en el directorio `WEB-INF/classes`, con su estructura de paquetes y subpaquetes
  - Empaquetado en un JAR dentro de `WEB-INF/lib`

- Debemos asociar los servlets a una URL
- A partir de servlets 3.0 se hace mediante anotaciones

```
@WebServlet("/UrlServlet")
public class ClaseServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) {
        ...
    }
}
```

Acceso a los servlets

- Podemos también dar un nombre al servlet

```
@WebServlet(name="miServlet", urlPatterns="/UrlServlet")
public class ClaseServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) {
        ...
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response) {
        ...
    }
}
```

- Una vez mapeado, podremos acceder al servlet en la dirección indicada dentro de nuestro contexto (`<contexto>`)

http://localhost:8080/<contexto>/UrlServlet

Declaración en web.xml

- Podemos también mapear los servlets mediante el descriptor de despliegue (web.xml)
  - En versiones previas a la 3.0 esta es la única forma
- Primero debemos declarar los servlets:

```
<servlet>
    <servlet-name>nombre</servlet-name>
    <servlet-class>unpaquete.ClaseServlet</servlet-class>
</servlet>
```

- Después debemos mapear el servlet a una URL

```
<servlet-mapping>
    <servlet-name>nombre</servlet-name>
    <url-pattern>/ejemploservlet</url-pattern>
</servlet-mapping>
```

Mapeo a múltiples URLs

- Podemos mapear a diferentes patrones:

```
<servlet-mapping>
    <servlet-name>nombre</servlet-name>
    <url-pattern>/ejemploservlet/*</url-pattern>
</servlet-mapping>
```

→ http://localhost:8080/<dir>/ejemploservlet/unapagina.html

```
<servlet-mapping>
    <servlet-name>nombre</servlet-name>
    <url-pattern>/ejemploservlet/*.do</url-pattern>
</servlet-mapping>
```

→ http://localhost:8080/<dir>/ejemploservlet/login.do

- Podemos mapear un servlet (o incluso JSPs) a varias URLs

Parámetros de inicio en servlets

- En los servlets podemos declarar parámetros
  - En la anotación del servlet

```
@WebServlet(urlPatterns="/UrlServlet",
            initParams ={
                @InitParam(name="param1", value="valor1"),
                @InitParam(name="param2", value="valor2") })
```

  - Mediante anotaciones independientes

```
@WebInitParam(name="param1", value="valor1")
```

- Podemos en su código acceder a estos parámetros con:

```
String s = getServletConfig().getInitParameter("param1"));
```

Parámetros en web.xml

- Al declarar un servlet o página JSP en web.xml también podemos definir uno o más parámetros de inicio

```
<servlet>
    <servlet-name>nombre</servlet-name>
    <servlet-class>ClaseServlet</servlet-class>
    <init-param>
        <param-name>param1</param-name>
        <param-value>valor1</param-value>
    </init-param>
    <init-param>
        <param-name>param2</param-name>
        <param-value>valor2</param-value>
    </init-param>
</servlet>
```

- Podemos modificar el valor de estos parámetros sin recompilar el servlet

Cargar servlets al inicio

- Podemos indicar que un servlet se cargue nada más iniciar el servidor Web:

```
@WebServlet(name="miServlet", urlPatterns="/UrlServlet", loadOnStartup="2")
```

- El parámetro numérico es opcional (la etiqueta puede abrirse y cerrarse sin más), e indica el orden en que cargar los servlets al inicio, si hay varios que cargar

- También en `web.xml` (previo a 3.0)

```
<servlet>
    <servlet-name>nombre</servlet-name>
    <servlet-class>ClaseServlet</servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>
```

Logging en aplicaciones web

- Utilizaremos *Log4J* encapsulado en *commons-logging* para enviar mensajes de *log* en aplicaciones web, como en una aplicación normal
  - Añadimos los JAR de *commons-logging* y *log4j* a `WEB-INF/lib`
  - Definimos los ficheros `commons-logging.properties` y `log4j.properties` dentro de la carpeta `WEB-INF/classes` de la aplicación

    Volcado automático desde carpeta "`resources`*"*
  - Colocar los mensajes en los servlets

Generación de mensajes en servlets

```
import org.apache.commons.logging.*;

public class ServletLog4J1 extends HttpServlet {

    static Log logger = LogFactory.getLog(ServletLog4J1.class);

    // Metodo para procesar una peticion GET

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                          throws ServletException, IOException {

        logger.info("Atendiendo peticion Servlet Log4J");
        PrintWriter out = response.getWriter();
        out.println ("Servlet sencillo de prueba para logging");
        logger.debug("Fin de procesamiento de petición");

    }

}
```

Introducción a los JSP

- Código Java en páginas HTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Mi primera página JSP</title>
</head>
<body>
 <p> Hoy es:
    <%= new java.util.Date() %>
  </p>
</body>
</html>
```

Comparación con los servlets

- En apariencia
  - Un JSP es HTML + Java Insertado
  - Un Servlet es Java + HTML insertado

- En realidad
  - Los JSP se traducen internamente a servlets

    El servidor web tiene una “plantilla de servlet”

    Inserta nuestro código JSP dentro

    Lo guarda en un directorio especial

    Lo compila y ejecuta

    En sucesivas llamadas a la página, solo hace falta ejecutar el servlet, salvo que se modifique el código del JSP ⇨ comenzar de nuevo

Scriptlets

- Sentencias Java `<% tam = 1; %>`

```
<%
  java.util.Calendar ahora = java.util.Calendar.getInstance();
  int hora = ahora.get(java.util.Calendar.HOUR_OF_DAY);
%>
<strong> Hola mundo,
<em>
<% if ((hora>20)||(hora<6)) { %>
     buenas noches
<% } else if ((hora>=6)&&(hora<=12)) { %>
     buenos días
<% } else { %>
     buenas tardes
<% } %>
</em>
</strong>
```

Expresiones

- Su valor se evalúa, se convierte a cadena y se imprime en el `Writer` del servlet, con un `write` o similar `<%= new Date() %>`

```
<strong>
  Esta pagina ha sido visitada <%= visitas %> veces
  Hoy es <%= new java.util.Date() %>
</strong>
```

¿Preguntas?