ESCUELA DE SISTEMAS Y TECNOLOGÍAS

Transparencias de **ANALISTA DE SISTEMAS**

Edición 2020 Materia: **Java Web**

TEMA: Servlets

Agenda

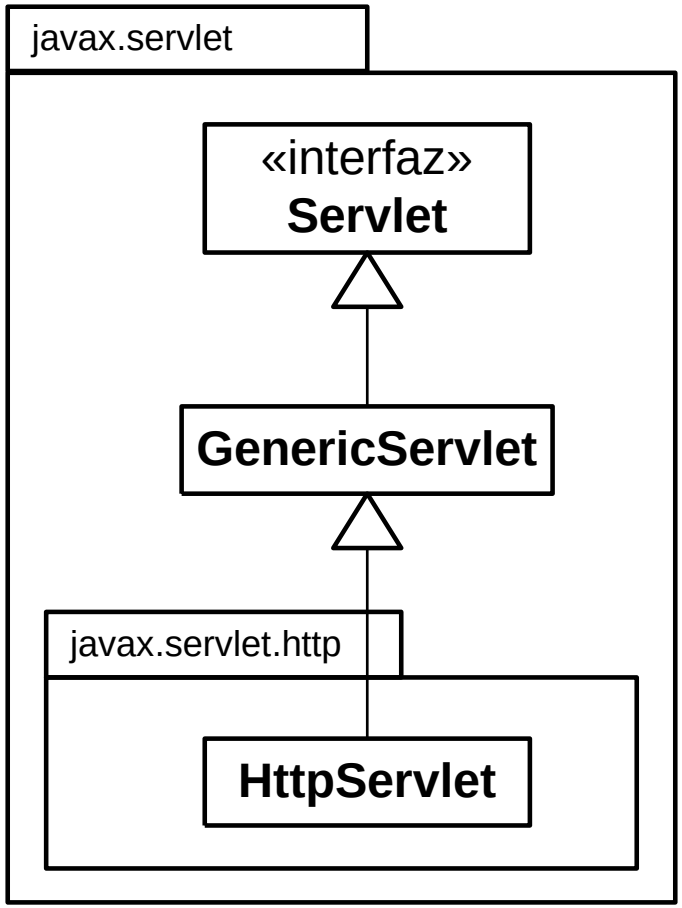
- ¿Qué es un Servlet?
- Utilizar sólo Servlets
- Ciclo de Vida de un Servlet:
- Inicialización
- Métodos de Servicio
- Destrucción
- Manejo de Estado
- Invocando otros Recursos
- Filtros

Que es un Servlet? (1)

- Un Servlet es una clase Java que se utiliza para extender las capacidades de un servidor que aloja aplicaciones accedidas mediante el modelo de programación de pedido - respuesta (request - response programming model).
- Típicamente se utilizan para extender los servicios provistos por Servidores Web.
- Para aplicaciones Web la tecnología Java Servlets define clases Servlet específicas para el protocolo HTTP.
- Los paquetes **javax.servlet** y **javax.servlet.http** proveen clases e interfaces para escribir clases servlet.

Que es un Servlet? (2)

- Todo servlet debe implementar la interfaz **Servlet** que define operaciones para el manejo del ciclo de vida del mismo por parte del servlet container.
- Para implementar un servlet genérico se utiliza o extiende la clase **GenericServlet** mientras que para un servlet para Web la clase **HttpServlet** provee métodos como **doGet(...)** y **doPost(...)** que manejan pedidos HTTP.



Que es un Servlet? (3)

- Al ser código Java “puro”, los servlets son adecuados para colocar código que represente cierta complejidad y que no interactúe directamente con el usuario (es decir que no capture ni muestre datos al usuario).
- No obstante, un servlet puede generar una respuesta en formato HTML, pero para esto la respuesta se debe generar como texto (**String**) dentro del servlet.

Utilizar solo Servlets

- Los servlets son la base de las aplicaciones web Java.
- Debido a que son clases Java, tienen todo el poder y acceso a las APIs de Java.
- Por tanto es posible desarrollar completamente una aplicación web utilizando únicamente servlets.
- Esto generalmente no es deseable ya que:
  - Se necesita buen conocimiento de lenguaje Java.
  - Diseñar y cambiar la apariencia de una página web requiere recodificar una clase Java y recompilarla.
  - Es difícil aprovechar herramientas que facilitan el diseño de páginas web (interfaz gráfica)

Ciclo de Vida de un Servlet

- El ciclo de vida de un servlet es manejado por su contenedor.
- Cuando un pedido es mapeado a un servlet, el contenedor realiza los siguientes pasos:
  1. Si no existe ninguna instancia del servlet:
     a. Carga la clase del servlet
     b. Crea una instancia
     c. Inicializa la instancia invocando al método init()
  2. Invoca al método **service**() pasando el objeto ***request*** y el objeto ***response***
- Al momento de finalizar el servlet se invoca al método **destroy**() aunque esto no ocurre luego de cada request, sino que es el container el que decide cuándo finalizar el servlet.

Inicialización (1)

- Luego que el ***servlet container*** carga la clase del servlet y la inicializa, pero antes de atender pedidos, ésta es inicializada mediante la invocación al método **init**()
- Por tanto, para modificar este comportamiento se debe redefinir el método **init**() de la interfaz **Servlet**
- Típicamente el código de inicialización puede:
  - Leer datos de configuración de un origen persistente,
  - Inicializar recursos,
  - Realizar cualquier acción por única vez.

Inicialización (2)

- En caso de no poder completar su inicialización, el servlet debe lanzar una **UnavailableException**
- En muchas ocasiones es útil contar con parámetros de inicialización para servlets dentro del deployment descriptor.
- Estos parámetros se definen (declarativamente) dentro del archivo web.xml y son luego utilizados (mediante código) desde la clase servlet.

Métodos de Servicio (1)

- El servicio provisto por un servlet es implementado en el método **service**() de la clase **GenericServlet** o en los métodos **do**…() de la clase **HttpServlet** que se corresponden con los métodos del protocolo Http como **doGet**() y **doPost**() correspondientes a:

Http GET	Http POST
Permite ver los parámetros.	Envía los parámetros dentro del cuerpo del pedido (no se ven en la URL).
Se puede hacer un *bookmark* y un *link* incluyendo los parámetros.	No se puede.
La página se recarga normalmente.	Para recargar la página se solicita la confirmación del usuario.

Métodos de Servicio (2)

- El trabajo habitual de un método de servicio es:
  1. Extraer información del pedido,
  2. Acceder a recursos externos,
  3. Generar una respuesta.
- Ya que el paso 2 variará significativamente en cada caso, y que podrá implicar utilizar cualquier recurso (archivos, BD, clases, APIs, etc.) desde la clase ***servlet***, a continuación se presentará cómo *"1. Extraer información del pedido"* y cómo *"3. Generar una respuesta"*

Métodos de Servicio (3)

- **Extraer información del pedido**:
  - Si bien el pedido (*request*) contiene mucha información (ej: el protocolo utilizado, identificación del cliente y servidor, cookies, la URL solicitada, el header Http, el método Http utilizado, etc.) la información más útil son los parámetros.
  - Para obtener un parámetro se utiliza el método **String getParameter(String)** el cual devuelve el valor del parámetro pasado por parámetro y **null** si no existe

Métodos de Servicio (4)

- **Generar una respuesta:**
  - El primer paso para que el servlet genere una respuesta es establecer el tipo de esa respuesta mediante el método ***setContentType(String)*** de la interfaz **HttpServletResponse**.
  - El tipo de contenido debe ser uno de los definidos por la IANA (http://www.iana.org/assignments/media-types/)
  - Luego se obtiene el flujo (*stream*) de salida mediante el método ***PrintWriter getWriter()*** de la misma interfaz.
  - Finalmente se escribe sobre el flujo de salida mediante el método ***println(String)*** de la clase **java.io.PrintWriter**

Métodos de Servicio (5)

- **Generar una respuesta (cont.):**
  - Otra acción que el servlet puede tomar es redirigir la respuesta hacia otra página JSP o servlet.
  - Esto se logra mediante el método ***sendRedirect(String)*** de la interfaz **HttpServletResponse**.

```
if (...) {
    out.println("...");
    out.println("...");
    out.println("...");
} else {
    response.sendRedirect("UnaPagina.jsp");
}
```

Destrucción (1)

- Cuando el contenedor determina que un servlet debe ser removido (ya sea para reclamar su memoria o cuando se apaga el servidor) el contenedor invoca al método **destroy**(…) de la interfaz **Servlet**.
- Este método debe liberar recursos utilizados por el servlet, por ejemplo aquellos obtenidos en el **init**()
- Ejemplo: liberar la conexión a la BD:

```
public void destroy() {
    con.close();
    con = null;
}
```

Destrucción (2)

- Normalmente, cuando **destroy**(…) es invocado, ya no existen procesos ejecutándose en respuesta a un ***request***.
- De existir (ej: nuevos ***threads*** creados para resolver un ***request***) se debe tener particular cuidado de darles una terminación adecuada.
- Estos se denominan ***long running threads*** y no serán tratados aquí (ver Referencias)

Manejo de Estado (1)

- Para compartir información entre diferentes componentes web, existen cuatro ámbitos (scopes) en los cuales el programador puede colocar información para compartir (de más general a menos):

Ámbito	Clase/Interfaz	Accesible desde
Web Context / Application	javax.servlet.ServletContext	Componentes Web dentro de un Web Context
Session	javax.servlet.http.HttpSession	Componentes Web manejando un pedido de una sesión
Request	javax.servlet.http.HttpServlet Request	Componentes Web manejando un pedido
Page	javax.servlet.jsp.JspContext	La página JSP

Manejo de Estado (2)

- El ámbito **Page** será tratado junto con Java Server Pages y el ámbito **Request** ya fue tratado anteriormente en "*Métodos de Servicio*".
- Por tanto se tratarán los ámbitos **Session** y **Web Context** (*application*).
- Los métodos para definir, obtener y eliminar atributos son, respectivamente:
  - void setAttribute(String name, Object value)
  - Object getAttribute(String name)
  - void removeAttribute(String name)

Manejo de Estado (3)

- Para obtener una referencia al **Web Context** desde un servlet:

```
import javax.servlet.ServletContext;
...
ServletContext context = this.getServletContext();
```

- Para obtener una referencia a la **Session** desde un servlet:

```
import javax.servlet.http.HttpSession;
...
HttpSession session = request.getSession();
```

Manejo de Estado (4)

- A diferencia de la Session, la Application (*Web Context*) puede manipular parámetros de inicialización definidos en el *deployment descriptor*.
- Dichos parámetros se definen por fuera (no tienen relación) con los parámetros de inicialización de los servlets (que también se definían en el *deployment descriptor*).
- Definición (web.xml):

```
<context-param>
      <param-name>parametro</param-name>
      <param-value>valor</param-value>
</context-param>
```

Manejo de Estado (5)

- Para obtener el valor del parámetro se utiliza el método String getInitParameter(String name) del ServletContext.

```
ServletContext context = this.getServletContext();
String valor = context.getInitParameter("parametro");
```

- Notar cómo JavaEE utiliza el término ***Attribute*** para referirse a valores de tipo **Object** que son almacenados tanto en Session o en Application, y el término ***Parameter*** para referirse a valores de tipo **String** que son definidos en el *deployment descriptor*.

Manejo de Estado (6)

- Cada sesión es válida mientras que el tiempo de inactividad no supere el ***timeout*** definido en el *deployment descriptor*.
- Para obtener / modificar el ***timeout*** se utilizan los métodos **int getMaxInactiveInterval()** y **void setMaxInactiveInterval(int seconds)**
- También puede ser importante invalidar una sesión programáticamente (desde código) de forma de eliminar el objeto **HttpSession** (por ejemplo luego de terminar una operación de negocio). Esto se logra mediante el método ***invalidate***() de **HttpSession**

Invocando otros Recursos (1)

- Un servlet tiene 2 formas de invocar a otro recurso:
  - Redirigir el pedido al otro recurso,
  - Incluir el contenido de otro recurso.
- **Redirigir hacia otro recurso**:
  - Si se trata de un **HttpServlet**, se puede redirigir la respuesta hacia otro recurso mediante el método ***void sendRedirect(String url)*** de la interfaz **HttpServletResponse**.
  - Este método genera un **nuevo pedido** que el browser del cliente solicitará al servidor.
  - Por tanto, los parámetros (si los hubiera) se pierden

Invocando otros Recursos (2)

- **Redirigir hacia otro recurso (cont.)**:
  - Otra forma de redirigir hacia otro recurso es mediante el método *void forward(ServletRequest request, ServletResponse response)* de la interfaz **RequestDispatcher** la cual se obtiene mediante el método *getRequestDispatcher(String url)* de la interfaz **ServletRequest**.
  - Este método redirige el mismo **pedido original** (incluyendo sus parámetros y atributos) hacia el recurso identificado mediante la URL (transfiere el control).

Invocando otros Recursos (3)

forward(...)	sendRedirect(...)
Reenvía el mismo pedido al destino, por lo que no le responde al browser.	Solicita un nuevo pedido al browser el cual implicará un nuevo Http GET hacia el nuevo destino.
Sólo redirige a recursos dentro del web container.	Puede redirigir a cualquier recurso, incluso fuera del web container.
La URL no cambia (se ve la original solicitada).	La URL cambia hacia la del nuevo recurso.
Se resuelve internamente dentro del web container.	Se comunica con el cliente para que éste solicite la nueva URL.
Se mantienen los parámetros originales del request.	Se pierden los parámetros (es un nuevo pedido) por lo que de ser necesario, deben ser agregados “a mano”.

Invocando otros Recursos (4)

- **Incluir el contenido de otro recurso**:
  - En lugar de redirigir el pedido hacia otro recurso, es también posible incluir el resultado de otro recurso en la respuesta.
  - Esto también se logra mediante el **RequestDispatcher** pero ahora invocando al método ***void include (ServletRequest request, ServletResponse response)***
  - La invocación a este método sobre otro componente web resulta en enviar el request y el response al otro componente web, ejecutarlo (ej: *doGet* o *doPost*) y regresar.
  - Por tanto, el otro componente web (el incluído) debería únicamente colocar contenido en la respuesta (response) sin modificar su encabezado (header) y sin cerrar el flujo de salida.

Filtros (1)

- Un filtro (***filter***) es un objeto que puede transformar el encabezado (***header***) y/o el contenido (***content***) tanto de pedidos (***request***) como de respuestas (response).
- Un filtro difiere de un componente web (ej: servlet) ya que generalmente no genera una respuesta, sino que filtra la respuesta de un componente web.
- Un filtro provee funcionalidades que pueden ser adjuntadas (***attach***) a cualquier componente web.
- Por tanto, un filtro debe ser lo más independiente posible de los demás componentes web, de forma de poder filtrar varios tipos de componentes

Filtros (2)

- Las funcionalidades principales de un filtro son:
  - Inspeccionar un pedido (request) y actuar en consecuencia.
  - Bloquear un pedido/respuesta.
  - Modificar el encabezado y/o contenido de un pedido o de una respuesta mediante el uso de una versión personalizada del pedido o respuesta original.
  - Interactuar con recursos externos.
- Ejemplos: autenticación, logueo, conversión de imágenes, compresión de datos, encriptación, etc.

Filtros (3)

- Los filtros siguen la arquitectura **Pipes and Filters** que se verá en el curso **Ingeniería de Software**.
- Por tanto, éstos pueden combinarse en lo que se denomina "cadena de filtros" (***filter chain***).
- Cada filtro se aplicará sobre la entrada / salida de un componente web y se invocarán entre sí siguiendo la cadena definida.
- La cadena es definida en el *deployment descriptor* y es instanciada cuando el contenedor carga el componente web.

Filtros (4)

- Los pasos para utilizar un filtro son:
  1. Codificar el filtro (clase Java).
  2. Codificar los pedidos (*request*) y respuestas (*response*) personalizados (clases Java).
  3. Definir la cadena de filtros para cada componente web que se desea afectar (en el *deployment descriptor*). Esto se denomina *filter mapping*.
- Las interfaces involucradas son:
  - javax.servlet.Filter
  - javax.servlet.FilterChain
  - javax.servlet.FilterConfig

Filtros (5)

- Para codificar un filtro se debe implementar la interfaz **Filter**, la cual posee 3 operaciones:
  1. ***void init(FilterConfig config)*** → se invoca al instanciar el filtro y permite acceder a parámetros de inicialización.
  2. ***void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)*** → se invoca para realizar el filtrado (para hacer lo que tiene que hacer) y luego invocar al próximo filtro (sin saber quien es).
  3. ***void destroy()*** → se invoca al momento de destruir la instancia del filtro.

Filtros (6)

- Los filtros pueden realizar **acciones previas** a la invocación del siguiente filtro en la cadena así como realizar **acciones posteriores** (pre-acciones y post-acciones).
- En el caso de necesitar modificar el contenido del request o response, se debe codificar una clase que sea un *wrapper* del *request* o *response* original (Paso 2).
- Este *wrapper* impide, por ejemplo, que el componente web filtrado cierre el flujo de salida del *response* (típicamente con ***out.close()***) de forma que el filtro pueda continuar escribiendo en él.

Filtros (7)

- En forma más general, el mapeo filtro - servlet permite mapear un filtro a varios servlet así como un servlet a varios filtros, como indica la siguiente figura (F son filtros y S servlets):