

Servlets

Conceptos avanzados

- 1 Repaso: características generales. Ejemplos de Servlet.
- 2 Configuración mediante archivo descriptor **WEB.XML** y mediante anotaciones
- 3 Deploy de Servlets
- 4 Respuestas de los Servlets
- 5 Alcances: **Request**, **Session**, **Application**. Atributos
- 6 Redireccionamiento del requerimiento: **sendRedirect()**
- 7 Delegación de requerimiento y respuesta: **forward()** - **include()**

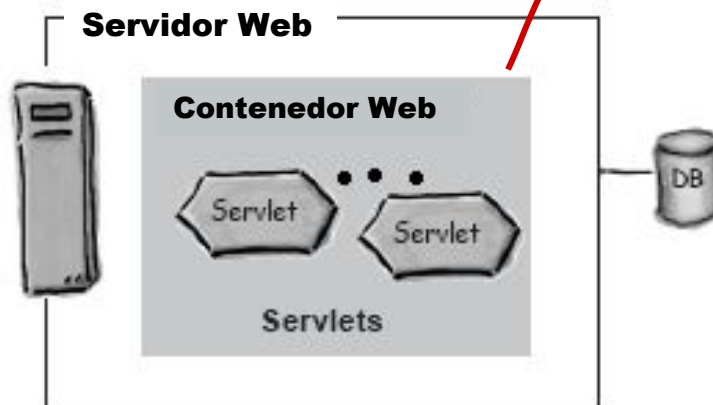
Servlets

Características generales

Un servlet es una componente web escrita en Java que es gerenciada por un Contenedor Web. Procesa requerimientos y construye respuestas dinámicamente.

El Contenedor Web es responsable de:

1. la conectividad con la red
2. capturar los requerimientos HTTP, traducirlos a objetos que el servlet entienda, entregarle dichos objetos al servlet quién los usa para producir las respuestas
3. generar una respuesta HTTP en un formato correcto (MIME-type)
4. gerenciar el ciclo de vida del servlet
5. manejar errores y proveer seguridad



El Contenedor Web interactúa con los servlets invocando métodos de gerenciamiento o métodos callback. Estos métodos definen la interface entre el Contenedor y los servlets (API de servlets).

Un ejemplo simple

Un servlet que recupera parámetros del requerimiento

Este Servlet genera una página HTML usando un parámetro del requerimiento

```
<html>
<body>
<form action="ServletHola" method="post">
  Ingresá tu nombre: <input type="text" name="nombre">
  <input type="submit" value="Enviar">
</form>
</body>
</html>
```

saludo.html

HttpServlet, extiende
GenericServlet, la cual
implementa la interface Servlet

```
package servlets;
public class ServletHola extends javax.servlet.http.HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
        ServletException, IOException {
        response.setContentType("text/html");
```

```
        PrintWriter out = response.getWriter();
```

A partir del objeto response se puede obtener
un objeto PrintWriter que nos permite escribir
texto HTML en la respuesta

```
        out.println("<html><body>");
        out.println("<h1> Hola " + request.getParameter("nombre")+" </h1>");
        out.print(" </body></html>");
        out.close();
```

A partir del objeto request se puede obtener
los parámetros del requerimiento

```
    }
}
```

Usando el archivo descriptor de la Aplicación Web

- Para que un cliente pueda acceder a un servlet, debe declararse una URL o un conjunto de URL's asociadas al servlet en el archivo descriptor de la aplicación web o web.xml (veremos que también se lo puede hacer mediante anotaciones).
- Además, el archivo .class del servlet se debe ubicar en la carpeta estándar /WEB-INF/classes de la aplicación web, junto con otras clases Java. Cualquier contenido de la carpeta /WEB-INF **no está accesible directamente por un cliente http**.

```
<?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" ... id="WebApp_ID" version="2.5">
  <servlet>
    <servlet-name>MiServlet</servlet-name>
    <servlet-class>misservlets.ServletHola</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MiServlet</servlet-name>
    <url-pattern>/ServletHola</url-pattern>
  </servlet-mapping>
</web-app>
```

web.xml

Se declara un servlet, asignándole un nombre único y una clase Java que lo implementa

Se mapea el servlet con una URL
Este es un mapeo 1 a 1
El url-pattern siempre empieza con /

URL completa del servlet: <http://www.servidor.gov.ar:8080/appPruebas/ServletHola>

URL de la aplicación web

url-pattern o mapping

Parámetros de Inicialización

ServletHola

Este Servlet retorna una página HTML con un mensaje concatenando un parámetro de inicialización con un parámetro del requerimiento.

```
public class ServletHola extends HttpServlet {
    private String saludo;
    public void init(){
        saludo = this.getServletConfig().getInitParameter("saludo");
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ..{
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.println("<h1>"+saludo+request.getParameter("nombre")+" </h1>");
        out.println("</body></html>");
        out.close();
    }
}
```

También podría **no sobrescribirse el método init()** y recuperar los valores de inicialización cuando se quiera usar, de alguna de estas dos maneras:

```
this.getInitParameter("saludo")
this.getServletConfig().getInitParameter("saludo")
```

```
. . .
<servlet>
  <servlet-name>MiServlet</servlet-name>
  <servlet-class>servlets.ServletHola</servlet-class>
  <init-param>
    <param-name>saludo</param-name>
    <param-value>Hola</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>MiServlet</servlet-name>
  <url-pattern>/ServletHola</url-pattern>
</servlet-mapping>
. . .
```

web.xml

Servlets con Anotaciones

Una alternativa al web.xml

A partir de la versión de Servlet 3.0 se pueden utilizar anotaciones para la configuración de los Servlets. Las anotaciones en la API de servlets se utilizan para reemplazar a las declaraciones/los mapeos del archivo `web.xml`.

¿Qué son las anotaciones?

- Las anotaciones son metadatos que nos permiten agregar información a nuestro código fuente para ser usado posteriormente –en tiempo de compilación o en tiempo de ejecución-.
- Las anotaciones fueron incorporadas al lenguaje java en la versión 5. La motivación de las anotaciones es la tendencia a combinar metadatos con código fuente, en lugar de mantenerlos en archivos descriptores separados.

La API de Servlets 3.0 o superior

Anotaciones

Las anotaciones **se declaran** de manera parecida a las interfaces, solo que el signo **@** precede a la palabra clave interface. Se compilan a archivos **.class** de la misma manera que las clases e interfaces.

```
package java.lang;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

Las anotaciones **se utilizan** en el código fuente precediendo a la declaración de la clase, atributos y métodos. En este caso, la anotación **@Override** es para métodos y se usa así:

```
public class Paciente {

    @Override
    public String toString() {
        return super.toString();
    }
}
```

Precede a un método

La anotación **@Override** indica que se está sobrescribiendo un método de la superclase. Si un método está precedido por esta anotación pero NO sobrescribe el método de la superclase, los compiladores deben generar un mensaje de error y la clase no compila.

@Target: indica dónde se aplican las anotaciones (métodos, clases, variables de instancia, variables locales, paquetes, constructores, etc.)

@Retention: indica dónde están disponibles las anotaciones y cuánto se mantiene la información de las anotaciones. Esto permite determinar si pueden ser leídas sólo por el compilador o por el intérprete en tiempo de ejecución. Los valores posibles son: **RetentionPolicy.SOURCE**, **RetentionPolicy.CLASS** y **RetentionPolicy.RUNTIME**.

La API de Servlets 3.0 o superior

Anotaciones

Este es un ejemplo de una declaración de la anotación `@Column` para el mapeo de objetos con tablas de una base de datos, donde tiene entre otros el método `name()` para identificar en nombre de la columna en la tabla de la base de datos.

Definición de la anotación `@column`

```
import javax.persistence.*;
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@interface Column extends Annotation{

    public String name() default "";
    . . .
}
```

La declaración de una anotación al igual que las interfaces tiene métodos abstractos pero además puede tener valores por defecto.

Uso de la anotación `@column`

```
package taller;

import javax.persistence.*;
@Entity
@Table(name="MENSAJES")
public class Mensaje {

    @Column(name="MENSAJE_ID")
    private Long id;
    . . .
}
```

Preceden a la declaración de la clase

La anotación tiene una lista entre paréntesis de pares **elemento-valor**. Los valores de los elementos deben ser constantes definidas en compilación

La API de Servlets 3.0

Un Servlet con Anotaciones

La anotación **@WebServlet** es usada para declarar la configuración de un Servlet. Si no se especifica el atributo **name** se usa el nombre de la clase.

```
package misServlet;
```

```
@WebServlet(  
    urlPatterns = { "/ServletHola" },  
    initParams = {  
        @WebInitParam(name = "saludo", value = "Hola")  
    })
```

El atributo **urlPatterns** define un conjunto de url-patterns que pueden ser usadas para invocar al Servlet.

```
public class ServletFecha extends HttpServlet{  
    private String saludo;
```

La anotación **@WebInitParam** se usa para definir los parámetros de inicialización del servlet

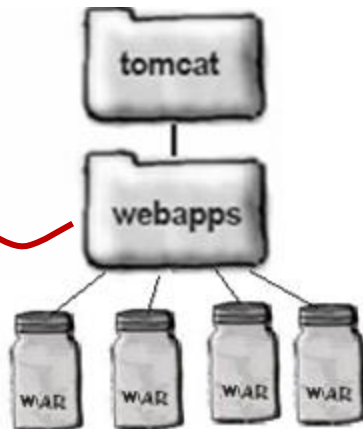
```
    public void init(){  
        saludo = this.getServletConfig().getInitParameter("saludo");  
    }
```

```
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ..{  
        PrintWriter out = response.getWriter();  
        out.println("<html><body>");  
        out.println("<h1>"+saludo+request.getParameter("nombre")+" </h1>");  
        out.println("</body></html>");  
        out.close();  
    }  
}
```

¿Cómo se hace el “deploy” de una aplicación?

- Las aplicaciones web JAVA pueden empaquetarse en un archivo *Web ARchive* (WAR). El archivo WAR es ideal para distribuir e instalar una aplicación. El formato “desempaquetado” es útil en la **etapa de desarrollo**.
- Un WAR tiene una estructura de directorios específica, donde la raíz, es el “*context root*” de la aplicación web.
- El archivo WAR es un archivo JAR que contiene un módulo web: páginas HTML, archivos de imágenes, JSPs, clases, páginas de estilo, código JavaScript, el directorio **WEB-INF** y sus subdirectorios (classes, lib, tag, el archivo web.xml, etc.)
- Los archivos WAR están definidos oficialmente en la especificación de Servlets a partir de la versión 2.2. **Son estándares**. Todos los contenedores que implementan la especificación de la API de Servlets 2.2 y superiores deben soportar archivos WAR.
- Los IDEs proveen opciones que permiten construir el WAR en forma automática. Se puede crear el archivo WAR usando la herramienta *jar* del JDK.

En el servidor Tomcat, el archivo WAR de la aplicación web se debe copiar en el directorio *webapps*

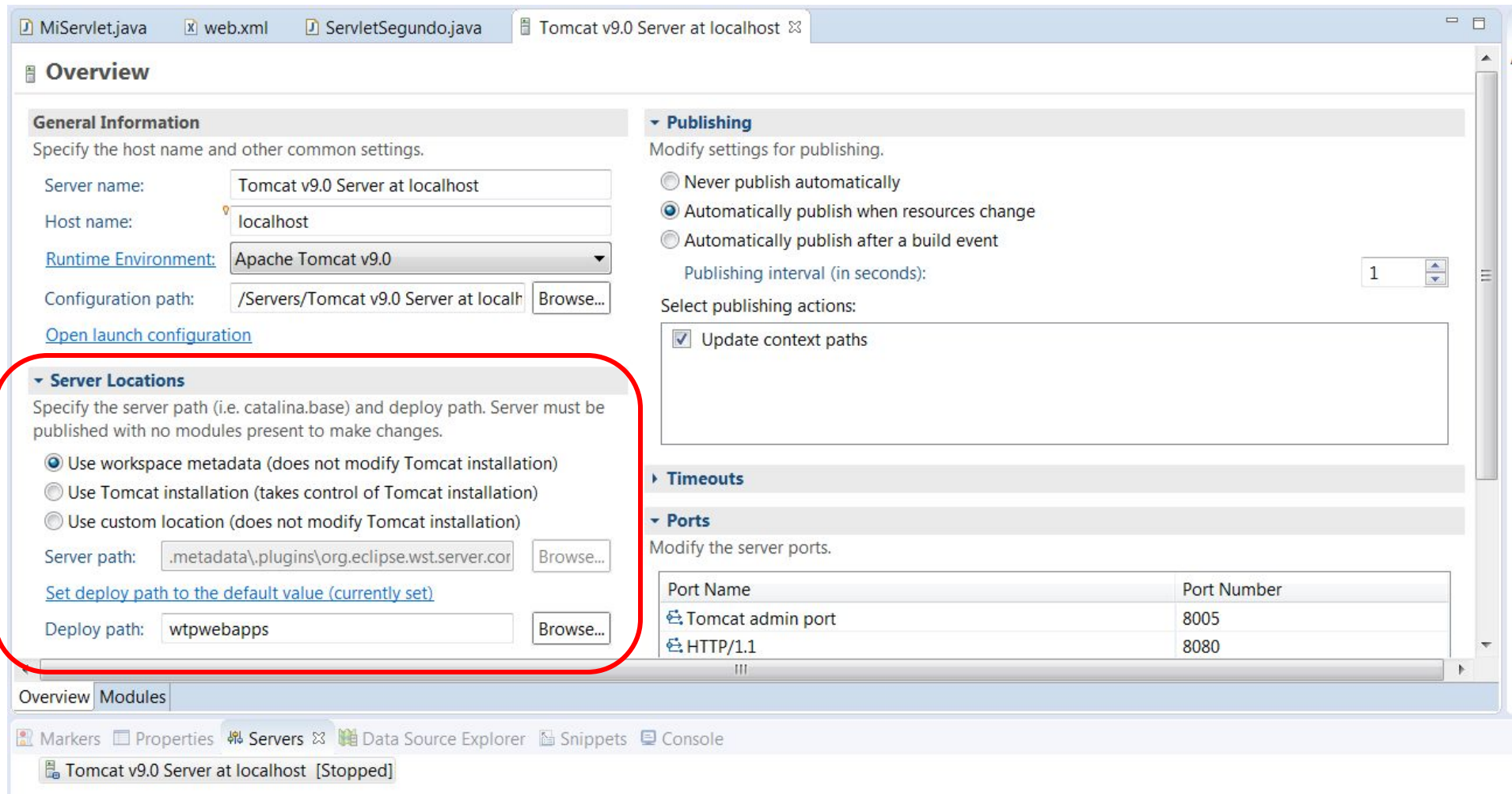


Cuando Tomcat arranca, automáticamente expande a partir de webapps el contenido de cada uno de los archivos .war al formato “desempaquetado”.

Si usamos esta técnica para hacer el “deployment” de nuestra aplicación y necesitamos actualizarla, debemos reemplazar el .WAR y **ELIMINAR** la estructura de directorios expandida y luego re-iniciar Tomcat.

¿Cómo se hace el “deploy” de una aplicación desde Eclipse?

- Cuando se ejecuta una aplicación web en un servidor, se puede configurar el lugar donde se hará el deploy de tal aplicación.



The screenshot displays the Eclipse IDE interface with the 'Overview' tab selected for the 'Tomcat v9.0 Server at localhost'. The 'Server Locations' section is highlighted with a red circle, indicating the configuration for where the application is deployed. The 'Publishing' section shows options for automatic publishing, and the 'Ports' section lists the server's ports.

General Information
Specify the host name and other common settings.

Server name: Tomcat v9.0 Server at localhost
Host name: localhost
Runtime Environment: Apache Tomcat v9.0
Configuration path: /Servers/Tomcat v9.0 Server at localhost/Browse...
[Open launch configuration](#)

Server Locations
Specify the server path (i.e. catalina.base) and deploy path. Server must be published with no modules present to make changes.

☒ Use workspace metadata (does not modify Tomcat installation)
☐ Use Tomcat installation (takes control of Tomcat installation)
☐ Use custom location (does not modify Tomcat installation)

Server path: .metadata\plugins\org.eclipse.wst.server.cor/Browse...
[Set deploy path to the default value \(currently set\)](#)
Deploy path: wtpwebapps/Browse...

Publishing
Modify settings for publishing.

☐ Never publish automatically
☒ Automatically publish when resources change
☐ Automatically publish after a build event
Publishing interval (in seconds): 1
Select publishing actions:
☒ Update context paths

Timeouts

Ports
Modify the server ports.

Port Name	Port Number
Tomcat admin port	8005
HTTP/1.1	8080

Overview Modules

Markers Properties Servers Data Source Explorer Snippets Console

Tomcat v9.0 Server at localhost [Stopped]

Atributos y Alcances



¿Qué es un atributo y dónde pueden guardarse?

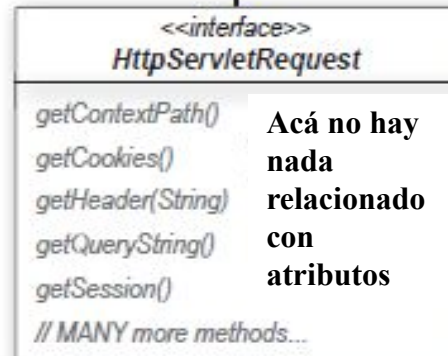
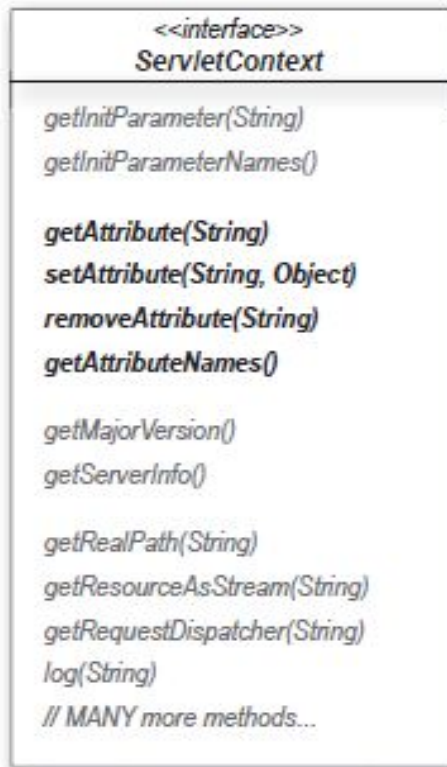
Un atributo es un objeto java guardado dentro de algunos de los siguientes objetos (conocidos como alcances) de la API de servlets: **ServletContext**, **HttpSession** o **HttpServletRequest**.

alcance aplicación	alcance sesión	alcance request
Los objetos ligados a este contexto pueden ser usados por todos los servlets/JSP de la aplicación y duran mientras la aplicación web esté ejecutando.	Los objetos ligados a la sesión de un usuario pueden ser usados por todos los servlets/JSP que accedan a esa sesión y permanecen ahí mientras dure la sesión.	Los objetos ligados al request pueden ser usados por todos los servlets/JSP que dispongan de ese request y permanecen en él, mientras dure el request. Los atributos NO son parámetros.
Se pueden usar los siguiente métodos para ligar, recuperar y eliminar atributos de cualquier alcance. void setAttribute(String nombre, Object attr) Object getAttribute(String nombre) void removeAttribute(String nombre) Enumeration getAttributeNames()		

Notar que el tipo de dato de retorno del método `getAttribute` es `Object` y comúnmente se necesita usar casting para recuperar el tipo original.

Atributos y Alcances

Los atributos pueden ser ligados a uno de los siguientes alcances: **requerimiento**, **sesión** o **aplicación**. Los métodos de la API para ligar atributos son exactamente los mismos.



Acá no hay nada relacionado con atributos



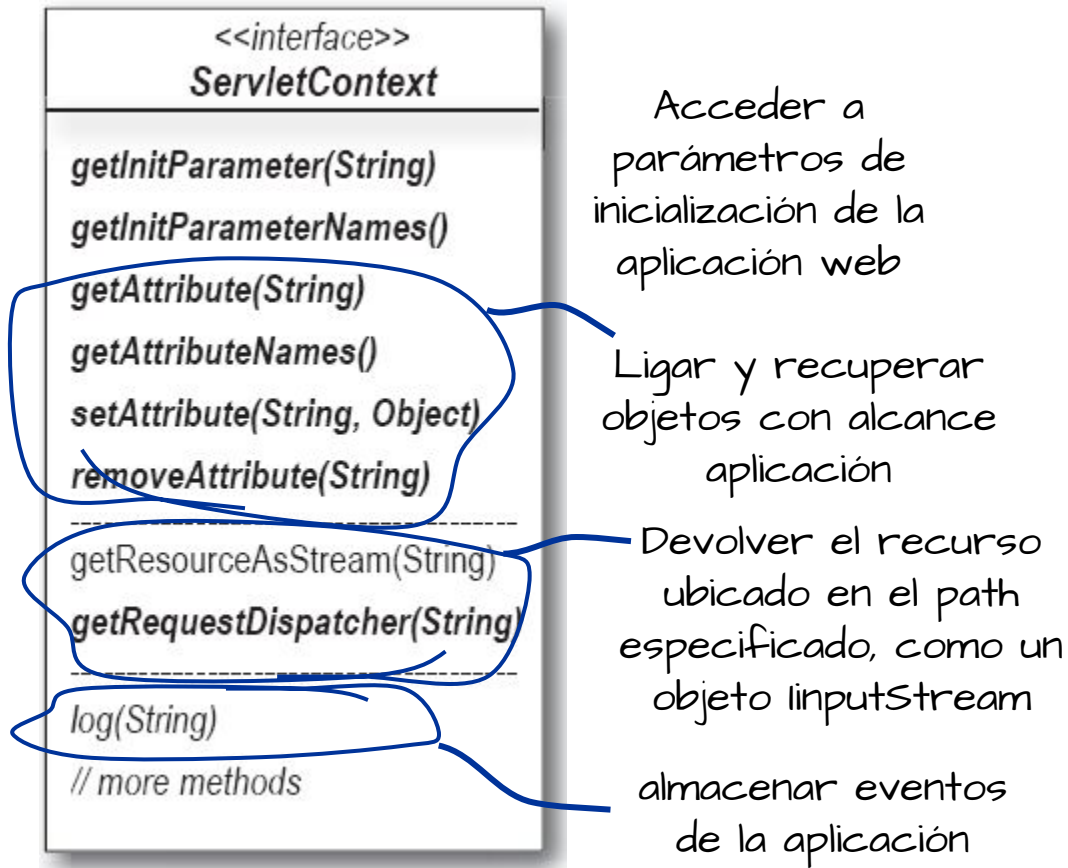
Veremos más adelante

Object getAttribute(String name)
void setAttribute(String name, Object value)
void removeAttribute(String name)
Enumeration getAttributeNames()

ServletContext

¿Qué es?

La interface **ServletContext** define una vista de la aplicación web para los **servlets**. A través del objeto **ServletContext** se pueden lograr varias funcionalidades observadas en el diagrama de la clase.



Cada Contenedor Web provee una implementación específica de la interface **ServletContext**.

Existe un único objeto **ServletContext** por aplicación web ejecutándose en el Contenedor Web.

Los servlets disponen del método **getServletContext()** que retorna una referencia al objeto **ServletContext**. El alcance del objeto **ServletContext** es toda la aplicación.

Para hacer logging de aplicaciones web se recomienda usar una API que permita desde cualquier clase acceder y usar el logging. Ejemplo: Log4j, <http://jakarta.apache.org/log4j> o la API estándar para logging del JDK (`java.util.logging`).

ServletContext

Parámetros de Inicialización de la Aplicación Web

De manera similar a los Servlets, las aplicaciones también pueden tener parámetros de inicialización

```
<?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 . . . id="WebApp_ID" version="2.5">
  <context-param>
    <param-name>email</param-name>
    <param-value>admin@info.unlp.edu.ar</param-value>
  </context-param>
  . . .
</web-app>
```

web.xml

```
package com.servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

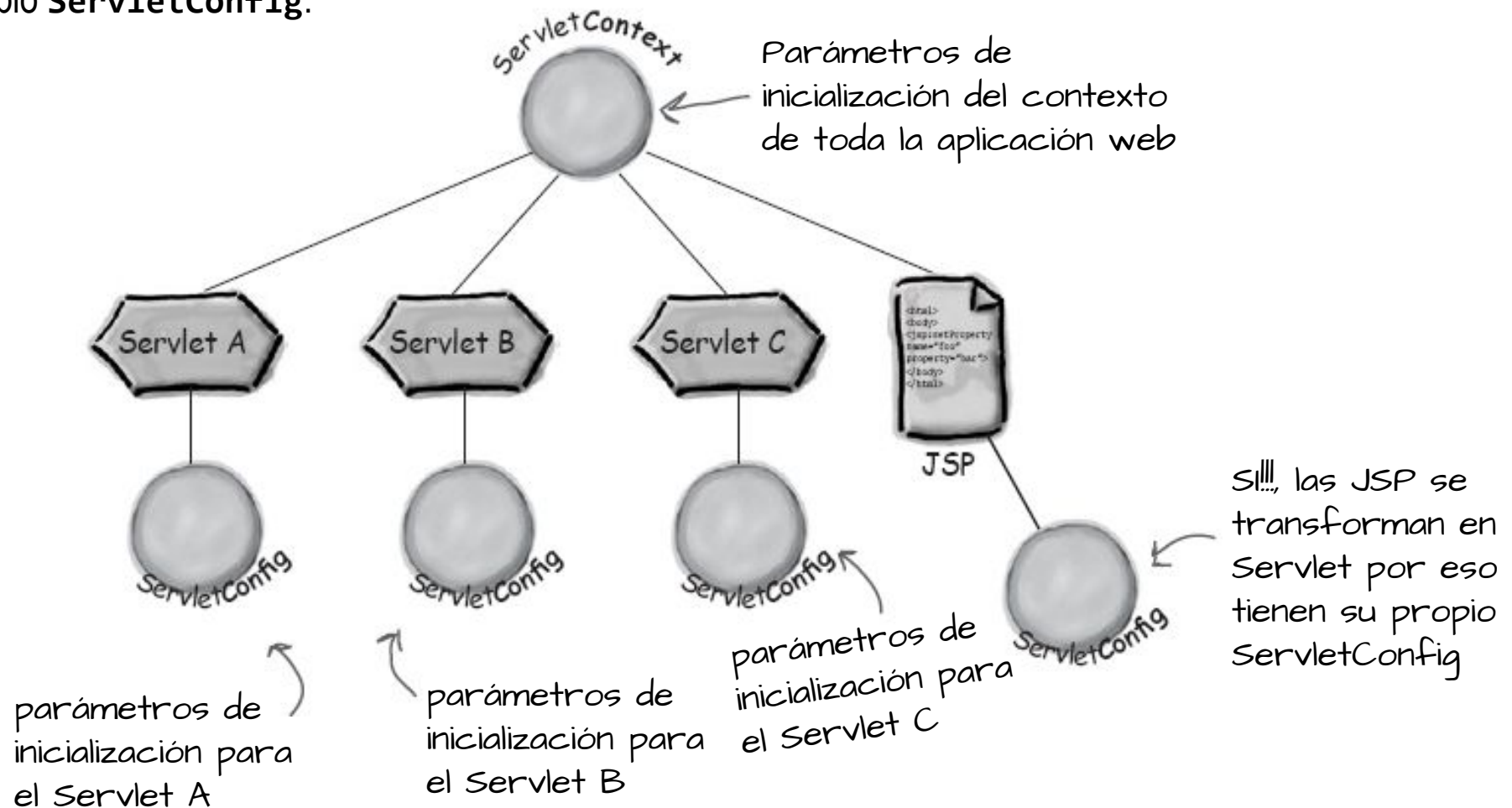
public class PaginaError extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)..  
        resp.setContentType("text/html");  
        PrintWriter out=resp.getWriter();  
        out.println("<html><head>");  
        out.println("<title> Ocurrió un Error </title>");  
        out.println("</head><body>");  
        ServletContext sc = this.getServletContext();  
        String mail = sc.getInitParameter("email");  
        out.println("<h1> Error inesperado </h1>");  
        out.println("Por favor, repórtelo a: " + mail);  
        out.println("</body></html>");  
    }  
}
```

Los parámetros de inicialización de la aplicación web pueden ser accedidos desde todas las componentes web.

Por ejemplo desde un Servlet

Parámetros de inicialización de Servlets y del Contexto

Hay un único **ServletContext** para toda la aplicación web y todas las componentes lo comparten. El contenedor crea el objeto **ServletContext** cuando se carga la aplicación. Cada servlet en la aplicación tiene su propio **ServletConfig**.



Los parámetros de inicialización

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/j2ee/dts/web-app_2_3.dtd">
<web-app>
```

```
  <context-param>
    <param-name>email</param-name>
    <param-value>admin@info.unlp.edu.ar</param-value>
  </context-param>
```

del Contexto o de la Aplicación

`getServletContext().getInitParameter("email")`

. . .

```
<servlet>
  <servlet-name>ServletFecha</servlet-name>
  <servlet-class>misServlets30.ServletFecha</servlet-class>
  <init-param>
    <param-name>dia</param-name>
    <param-value>Hoy es:</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>ServletFecha</servlet-name>
  <url-pattern>/ServletFecha</url-pattern>
</servlet-mapping>
. . .
</web-app>
```

del Servlet ServletFecha

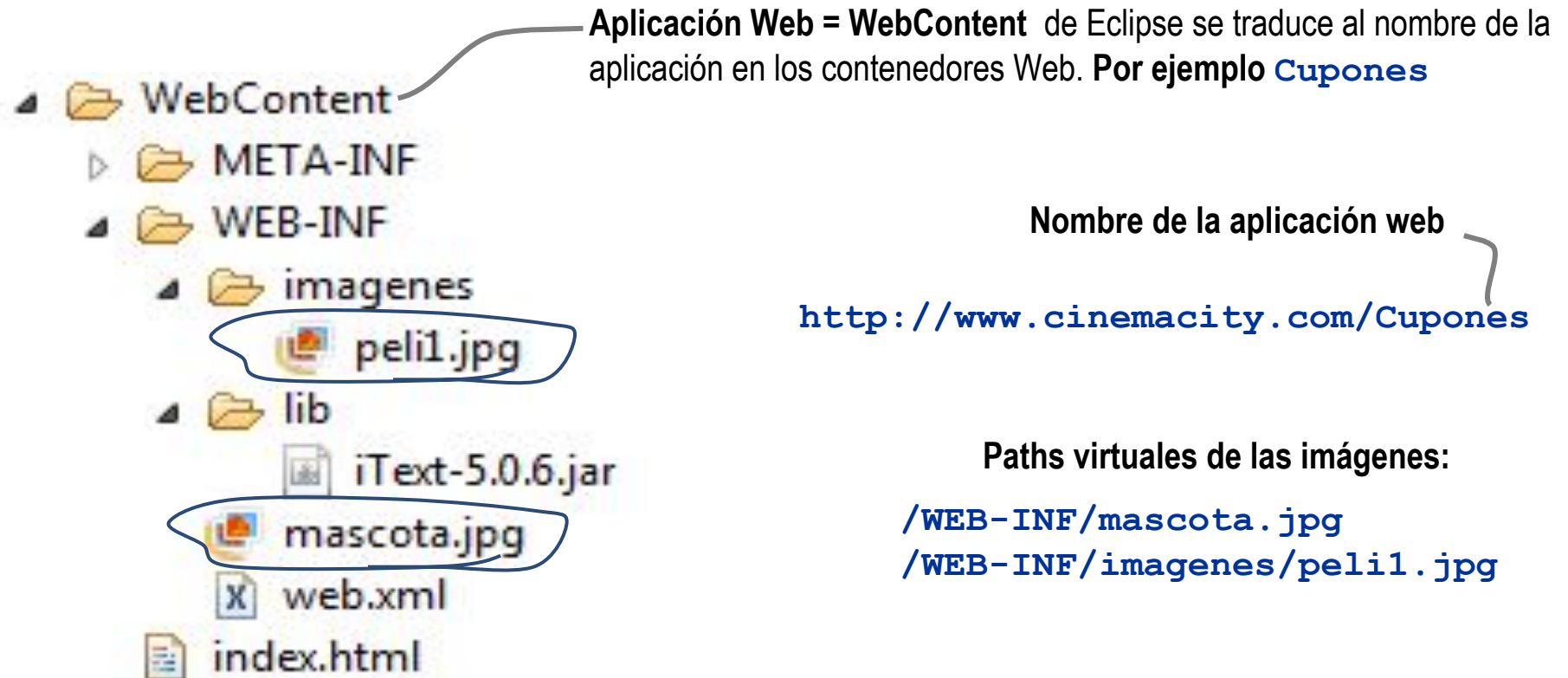
`getServletConfig().getInitParameter("dia")`

web.xml

ServletContext

Recursos web estáticos

El objeto `ServletContext` también provee acceso a la jerarquía de documentos estáticos que forman parte de la aplicación Web, como por ej. archivos TXT, GIF, JPEG.



Todos los recursos de una aplicación web son abstraídos en directorios virtuales a partir de la raíz de la aplicación web. Un directorio virtual comienza con “/” y continúa con un path virtual a directorios y recursos.

ServletContext

Recursos web estáticos

La forma correcta para leer recursos de una aplicación web es usando los métodos **getResource()** o **getResourceAsStream()**. Estos métodos garantizan que el servlet siempre tendrá acceso al recurso deseado. Aún cuando el *deploy* de la aplicación web se realice en múltiples servidores o en un archivo WAR.

El objeto **ServletContext** provee acceso a la jerarquía de documentos estáticos a través de los siguientes métodos:

- **URL getResource(String path)**

Devuelve la URL al recurso que coincide con el **path** dado como parámetro. El **path** debe comenzar con "/" y es relativo a la raíz de la aplicación web.

```
URL unaUrl = getServletContext().getResource("/WEB-INF/mascota.jpg");
```

- **InputStream getResourceAsStream(String path)**

Devuelve el recurso ubicado en el **path** especificado como parámetro, como un objeto **InputStream**. Los datos en el **InputStream** pueden ser de cualquier tipo y longitud. El path debe empezar con "/" y es relativo a la raíz de la aplicación web.

```
InputStream is=this.getServletContext().getResourceAsStream("/WEB-INF/salas.txt")
```

ServletContext

Recursos web estáticos – Archivos de texto

Utilizando el método `getResourceAsStream()` podríamos leer un archivo de texto ubicado por ejemplo en el directorio **WEB-INF** de la aplicación, de la siguiente manera:

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

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws . . .{

        String filename = "/WEB-INF/datos.properties";
        ServletContext context = this.getServletContext();

        // Obtenemos un InputStream usando ServletContext.getResourceAsStream()
        InputStream is = context.getResourceAsStream(filename);
        if (is != null) {
            InputStreamReader isr = new InputStreamReader(is);
            BufferedReader reader = new BufferedReader(isr);
            String text;
            // Leemos por linea y escribimos en la salida
            while ((text = reader.readLine()) != null) {
                // Acá se obtiene la linea de texto en la variable text
                // Por ejemplo -> jdbc.driver=com.mysql.jdbc.Driver }
            }
        }

        protected void doPost(HttpServletRequest request, HttpServletResponse response) throws . . . {
            doGet(request, response);
        }
    }
}
```

datos.properties

```
jdbc.driver= "com.mysql.jdbc.Driver"
jdbc.user= "admin"
jdbc.password = "tps@admin2017"
jdbc.url="jdbc:mysql://localhost:3306/myBase"
```

ServletContext

Recursos web estáticos – Archivos de Propiedad

Un archivo de propiedades está formado por líneas de texto de la forma **clave=valor**. La extensión de estos archivos es **properties**.

La clase **ResourceBundle**, del paquete **java.util**, facilita la lectura de archivos de propiedades. Esta clase tiene un método **static** **getBundle(String name)** que obtiene un objeto **ResourceBundle** usando el nombre del archivo.

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

    PrintWriter out = response.getWriter();
    ResourceBundle rb = ResourceBundle.getBundle("recursos.datos");
    Enumeration<String> claves = rb.getKeys();
    while (claves.hasMoreElements()) {
        String clave = (String) claves.nextElement();
        out.println(rb.getString(clave)+" ");
    }
    // out.println(rb.getString("jdbc.driver"));
}
```

en este caso, el archivo `datos.properties` debe ubicarse en la carpeta `recursos`

Estos archivos de texto son comúnmente usados para guardar datos de configuración de una aplicación. A continuación se muestra el contenido de un archivo llamado **datos.properties** y ubicado en **src/recursos**

datos.properties

```
jdbc.driver= "com.mysql.jdbc.Driver"
jdbc.user= "admin"
jdbc.password = "https@admin2017"
jdbc.url="jdbc:mysql://localhost:3306/myBase"
```

ServletContext

Listeners de Contexto

¿Cómo haríamos para correr algún código antes de que un Servlet -o JSP- pueda responder a un requerimiento? ¿Podemos saber cuando la aplicación web arranca?

Necesitamos de un objeto java que **inicialice la aplicación web**. Podemos crear una clase separada (ni Servlet, ni JSP), que pueda escuchar por 2 eventos del ciclo de vida de la aplicación: **creación** y **destrucción** de la aplicación web. Para lograr esto, la clase debe implementar la interface **ServletContextListener** => Listeners de Contexto

```
package misServlet;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class InicializaSalas implements ServletContextListener {

    public void contextDestroyed(ServletContextEvent sce) { }

    public void contextInitialized(ServletContextEvent sce) {
        // Se leen parametros de inicializacion de la aplicación
        String peli1 = sce.getServletContext().getInitParameter("sala1");
        String peli2 = sce.getServletContext().getInitParameter("sala2");
        String peli3 = sce.getServletContext().getInitParameter("sala3");
        // Se guardan en el contexto, las peliculas en Cartelera
        ServletContext contexto = sce.getServletContext();
        contexto.setAttribute("sala1", peli1);
        contexto.setAttribute("sala2", peli2);
        contexto.setAttribute("sala3", peli3);
    }
}
```



Se ligan los atributos al contexto de la aplicación. Ahora cualquier componente puede recuperarlos usando por ejemplo:
`getServletContext().getAttribute("sala1")`

ServletContext

Listener de contexto – configuración en el WEB.XML

Los **servlet listeners** deben configurarse en el archivo `web.xml` y de esta manera el Contenedor Web se enterará de su existencia. Para publicarlos se utiliza el tag **<listener>**.

`web.xml`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web
Application 2.3//EN" "http://java.sun.com/j2ee/dts/web-app_2_3.dtd">
<web-app>
  <context-param>
    <param-name>sala1</param-name>
    <param-value>"Parasitos"</param-value>
  </context-param>
  <context-param>
    <param-name>sala2</param-name>
    <param-value>"Joker"</param-value>
  </context-param>
  <context-param>
    <param-name>sala3</param-name>
    <param-value>"El irlandés"</param-value>
  </context-param>
  <listener-class>mislisteners.InicializaSalas</listener-class>
</web-app>
. . .
</web-app>
```

Es posible declarar múltiples listeners.
El orden en que son declarados determina el orden en el que son invocados por el Contenedor Web.

```
package misListeners;

@WebListener
public class InicializaSalas implements ServletContextListener {

    public void contextDestroyed(ServletContextEvent sce) {
        ..
    }
    public void contextInitialized(ServletContextEvent sce) {
        ...
    }
}
```

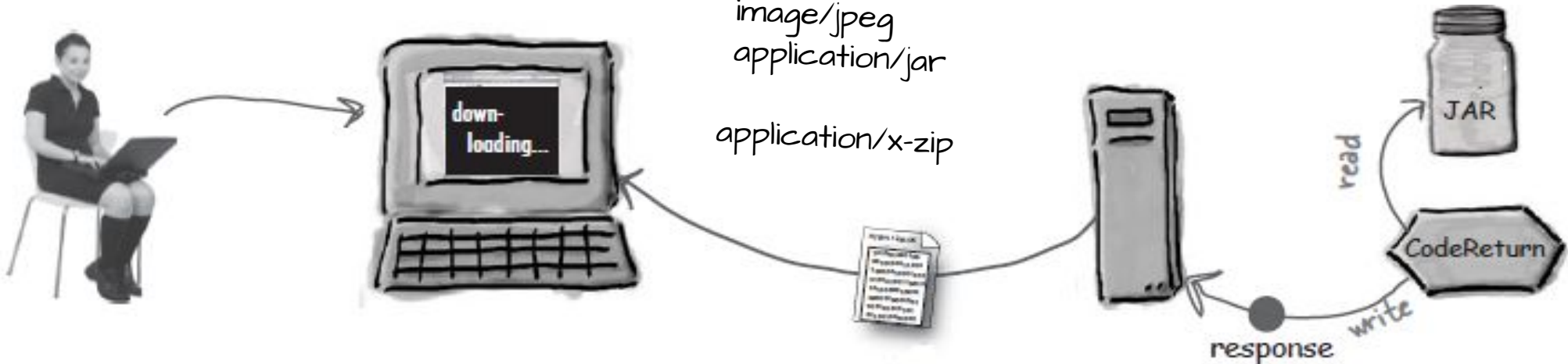
Si usamos anotaciones evitamos el tag **<listener>** en el archivo `web.xml`

Servlets

Tipos de respuestas

Los tipos mas comunes de MIME son:

text/html
application/pdf
video/quicktime
image/jpeg
application/jar
application/x-zip



Tenemos un JAR con código fuente java que queremos enviárselo a nuestros clientes usando un Servlet.

¿Qué tipo MIME usamos?

¿Qué objeto usamos para escribir la respuesta?

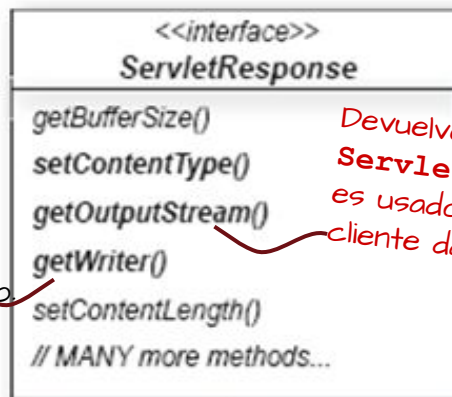
Las interfaces de programación

HttpServletResponse y ServletContext

Vimos que el objeto **HttpServletResponse** tiene métodos que retornan objetos donde se puede escribir el contenido que se enviará en la respuesta.

También existe la interface **ServletContext** que permite obtener referencias a URLs de recursos web.

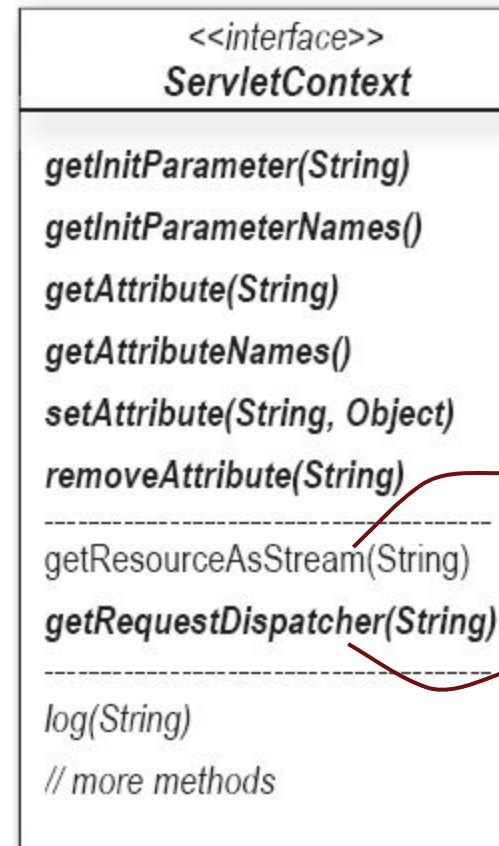
ServletResponse interface (javax.servlet.ServletResponse)



Devuelve un objeto **PrintWriter** que, es usado por el servlet para escribir la respuesta como texto

Devuelve un objeto **ServletOutputStream** es usado para enviar al cliente datos binarios.

HttpServletResponse interface (javax.servlet.http.HttpServletResponse)



Cada Contenedor Web provee una implementación específica de esta interface. La funcionalidad es idéntica, no depende de la implementación, está establecida en la interface.

Devuelve una referencia a un objeto **InputStream** para acceder a un recurso web indicado en el parámetro

Devuelve un objeto **RequestDispatcher** al que se le puede delegar un requerimiento.

Servlets

Tipos de respuestas – Retornando una imagen

Para enviar datos binarios desde un servlets se debe usar el objeto **OutputStream** retornado por el método **getOutputStream()**. En este caso el archivo está en memoria.

```
// imports
@WebServlet("/ServletImage")
public class ServletImage extends HttpServlet {

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

        OutputStream outputStream = response.getOutputStream();
        BufferedImage image = new BufferedImage(500, 300, BufferedImage.TYPE_INT_BGR);
        Graphics2D graphics = image.createGraphics();
        graphics.setBackground(Color.WHITE);
        graphics.clearRect(0, 0, 500, 300);
        graphics.setFont(new Font("TimesRoman", Font.BOLD, 14));
        graphics.setColor(Color.DARK_GRAY);
        graphics.drawString("    Comienza tu día con una sonrisa y verás,", 30, 30);
        graphics.drawString("lo divertido que es andar por ahí desentonando", 30, 45);
        graphics.drawString("                                con todo el mundo!!!", 30, 60);

        BufferedImage img = ImageIO.read(this.getServletContext().getResourceAsStream("/WEB-INF/mafalda.png"));
        graphics.drawImage(img, 40, 80, null, null);

        javax.imageio.ImageIO.write(image, "png", outputStream);
        outputStream.close();

    } . . .
}
```



Transferir el control

sendRedirect() del objeto HttpServletResponse

Algunas veces el servlet puede redireccionar el requerimiento a otro recurso del mismo contenedor web o a una URL de otro dominio.

El sendRedirect() hace trabajar al navegador

¿Cómo lo hace?

El servlet invoca al método `sendRedirect(String url)` sobre la respuesta. La respuesta HTTP lleva el código 302 que indica "El recurso que está buscando el cliente fue temporariamente movido". El navegador obtiene la respuesta, ve el código de estado "302" genera un nuevo requerimiento usando la URL que recibió como parámetro (el usuario puede observar en la barra del navegador que la URL cambia) y finalmente muestra una página al usuario que no fue la que el originalmente pidió.

<<interface>>
HttpServletResponse
<code>addCookie()</code>
<code>addHeader()</code>
<code>encodeURL()</code>
<code>sendError()</code>
<code>setStatus()</code>
<code>sendRedirect()</code>
// MANY more methods...

```
public void doPost (HttpServletRequest request, HttpServletResponse response) {  
    String person = request.getParameter("name");  
    if (person==null) {  
        response.sendRedirect("/app/DatosMal.html");  
        return;  
    }  
    . . .  
}
```

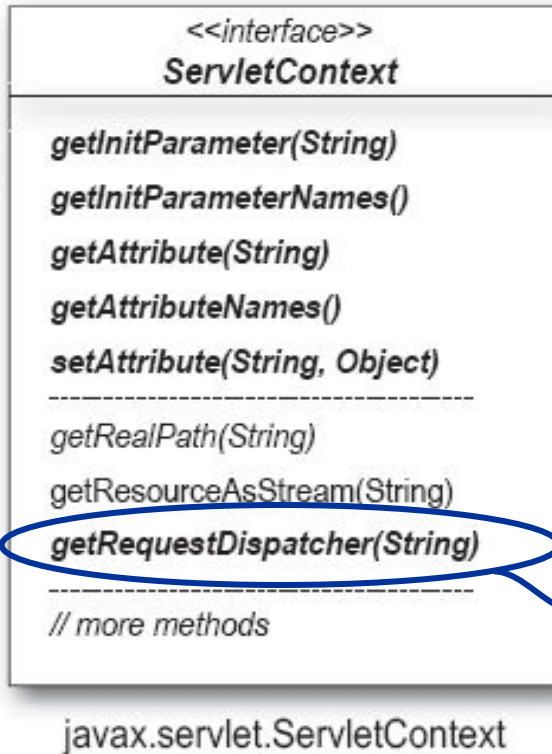
La url que se quiere que el navegador use para crear el requerimiento. Es la que verá el cliente

Se puede usar una url relativa o absoluta:

"http://www..."

Transferir el control

`forward()` del objeto `ServletContext`



Usar el método `forward()` del objeto `RequestDispatcher` es otro mecanismo para transferir el control.

A diferencia del redireccionamiento de la respuesta, este mecanismo no requiere de ninguna acción por parte del cliente ni del envío de información extra entre el cliente y el servidor.

El proceso íntegro de delegación del requerimiento se realiza del lado del servidor. Además, este mecanismo permite pasar el requerimiento a otro servlet para que continúe el procesamiento y responda al cliente.



Instancia un `RequestDispatcher` para un servlet

```
public class ForwardServlet extends HttpServlet {  
    public void doGet(HttpServletRequest request, HttpServletResponse response) {  
        . . .  
        RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/mostrar") ;  
        if (dispatcher != null) {  
            request.setAttribute("hora", new Date());  
            dispatcher.forward(request, response) ;  
        }  
        . . .  
    }  
}
```

Con el `forward()` trabaja más el servidor

Transferir el control

include() del objeto ServletContext

<<interface>>
RequestDispatcher

forward(ServletRequest, ServletResponse)
include(ServletRequest, ServletResponse)

El objeto **RequestDispatcher** también cuenta con el método **include()** que se utiliza de manera similar que el **forward()** y que incluye contenido del lado del servidor en la respuesta que se está generando. El servlet que funciona como receptor del método **include()** – en el ejemplo los servlets **header** y **footer**– tienen acceso al requerimiento y a la respuesta original.

```
public class IncludeServlet extends HttpServlet {  
    public void doGet(HttpServletRequest req, HttpServletResponse res) {  
        . . .  
        RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/header") ;  
        if (dispatcher != null)  
            dispatcher.include(request, response);  
        . . .  
        dispatcher = getServletContext().getRequestDispatcher("/footer") ;  
        if (dispatcher != null)  
            dispatcher.include(request, response);  
    }  
    . . .  
}
```

Acá se podrían setear atributos en el request antes de delegar