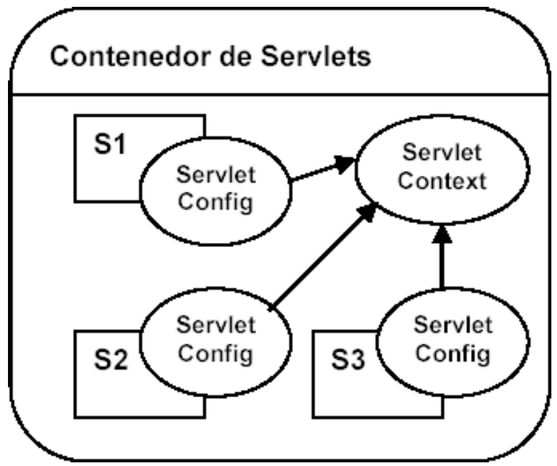
Componentes Web

Sesión 4 - Contexto global de la aplicación web

Índice

Contexto compartido

- Cada contexto (aplicación web) tiene asociado un único objeto `ServletContext` que lo representa
  - Compartido por todos los componentes de la aplicación web
  - En sistemas distribuidos existe uno por cada VM



- Se obtiene desde dentro de cualquier servlet con

```
ServletContext contexto = getServletContext();
```

Funciones de ServletContext

- Compartir información (objetos Java) entre los distintos componentes de la aplicación web
- Leer los parámetros de inicialización de la aplicación web
- Leer recursos estáticos de la aplicación web
- Hacer redirecciones de la petición a otros recursos
- Escribir *logs*

Atributos del contexto

- Nos permiten compartir información entre los distintos elementos de la aplicación web
- Esta información se almacena en forma de objetos Java
- Establecemos un atributo con

```
contexto.setAttribute(nombre, valor);
```

- Este objeto podrá ser leído desde cualquier componente de la aplicación web con

```
Object valor = contexto.getAttribute(nombre);
```

Listeners del contexto

- Escuchan eventos producidos en el objeto `ServletContext`
- Distinguimos dos tipos:
  - `ServletContextListener`

    Notifica la creación y la destrucción del contexto
  - `ServletContextAttributeListener`

    Notifica los cambios en los atributos del contexto

    Creación de atributos

    Eliminación de atributos

    Modificación de atributos

- Deben anotarse con `@WebListener`
  - En versiones previas se declaran en `web.xml`

ServletContextListener

- Por ejemplo, nos permite inicializar estructuras de datos al crearse el contexto.

```
@WebListener
public class MiContextListener implements ServletContextListener {

    public void contextDestroyed(ServletContextEvent sce) {
        // Destruccion del contexto
    }

    public void contextInitialized(ServletContextEvent sce) {
        // Inicialización del contexto
    }
}
```

ServletContextAttributeListener

```
@WebListener
public class MiContextAttributeListener
                    implements ServletContextAttributeListener {

    public void attributeAdded(ServletContextAttributeEvent scae) {
        // Se ha añadido un nuevo atributo
    }

    public void attributeRemoved(ServletContextAttributeEvent scae) {
        // Se ha eliminado un atributo
    }

    public void attributeReplaced(ServletContextAttributeEvent scae) {
        // Un atributo ha cambiado de valor
    }
}
```

Declaración programática en Servlet 3.0

- Podemos declarar o configurar *servlets* en la inicialización del contexto

```
@WebListener
public class MiListener implements ServletContextListener {
  public void contextInitialized (ServletContextEvent sce) {
    ServletContext sc = sce.getServletContext();

    // Declara un nuevo servlet y lo configura
    ServletRegistration servletNuevo = sc.addServlet("miServlet",
      "org.especialistajee.servlet.MiServlet");
    servletNuevo.addMapping("/UrlServlet");

    // Obtiene un servlet ya declarado para configurarlo
    ServletRegistration sr = sc.addServlet("otroServlet");
    sr.addMapping("/UrlOtroServlet");
    sr.setInitParameter("param1", "valor1");
  }
}
```

Parámetros globales

- Se pueden declarar parámetros globales del contexto en el descriptor de despliegue

```
<context-param>
    <param-name>param1</param-name>
    <param-value>valor1</param-value>
</context-param>
```

- Podemos leer estos parámetros a través del objeto `ServletContext`

```
String valor = contexto.getInitParameter(nombre);
```

Lectura de recursos

- Podemos leer recursos estáticos del contexto con

```
InputStream in = contexto.getResourceAsStream(ruta);
```

- Donde la ruta debe comenzar por "/"
  - Es una ruta relativa a la raíz del contexto
  - P.ej. "`/index.htm`" busca el fichero `index.htm` en el directorio raíz del contexto

- Leerá cualquier recurso como estático
  - P.ej, si leemos un recurso "`/index.jsp`" leerá el código fuente del JSP, no procesará este código
  - Si queremos que sea procesado el contenido dinámico deberemos utilizar los métodos `include` o `forward`

Tipos de redirecciones

- Redirecciones en el cliente
  - El servidor envía cabecera de redirección al cliente en la respuesta indicando la URL a la que debe redirigirse
  - El navegador de la máquina cliente realiza una nueva petición a esta URL
- Redirecciones en el servidor
  - *Forward*

    El servidor sirve un recurso distinto al indicado en la URL de forma transparente para el cliente
  - *Include*

    Se incluye el contenido de un recurso dentro del contenido generado por nuestro servlet

Redirecciones en el cliente

- Realizamos estas redirecciones enviando una cabecera de redirección en la respuesta

```
response.sendRedirect(url);
```

- Será el cliente el responsable de interpretar esta cabecera y realizar una nueva petición a la dirección indicada
  - Se mostrará la nueva URL en la barra de dirección del navegador

- Sólo debemos llamar a este método antes de haber generado la respuesta del servlet
  - Si no, se producirá una excepción `IllegalStateException`

Redirecciones en el servidor

- Obtenemos un objeto `RequestDispatcher` a partir del contexto

```
RequestDispatcher rd = contexto.getRequestDispatcher(ruta);
```

- La ruta comenzará con "/"
  - Relativa al contexto
- Realizamos la redirección con

```
rd.forward(request, response);
```

- Esto debe hacerse antes de haberse comenzado a escribir la respuesta
  - Si no, se producirá una excepción `IllegalStateException`

Inclusiones

- Podemos utilizar el `RequestDispatcher` para incluir el recurso indicado dentro de la respuesta generada por nuestro servlet

```
rd.include(request, response);
```

- En este caso podremos haber escrito ya contenido en la respuesta
  - El recurso solicitado se escribirá a continuación de este contenido
  - Podremos seguir escribiendo contenido tras la inclusión del recurso

- Tanto con `forward` como con `include` se procesarán los recursos dinámicos solicitados (por ejemplo los JSP)

Escritura de logs

- Podemos escribir mensajes en el fichero de *logs* de Tomcat
  - Registros de eventos sucedidos en nuestra aplicación
  - Depuración de errores
- Escribimos los *logs* utilizando el objeto del contexto

```
contexto.log(mensaje);
```

Inyección de dependencias

- Unos objetos dependen de otros
  - Por ejemplo, un servlet puede depender de una fuente de datos para obtener los datos a mostrar en el HTML

- Tradicionalmente un objeto debe obtener sus dependencias

- El patrón de Inyección de Dependencias (*DI*) invierte este comportamiento
  - Las dependencias son inyectadas en el objeto
  - Reduce el acoplamiento entre objetos
  - Podemos reemplazar la implementación de forma sencilla

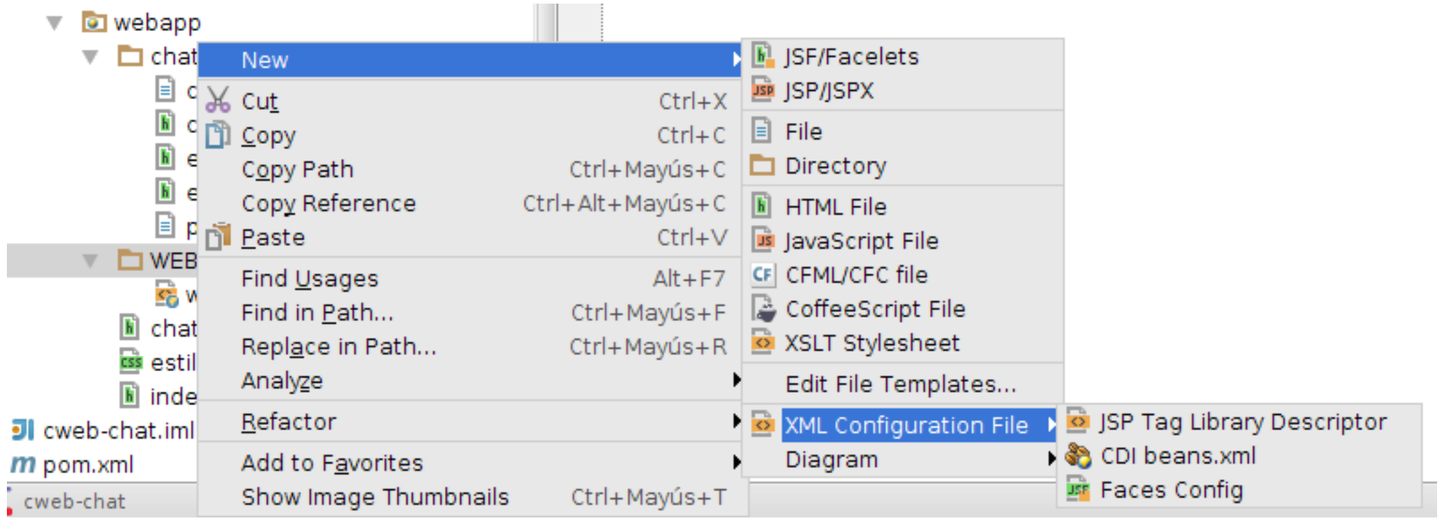
    Por ejemplo, cambiar una implementación por un *mock*

JSR-299 y Weld

- En Java SE 6 la inyección de dependencias se integra mediante la especificación JSR-299 (*Contexts and Dependency Injection*)
  - Su implementación de referencia se llama Weld

    http://seamframework.org/Weld

- Configuración de Weld
  - Definir fichero `beans.xml` en `WEB-INF` (puede estar vacío)



Managed beans

- Casi cualquier tipo de objeto puede ser inyectado
  - POJOs, contextos de persistencia, fuentes de datos, etc

- Los objetos inyectados son denominados *managed beans*

- Para que un objeto pueda ser un *managed bean* debe cumplir
  - No puede ser una clase interna, a no ser que sea `static`
  - Debe tener un constructor sin parámetros
  - Puede no tener un constructor sin parámetros, siempre que alguno de sus constructores se etiquete con `@Inject`

Inyección de objetos

- Dado el siguiente *managed bean*

```
public class HolaMundo {
    public String saluda(String nombre) {
        return "Hola " + nombre;
    }
}
```

- Podemos inyectarlo en cualquier clase mediante `@Inject`

```
@WebServlet(urlPatterns = "/miServlet")
public class MiServlet extends HttpServlet {
    @Inject
    private HolaMundo holaMundo;
}
```

- El objeto inyectado se instanciará automáticamente

Ámbito de los *beans*

- Podemos etiquetar los *beans* con distintos ámbitos
  - El tiempo de vida del *bean* dependerá del ámbito indicado

Ámbito	Descripción
`@RequestScoped`	Se mantiene durante el tiempo que dure la petición (`request`) actual. Sólo será accesible por la petición actual.
`@SessionScoped`	Se mantiene durante el tiempo que dure la sesión (`session`) actual. Será accesible por todas las peticiones que se hagan dentro de la sesión actual.
`@ApplicationScoped`	Se mantiene durante toda la vida de la aplicación web (contexto). Será accesible por todas las peticiones de todas las sesiones.
`@Dependent`	Es el valor por defecto. Se mantiene con vida mientras exista el objeto en el que se ha inyectado.

Ejemplo de ámbito

- Por ejemplo, el siguiente bean sólo se mantendrá con vida mientras dure la sesión actual

```
@RequestScoped
public class HolaMundo {
    public String saluda(String nombre) {
        return "Hola " + nombre;
    }
}
```

- En caso de los *beans* en el ámbito de sesión, deberemos hacerlos `Serializable`
  - Esto es necesario porque las sesiones pueden moverse entre máquinas virtuales en caso de sistemas distribuidos

Definición de alternativas

- Podemos definir implementaciones alternativas

```
public class FuenteDatos implements IFuenteDatos
```

```
@Alternative
public class MockFuenteDatos implements IFuenteDatos
```

- Decidimos la implementación a inyectar en `beans.xml`

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
                  http://java.sun.com/xml/ns/javaee
                  http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
    <alternatives>
        <class>es.ua.jtech.MockFuenteDatos</class>
    </alternatives>
</beans>
```

- Una implementación puede reemplazar a la existente

```
@Specializes
public class MockFuenteDatos extends FuenteDatos
```

¿Preguntas?