Ficha 03

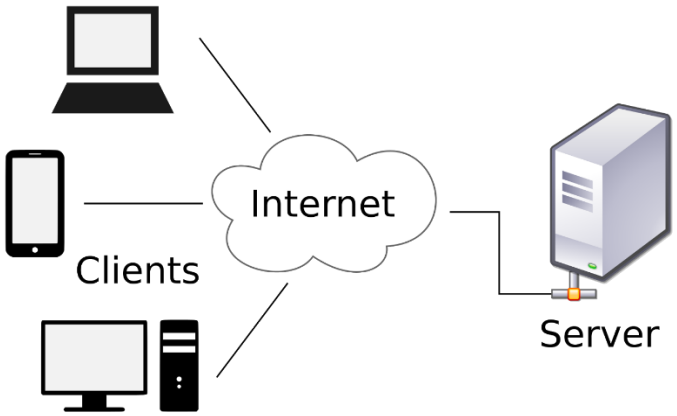
Programación de Componentes del Servidor

1.] Introducción a las Aplicaciones Cliente-Servidor.

En esta ficha de clases nos proponemos introducir los elementos mínimos e indispensables para programar componentes del servidor. Esto nos llevará evidentemente a través de la revisión de los elementos asociados a la plataforma Java EE, alguna breve revisión histórica de la edición empresarial de java, los elementos fundamentales de los servidores de aplicaciones, y repasaremos de forma introductoria y muy brevemente la API de Servlets de Java EE.

La arquitectura cliente-servidor es un patrón de diseños de software que divide las tareas a implementar entre un extremo llamado **cliente,** que interactúa con el usuario o los agentes externos y realiza las peticiones al otro extremo llamado **servidor,** que es quien provee los recursos o la ejecución de procesos.

Estos dos extremos de la comunicación pueden estar en la misma computadora o en los extremos opuestos del mundo, pero la clave aquí es comprender que son dos programas independientes, sea cual fuere la tecnología en la que estén implementados ser dos programas implica que no comparten memoria principal y por lo tanto las peticiones deben ser enviadas a través de la red.



Si se estudian los conceptos asociados al modelo cliente-servidor desde la arquitectura de software aparece gran cantidad de opciones distintas como arquitecturas de dos capas, tres capas, **n** capas, de workflow u orientada a servicios, sin embargo, en la mayoría de los casos estas se diferencian entre sí por la forma en la que se implementa el servidor, pero todas concuerdan en la relación existente entre el cliente y el servidor. También vale la pena decir, que los posibles clientes tienen hoy distintos esquemas de desarrollo y gran cantidad de tecnologías posibles. En fin, lo que nos importa a modo de introducir el tema es que hay un cliente y un servidor y entre ellos existirán peticiones a través de una red.

Insistimos en que, sobre todo mientras se trabaja en el desarrollo de los componentes, normalmente se da que ambos integrantes del modelo residen en la misma computadora, pero, de todos modos la comunicación se da por TCP/IP ya que por más que estén en la misma computadora son programas independientes. En esta ficha de estudio luego de comentar los elementos fundamentales de la comunicación nos centraremos en la programación del servidor.

Componentes:

- **Servidor**: Programa que se transforma en proveedor de recursos o procesos que pueden ser consumidos de manera remota.
- **Cliente**: Programa que consume los recursos o procesos del servidor, e interactúa con el usuario o con agentes externos al sistema.
- **Red**: Infraestructura de conexión física o inalámbrica a la que están conectados los servidores y los clientes y a través de la que se envían las peticiones y las respuestas.
- **Protocolo**: Especificación del formato de los mensajes dentro de los cuales se envían las peticiones y las respuestas a través de la Red.

Acerca de los protocolos cabe agregar que definen un conjunto de operaciones a implementar para enviar y recibir peticiones, además de cuestiones de bajo nivel como la estructura interna de los paquetes que serán enviados a través de la red. El modelo cliente servidor puede ser implementado sobre una gran cantidad de protocolos distintos como por ejemplo SMTP por Simple Mail Transfer Protocol, que se utiliza para enviar correos electrónicos, o POP por Post Office Protocol, que actualmente en su versión 3 se utiliza para descargar correos electrónicos, FTP o SFTP, por File Transfer Protocol que se utilizan para acceder a unidades de almacenamiento, son esencialmente el mismo protocolo, la 'S' de la segunda versión hace referencia a Seguro o Encriptado. Y finalmente uno de los más utilizados a nivel global, ya que es el que da soporte a la web es el protocolo HTTP por Hypertext Transfer Protocol, el cual también tiene su versión segura llamada HTTPS.

2.] Hypertext Transfer Protocol (http).

Es un protocolo de texto que actualmente se encuentra en la versión 2.0, aunque la versión más ampliamente distribuida o implementada es la versión 1.2 lanzada en febrero de 2000, el hecho de ser un protocolo de texto implica que lo que se podrá enviar dentro de los paquetes que este protocolo define es un bloque de texto, de hecho en las primeras versiones para enviar una imagen entre el servidor y el cliente había que tomar los bytes de la imagen y codificarlos con un esquema llamado base64 string donde cada 3 bytes se generan 4 caracteres ASCII lo que agregaba una sobrecarga de peso a la trasmisión. Hoy gracias al soporte del concepto de encoding de contenido, permite enviar información binaria sin la necesidad de conversión a base64 string evitando así la sobrecarga en el tamaño de la información a transferir.

Es un protocolo sin estado, es decir no guarda información sobre las conexiones anteriores por lo que de ser necesaria esta información queda a cargo tanto del servidor como del cliente implementar los mecanismos necesarios para almacenar datos entre una petición y otra, revisar estas técnicas será el objetivo final de esta ficha.

Es un protocolo apoyado en TCP y orientado a conexión, en la versión actual existe un concepto llamado pipelines que permite encadenar varias peticiones y respuestas sobre la misma conexión lo que acelera la comunicación.

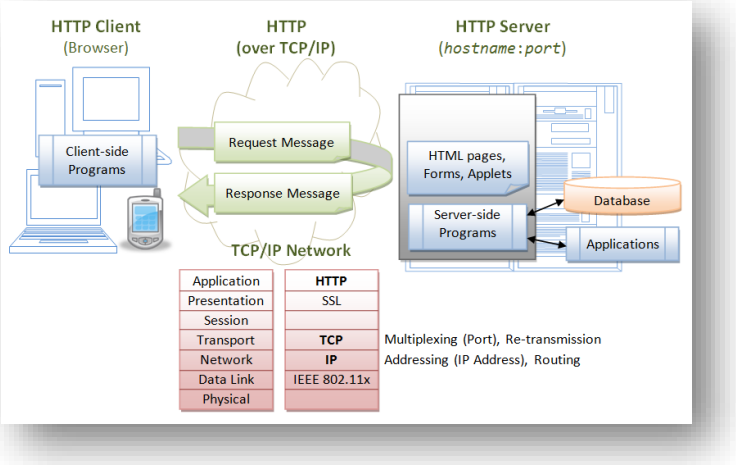
Define un conjunto de métodos de petición (algunas veces referidos como 'verbos') que pueden utilizarse para administrar la comunicación entre el cliente y el servidor, el protocolo es flexible en la posibilidad de agregar nuevos métodos de petición. Todos ellos están asociados a un recurso identificado de manera univoca a través de la URL del recurso, Localizador del Recurso Uniforme por su sigla en inglés.

A continuación, una lista de los métodos tradicionales:

- **GET**: solicita una representación de un recurso identificado, según http las peticiones que utilizan GET no deben tener ningún otro efecto más allá de obtener un recurso.
- **POST**: envía datos para que sean procesados por el recurso identificado, esto puede resultar en la creación de nuevos recursos o la actualización de recursos existentes.

- **HEAD**: Pide una respuesta idéntica a la que correspondería a una petición GET, pero en la respuesta no se devuelve el cuerpo. Esto es útil para poder recuperar los metadatos de los encabezados de respuesta sin tener que transportar todo el contenido.
- **PUT**: sube, carga o realiza un upload de un recurso especificado.
- **DELETE**: borrar el recurso especificado.
- Entre otros cabe mencionar: TRACE, OPTIONS, CONNECT, PATCH…

Además, define códigos de respuesta y las estructuras específicas de mensajes. Se puede estudiar mucho más acerca de los paquetes HTTP, pero por ahora nos alcanza con lo revisado hasta aquí y nos da pie para revisar entonces el mecanismo de comunicación entre el cliente y el servidor:



Ejemplo de diálogo HTTP: para obtener un recurso con el URL http://www.example.com/index.html

1. Se abre una conexión en el puerto 80 del host www.example.com. El puerto 80 es el puerto por defecto para HTTP, si se quisiera utilizar un puerto #### habría que codificar la URL de la forma http://www.example.com:####
2. Se envía un mensaje de la siguiente forma

```
GET /index.html HTTP/1.1
Host: www.example.com
Referer: www.google.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:45.0) Gecko/20100101 Firefox/45.0
Connection: keep-alive
[Línea en blanco]
```

3. La respuesta del servidor está formada por encabezados seguidos del recurso solicitado

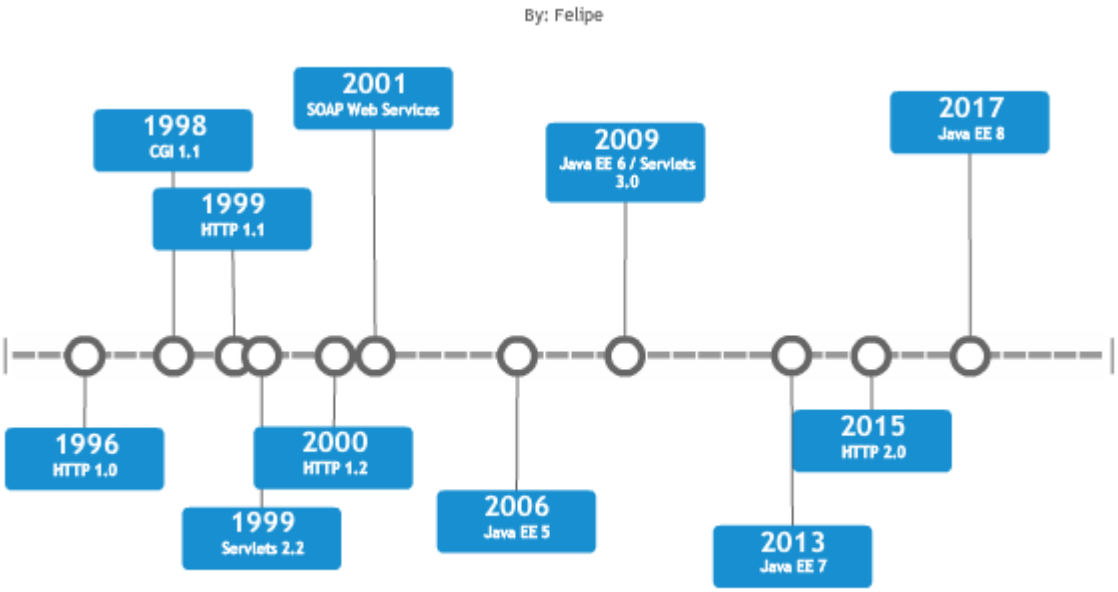
```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 2003 23:59:59 GMT
Content-Type: text/html
Content-Length: 1221

<html lang="eo">
<head>
<meta charset="utf-8">
<title>Título del sitio</title>
</head>
<body>
<h1>Página principal de tuHost</h1>
(Contenido)
  .
  .
  .
</body>
</html>
```

3.] Programación del Servidor, una breve historia.

Visto lo anterior nos vamos a centrar en la programación del servidor, es decir qué programamos, cómo lo programamos y cuáles son las herramientas necesarias para recibir la petición del cliente, escribir un programa que la procese y enviar una respuesta a la petición recibida.

A lo largo de la historia ha habido varias tecnologías distintas para dar respuesta a esta necesidad, a continuación, una brevísima referencia a los principales hitos de esa historia:



Luego de la especificación de HTTP como comentamos, de la que aparece la primera versión allá por 1996 y la web comienza a tomar forma y a estandarizarse, a finales 1997 Ken Cora lidera un equipo para tomar una especificación previa de CGI que era la que había dado forma a los servidores web originales y formarlizarla para dar así vida al primer modelo de programación del lado del servidor.

Los CGI, por su sigla en inglés para Common Gateway Interface, son pequeños programas escritos originalmente en lenguaje C o Perl que tenían la capacidad de recibir una petición que venía de internet, procesar los datos de la petición y generar un buffer con la respuesta que al finalizar el programa era nuevamente enviada al cliente que había realizado la petición. Quiero subrayar que eran programas, con lo que para cada petición ejecutan un proceso del sistema operativo independiente con los inconvenientes que eso trae aparejado.

Más o menos al mismo tiempo aparecían las primeras versiones de lo que inicialmente se llamó Java Profesional Edition y que luego pasó a llamarse J2EE por Java 2 Enterprise Edition. En 1999 aparece la versión 1.3 de J2EE con Servlets 2.2, que puede estudiarse como la primera versión ampliamente aceptada de Servlets como alternativa a los CGI, ambas tecnologías compartieron espacios durante mucho tiempo por cuestiones que tenían que ver con seguridad y eficiencia en lo que los CGI eran muy fuertes, sin embargo, los servlets de Java venían asociados a un concepto mayor, que era el concepto de Servidor de Aplicaciones en el que, en oposición al concepto de programas independientes de los CGI, delegaban en un programa contenedor las cuestiones como administración de la red, de las conexiones, de la memoria y luego cada vez más servicios como seguridad, transacciones y más aún que revisaremos oportunamente.

Posterior a la aparición de los servlets, y acompañando la evolución de HTTP, siguieron la aparición de los Web Services primero basados en XML a los que se llamó SOAP Web Services por la utilización del protocolo SOAP (Simple Object Access Protocol) para empaquetar los datos de la comunicación en texto xml, y luego aparecen los REST Web Services con un concepto basado en los verbos de HTTP y fuertemente apuntados a evitar el XML y a orientarse a otros mecanismos de estructurar el texto de las respuestas, como JSON.

Pero todo esto es harina de otro costal que deberemos revisar más adelante. Por ahora nos vamos a centrar en los Servlets, que, si bien no son el objeto de estudio al que apuntamos para implementación del Trabajo Práctico Único, nos permiten comprender la diferencia entre programar aplicaciones Java SE o de escritorio, en donde tenemos el control de la aplicación desde

que inicia la misma con el método main y hasta que termina; a desarrollar componentes del servidor, como por ejemplo los servlets, donde no tenemos el control del main, sino que tendremos que programar una clase o conjunto de clases que den forma al componente, escribir la configuración de despliegue y mapeo de ese componente para avisarle al servidor como desplegarlas e interactuar con él y luego empaquetar todo esto en un formato que el servidor de aplicaciones sea capaz de instalar y hacer funcionar.

4.] El servidor de aplicaciones.

Como hemos venido expresando, un servidor de aplicaciones es un producto de software (un programa más a instalar en sistema operativo del servidor) que tiene por función ser el contenedor y ejecutor de los componentes que programamos, por ende, debemos comenzar por comprender el funcionamiento de este software que va a ser a nuestros componentes algo así como lo que la Máquina Virtual Java era a nuestros programas Java SE en TSB.

Por nombrar algunos servidores Java EE, o Jakarta EE como se llama a la última versión de la comunidad de software libre, podemos mencionar entre los libres o gratuitos a:

- **Glassfish**: implementación de referencia de cada nueva versión de la plataforma
- **Payara**: basado en el código original de cada nueva versión de Glassfish y mantenido y soportado por Payara Foundation (con aportes de Oracle) quien dice encargarse de que Payara Server sea siempre un servidor Open Source
- **Wildfly**: basado en JBoss de Red Hat, también libre y soportado por la comunidad, a diferencia de Payara es un servidor ampliamente instalado a nivel global puesto que heredó las implementaciones que existían originalmente en JBoss, el servidor Java EE gratuito más difundido.

Por otro lado, tenemos algunos ejemplos de servidores de pago:

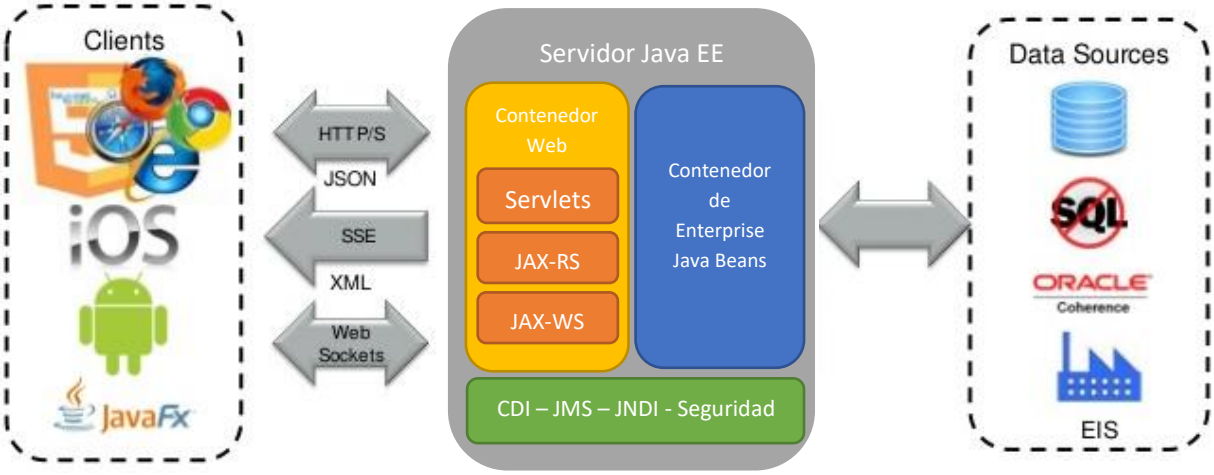
- **Weblogic** de Oracle
- **Websphere** de IBM
- O el propio **JBoss Enterprise Platform** de Red Hat

Para el lector que haya tenido algún contacto con estos productos podría estar faltando uno también ampliamente distribuido que es Tomcat, no incluimos Tomcat en las listas anteriores ya que Tomcat no es un servidor Java EE sino una parte de estos, específicamente enfocada en servir aplicaciones web, o lo que es igual, servir los componentes de aplicaciones que dan respuesta a peticiones http. Para nuestra propuesta en DLC nos bastaría. Sin embargo, no es correcto compararlo o agruparlo con servidores Java EE que son más que simples contenedores web y esto nos da el lugar a plantear los componentes de un servidor Java EE y la diferencia con un servidor exclusivamente web como el propio Tomcat.

Un servidor Java EE está compuesto por ***contenedores***, cada contenedor es capaz de desplegar y hostear (actuar de un tipo o conjunto de tipos distinto de componentes, los contenedores fundamentales de un servidor Java EE son el ***Enterprise Java Bean Container*** y el ***Web Container***, además el servidor cuenta con una cola de mensajes y un servicio de directorio que nos permite acceder a recursos del servidor y herramientas de administración de la seguridad entre otros servicios.

Vamos entonces comenzando a definir un poco más nuestra inicial visión trivial del modelo cliente servidor, así podemos poner algunas tecnologías a nuestro servicio (ver gráfico de página siguiente). Si bien es adecuado que ya queden mencionadas algunas tecnologías del cliente, como las herramientas para construir las SPA (por Single Page Application) que ejecutan en los navegadores, o aplicaciones para dispositivos móviles, o las propias aplicaciones Java FX que programamos en TSB y decir que todas ellas pueden ser clientes de nuestro servidor. O también revisar que nuestro

código de servidor puede conectarse bases de datos, relacionales o no, o a otros sistemas lo que en el gráfico figura como EIS (por Enterprise Information System). Lo que en realidad nos ocupa es el Servidor Java EE propiamente dicho que es donde va a ejecutar nuestro código.



Cabe aclarar que este servidor es, él mismo, una aplicación Java y por lo tanto está ejecutando dentro de una Máquina Virtual Java que será su interfaz con el sistema operativo donde esté hosteado. Este primer paso ya nos define que los servidores de aplicaciones Java EE son multiplataforma y pueden ejecutar en cualquier sistema operativo para el cual exista una Máquina Virtual Java y también que dependen de todo lo que ya hemos aprendido en TSB de Java SE. El servidor está dividido en contenedores, estos contenedores son los encargados de motorizar nuestros componentes, de administrar su ciclo de vida. Cada contenedor está preparado para desplegar un tipo de aplicación o componentes distinto y la forma de empaquetar estos componentes será distinta dependiendo de donde lo quiera desplegar.

Así tendremos entonces:

- ***Contenedor Web***: base del servidor web encargado de montar y ejecutar componentes de las aplicaciones web, entre otras APIs, es el encargado de implementar las APIs de Servlets y configurar y hostear las implementaciones de JAX-RS y JAX-WS.
  Los archivos que se despliegan en el contenedor web son los archivos .war y la configuración de dicho despliegue queda guardada en un archivo dentro del archivo .war, llamado web.xml aunque dicha configuración puede requerir otros archivos más.
- ***Contenedor EJB***: parte del servidor Java EE encargado de montar y ejecutar los Enterprise Java Beans, brinda a estos componentes servicios de transacciones, interceptores y seguridad entre otros.
  Los archivos que se despliegan dentro de este contenedor son archivos .jar. No vamos a utilizar este contenedor en DLC.
- ***Otros componentes***: también veremos normalmente implementación de la API JMS (cola de mensajes) y la API CDI (por Context and Dependency Injection) en los servidores Java EE entre otras tantas. Además los servidores Java EE también pueden desplegar archivos que contienen tanto desplegables de aplicaciones web, como paquetes de EJBs, estos archivos tienen la extensión: .ear y contienen al archivo o archivos .war y al archivo o archivos .jar en conjunto con la configuración de despliegue de ambos.

5.] Servidor de Aplicaciones vs Servidor de Recursos.

Por último y antes de adentrarnos en la programación de componentes del servidor, haremos una diferenciación que suele provocar confusión. Muchas veces se confunde el concepto de servidor de

recursos con el de servidor de aplicaciones, es decir se mezcla el concepto del Apache Server multiplataforma o Internet Information Server de Microsoft, que son servidores de recursos, con los servidores de aplicaciones como Payara o incluso Tomcat que si bien ya dijimos que no es un servidor Java EE completo, si es un Contenedor Web y por lo tanto si es capaz de servir componentes del contenedor web.

Entonces, ¿por qué decimis que Apache o IIS no son servidores de aplicaciones? o ¿en que se diferencian estos servidores de recursos de un servidor de aplicaciones?, pues bien los servidores de recursos, si son capaces de responder peticiones HTTP, pero responden a esas peticiones con recursos, es decir con archivos completos, como pueden ser imágenes, archivos de estilo (.css), archivos de scripts (.js) o incluso archivos html (.html) pero no son capaces de ejecutar un proceso para dar respuesta a una petición, en el caso que hiciera falta esto, derivan la petición a un servidor de aplicaciones.

Es decir, que son capaces de mapear una parte de la URL la que está estrictamente después de la identificación del servidor y el puerto a un directorio base en el sistema de archivos donde ejecuta el servidor, así entonces ante la petición:

http://www.example.com:8080/imagenes/logo.png

El servidor de recursos toma el sector que está luego de la barra posterior al puerto y genera una respuesta con el archivo ubicado en el sistema de archivos local, sumando a la raíz de la aplicación la ruta recibida en la petición...

/<path de mi aplicación>/imagenes/logo.png

O en el caso de Windows:

x:\<path de mi aplicación>\imagenes\logo.png

En este proceso no hay ejecución de código alguno, en cambio el servidor de aplicaciones recibe una petición con un recurso asociado que incluso puede tener datos enviados cómo parámetro para el proceso asociado…

http://www.example.com:8080/alumnos?filtroApellidos=Perez

Y va a ejecutar el componente que esté mapeado con el path “alumnos” pasando un parámetro con nombre “filtroApellidos” y valor “Perez”, que realizará las tareas necesarias como conectarse a una base de datos, recuperar los datos y generar una respuesta con la lista de alumnos de apellido “Perez” y devolverá esa lista de acuerdo con el formato esperado.

6.] Los Servlets.

Los Servlets son componentes del contenedor web dentro de la estructura Java EE, esencialmente son clases Java que por medio de la herencia y la implementación de interfaces extienden las capacidades del servidor agregando la posibilidad de servir nuevas peticiones con procesos generados por él desarrollador.

En el presente apartado proponemos revisar todos los elementos fundamentales necesarios para realizar lo que llamaríamos “Hola Mundo HTTP” con Servlets en Java, y recordando la idea de los “hola mundo“, intentaremos conocer todos los elementos necesarios para llevar ese servlet desde su creación/codificación hasta su prueba desde un cliente que en este caso será un navegador web.

Para programar un Servlet entonces, tendremos que:

1. Instalar un servidor Java EE, para estos ejemplos de DLC vamos a utilizar Payara Server, a continuación queda un link a un video que explica como instalar e iniciar Payara Server en su computadora (vean que hacia el final del video luego de iniciar la consola de administración el presentador crea un par de grupos de despliegue y agrega instancias a los

grupos, no es necesario esto para desarrollar): https://blog.payara.fish/getting-started-with-payara-server-5

2. Además, vamos a necesitar trabajar con un IDE para programación de Proyectos Java EE, en este caso elegimos Apache NetBeans, que es gratuito y tiene una integración automática con Payara server. Una vez instalado NetBeans con el instalador para Windows, o para Linux (recomiendo la instalación mediante el repositorio snap), tendremos que configurar y dejar lista la integración con Payara Server. A continuación, un link para crear un primer proyecto y configurar Payara Server como el servidor de despliegue para la prueba de ese proyecto: https://blog.payara.fish/adding-payara-server-to-netbeans
3. Con el proyecto de Prueba Web creado, a partir del ejemplo de proyecto web maven para NetBeans, crear nuestro primer Servlet.
4. Configurar el despliegue de dicho servlet.
5. Desplegar y probar dicho servlet observando los elementos que van a ir surgiendo mientras tanto.

El primer Servlet

Para crear un servlet los entornos como NetBeans tienen un wizard o ventana guía, que justamente nos guía a través de una plantilla que solicita todos los datos necesarios y crea y modifica todos los archivos que intervienen para que solo nos quede programar la respuesta a las peticiones, sin embargo, aquí vamos a realizar este primer servlet de forma básica paso por paso y ubicando y realizando todas las modificaciones de forma manual. Dijimos que un servlet es un componente, y lo es en su conjunto, pero ¿qué programamos?, ¿dónde escribimos el código del componente?, en java el código se escribe en clases ¿?...

Codificando nuestro primer Servlet

Lo que vamos a escribir entonces es una clase Java, y el camino para conectarnos con el servidor será por medio de la herencia que es la forma más natural de java para extender funcionalidad. Vamos a programar entonces una clase java que herede de javax.servlet.http.HttpServlet, que a su vez hereda de javax.servlet.GenericServlet. Ambas son clases abstractas, la clase GenericServlet es una clase abstracta que además implementa la interfaz Servlet que es la que define los métodos necesarios para que el servidor pueda interactuar con nuestro servlet, así GenericServlet implementa los métodos básicos para un servlet genérico independiente del protocolo, como por ejemplo el método service que responde a una petición genérica y recibe un ServletRequest y un ServletResponse y nos brinda el espacio para programar allí nuestro código para la respuesta que sea requerida.

Y la clase HttpServlet implementa los métodos asociados al protocolo HTTP, que es el que nos interesa en este momento. Esta clase entonces presenta los métodos doXxxx para cada uno de los métodos de petición de HTTP, así tendremos doGet, doPost, doPut, doDelete, etc.

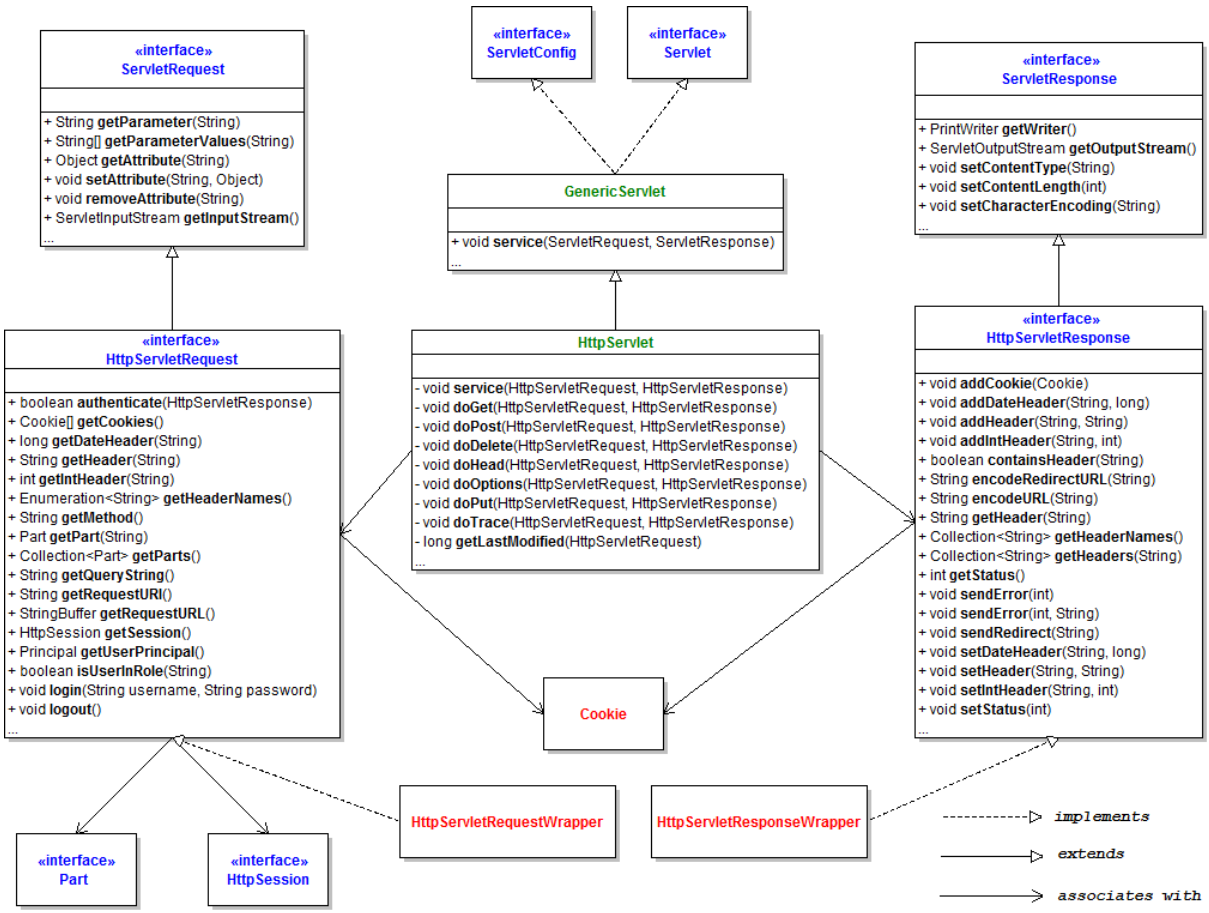
En el diagrama de clases de la página siguiente podemos revisar las relaciones entre los principales componentes de la API como también tener una referencia para los accesos a las distintas herramientas que utilizaremos más adelante.

En nuestro primer servlet solo vamos a redefinir el método doGet que es el que nos permite implementar la respuesta a una petición por método GET a nuestro servlet, sin embargo, esta implementación nos sirve de ejemplo ya que todos los métodos doXxxx comparte la misma estructura de firma.

El método doGet (y todos los métodos doXxxx) es de tipo void y recibe dos parámetros, uno de tipo javax.servlet.http.HttpServletRequest que llamaremos simplemente **request** y el otro de tipo javax.servlet.http.HttpServletResponse que llamaremos simplemente **response**. Ambas, interfaces

de la especificación de Servlets, que son implementadas por clases propias del servidor en el que estemos ejecutando.

Diagrama de clases general de la API de Servlets.



A través del request nuestro método tiene acceso a toda la información que llega con la petición, es decir, los parámetros que envió el cliente, el encabezado y la información de quién realizó la petición, etc. Y a través de response nuestro método tendrá acceso a los objetos necesarios para generar la respuesta que será enviada al cliente.

Así pues, un simple servlet, llamado “echo” que reciba un parámetro llamado “parametro” y devuelva un texto html con el eco (la cadena repetida dos veces y separada por espacio) sobre el texto recibido en el parámetro quedaría como se en la captura de código fuente de la página siguiente.

Muchas observaciones derivan de este primer bloque de código que no es un programa Java, sino un componente, en rigor de la verdad es la materia prima del componente, porque sin la información de despliegue no es más que una clase Java común y corriente, pero eso viene luego.

Ahora observemos algunos elementos, no hay mucho para decir de los imports o la firma de la clase que no hayamos dicho ya. Una clase pública que hereda de HttpServlet.

Sin embargo, cuando llegamos a la firma del método, que podemos ver en la línea 57, encontramos que aparecen dos excepciones chequeadas que vienen impuestas por el mismo método de la clase base y por lo tanto las excepciones de tipo java.io.IOException y javax.servlet.ServletException quedan informadas en el método mediante la cláusula ***throws***.

Luego, en el cuerpo del método lo primero que encontramos es la utilización del ***response*** para tomar contacto con el buffer de salida y, primero, en la línea 59 establecer el tipo de contenido, en este ejemplo texto/html (también se está definiendo el encoding de la respuesta) y luego, en la línea

60, tomar una referencia a un objeto de la clase java.io.PrintWriter, que es esencialmente un OutputStream de Texto, a la que llamamos ***out***, que nos permitirá escribir la respuesta que será enviada al cliente.

```
11  import javax.servlet.http.HttpServlet;
12  import javax.servlet.http.HttpServletRequest;
13  import javax.servlet.http.HttpServletResponse;
14
15  /**...4 lines */
19  public class EchoServlet extends HttpServlet {
20
21      /** Processes requests for both HTTP <code>GET</code> and <code>POST</code> ...9 lines */
30      protected void processRequest(HttpServletRequest request, HttpServletResponse response)
31              throws ServletException, IOException {...15 lines }
46
47      // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click on the + sign on the left
48      /** Handles the HTTP <code>GET</code> method ...8 lines */
56      @Override
        protected void doGet(HttpServletRequest request, HttpServletResponse response)
58              throws ServletException, IOException {
59          response.setContentType("text/html;charset=UTF-8");
60          try (PrintWriter out = response.getWriter()) {
61              String valorParametro = request.getParameter("parametro");
62              out.println("<!DOCTYPE html>");
63              out.println("<html>");
64              out.println("<head>");
65              out.println("<title>Servlet EchoServlet</title>");
66              out.println("</head>");
67              out.println("<body>");
68              out.println("<h3>Servlet EchoServlet at " + request.getContextPath() + "</h3>");
69              if (valorParametro != null && !valorParametro.isEmpty())
70                  out.println("<h1>" + valorParametro + " " + valorParametro + "</h1>");
71              else
72                  out.println("<h1>sin parámetro</h1>");
73              out.println("</body>");
74              out.println("</html>");
75          }
76      }
77
78      /** Handles the HTTP <code>POST</code> method ...8 lines */
86      @Override
        protected void doPost(HttpServletRequest request, HttpServletResponse response)
88              throws ServletException, IOException {...3 lines }
91
92      /** Returns a short description of the servlet ...5 lines */
97      @Override
        public String getServletInfo() {
99          return "Short description";
100     }// </editor-fold>
101
102 }
```

Luego, podemos observar también que, en la línea 61, utilizamos el ***request*** para tener acceso a los parámetros que recibimos del cliente y, mediante la invocación de request.getParameter("parametro"); tomar, de un diccionario en el que se parsearon los nombre=valor de la petición, el valor del parámetro con nombre "parametro" y asignarlo en la variable valorParametro para su posterior utilización en la generación de la respuesta. Aclaro aquí que queda a cargo del estudiante revisar los distintos tipos de parámetros que pueden llegar en una petición HTTP y el esquema de funcionamiento en cada caso.

Finalmente, en este primer servlet estamos generando un bloque te texto html que escribe en el buffer de salida mediante el objeto out, cabe aclarar aquí que actualmente los Servlets no tienen por propósito generar texto html como si lo hacían en sus comienzos, sino recibir datos del cliente, procesar esos datos y canalizar la petición a quien construya la respuesta o bien construir la respuesta, pero generando una estructura de datos en texto plano como por ejemplo JSON.

En el proyecto de ejemplo que acompaña la ficha, junto con el Servlet Echo que acabamos de mostrar también programamos un servlet EchoJson que ejemplifica este aspecto.

Configurando nuestro primer Servlet

Pues bien, ya hemos programado la clase que implementa el Servlet, la que por cierto, en el ejemplo que acompaña la ficha quedó en el paquete dlc.holaworldwideweb.servlets. La clase está lista compila y no produce errores, pero, ¿es un componente ya?, la respuesta es NO, para actuar como un componente además de construir el proyecto en el que la clase está contenida en el archivo .war correspondiente y desplegarlo en el servidor, hace falta avisarle al servidor que la clase EchoServlet es un Servlet y además avisarle cuál será la ruta a la que vamos a mapear nuestro servlet, con estas herramientas el servidor será capaz de comprender las peticiones a nuestro servlet y redireccionarlas a él trasladando los parámetros recibidos.

Es decir, cuando llegue la petición:

http://www.example.com:8080/echo?parametro=Texto

Ruta mapeada a nuestro Servlet

QueryString: cadena de parámetros con el formato <nombre>=<valor>[&<nombre>=<valor>]

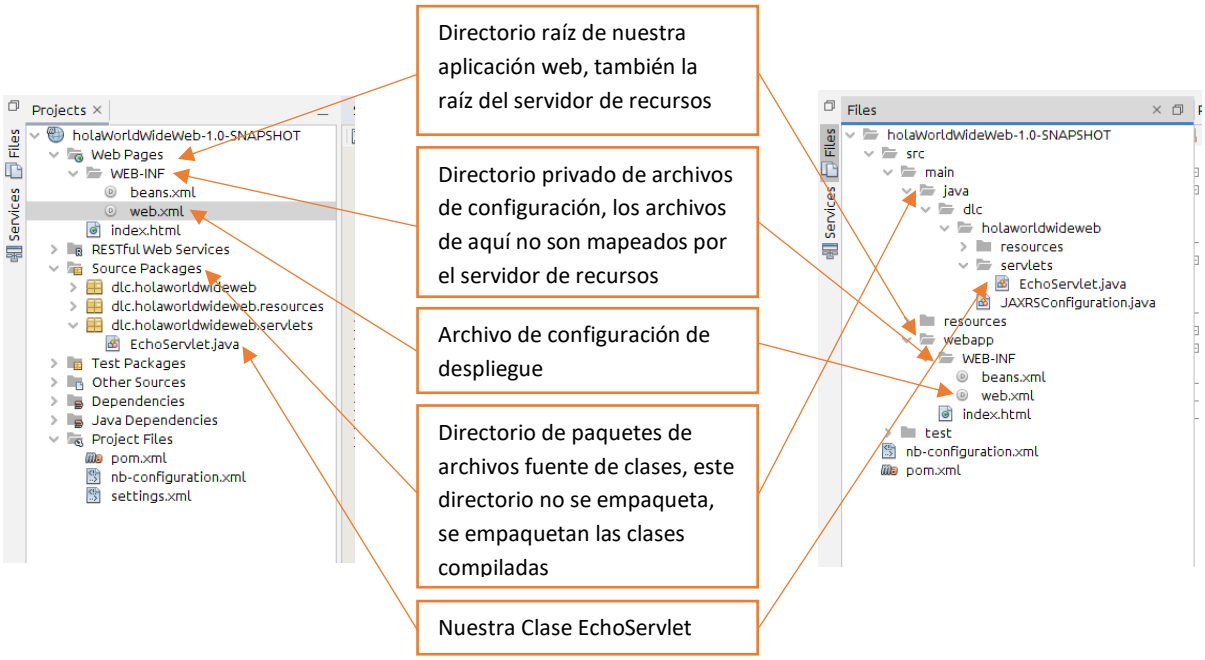
derive en la ejecución de nuestro método doGet(…).

Para esto vamos a usar el archivo de configuración de despliegue llamado web.xml, este archivo es el que lleva la información dentro del archivo empaquetado .war que se producirá al empaquetar nuestro proyecto para desplegarlo en el servidor, y le dice al servidor cómo desplegar y disponibilizar los componentes que allí residen.

Es decir que cuando el servidor comience el despliegue de nuestra aplicación web, va a buscar el archivo web.xml y se va a nutrir de la información que allí encuentre para interactuar con nuestra aplicación web.

Para ubicar dicho archivo estudiemos un poco la estructura de archivos de nuestro proyecto web, a la izquierda podemos observar la estructura que presenta NetBeans de nuestro proyecto y a la derecha podemos ver cómo se estructuran los archivos en los directorios del sistema operativo, sin embargo, en ambos casos podemos identificar los archivos que nos interesan:

Nota: vale aclarar que en el proyecto web que se crea a partir de maven en NetBeans se agregan elementos que en este momento no son necesarios, pero es una de las desventajas de trabajar con plantillas de proyecto, con las que tenemos que lidiar para obtener todo lo que sí nos sirve.



Habiendo ubicado entonces, el archivo web.xml veamos como podemos utilizar dicho archivo para realizar la configuración de nuestro Servlet:

```
3   <servlet>
4       <servlet-name>Echo</servlet-name>
5       <servlet-class>dlc.holaworldwideweb.servlets.EchoServlet</servlet-class>
6   </servlet>
7   <servlet-mapping>
8       <servlet-name>Echo</servlet-name>
9       <url-pattern>/echo</url-pattern>
10  </servlet-mapping>
```

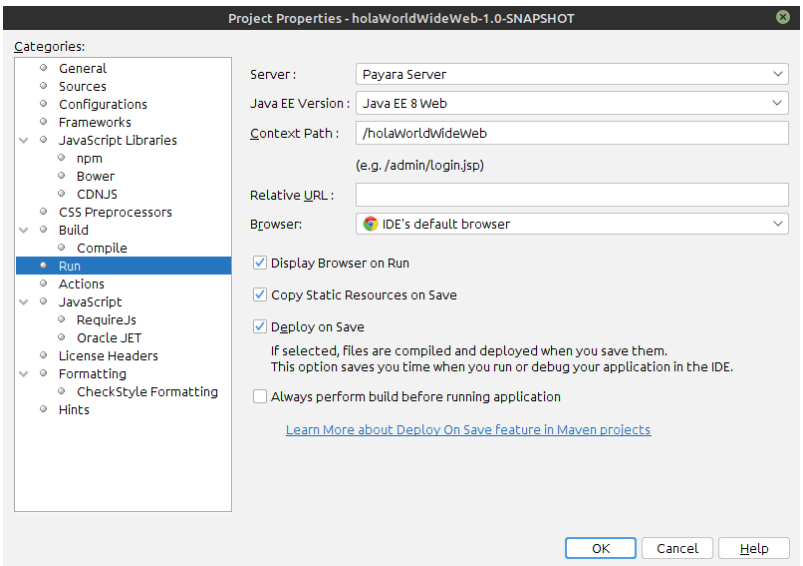
Para configurar un servlet son necesarios dos pasos. Por un lado, configurar el servlet como componente en el servidor y por otro lado definir el mapeo para que las peticiones lleguen a nuestro servlet.

Para configurar el servlet vamos a agregar un tag <servlet> en el que vamos a especificar, al menos, un nombre para el servlet, que debe ser único en toda la aplicación, mediante el tag <servlet-name> como se puede ver en la línea 4 y, la clase que implementa dicho servlet, como se ve en la línea 5, mediante el tag <servlet-class>.

Una vez configurado el servlet, ya nos podemos referir a él por su nombre y así lo hacemos al configurar el mapeo, para lo cual agregamos el tag <servlet-mapping>, donde asociamos un servlet por su nombre, mediante el tag <servlet-name>, como vemos en la línea 8, y el patrón de url para dicho servlet, mediante el tag <url-pattern>, como vemos en la línea 9 del ejemplo.

Para comprender la petición del ejemplo que vamos a ver luego, hace falta un elemento extra, que es el path del contexto de la aplicación, ese nombre va guardado en el archivo meta.inf del propio archivo .war.

Este path se configura en el momento de realizar el despliegue en el servidor, en nuestro caso, NetBeans utiliza el artifactid de nuestro proyecto para esta configuración y por eso el context path de nuestro ejemplo es: /holaWorldWideWeb, pero lo podemos ver en la pantalla de propiedades del proyecto en el apartado Run:



Entonces en base a todo lo expuesto podemos ir cerrando la definición de nuestra tarea, para invocar nuestro servlet deberemos indicar, el protocolo, el servidor, más el puerto si no es el puerto 80 (como es el caso por defecto de Payara que es 8080), el path de contexto de la aplicación y finalmente el patrón url de mapeo de mi servlet, para el ejemplo eso queda:

http://localhost:8080/holaWorldWideWeb/echo

Desde aquí podemos trazar cómo, el servidor, va recuperando la información para llegar a nuestra clase, pues luego de ubicar el servidor mediante su dirección ip o nombre (***localhost***) más él puerto

si hace falta (:***8080***) y la aplicación dentro de este mediante el contexto (/***holaWorldWideWeb***), obtiene el patrón url del recurso (nuestro servlet /***echo***), al buscarlo en el mapeo lo relaciona con un nombre único de componente (***Echo***), y al buscar el componente (nuestro Servlet) encuentra la clase que lo implementa (***dlc.holaworldwideweb.EchoServlet***).

Bonus, configurando un servlet más, por anotaciones

A partir de la versión 4 de la Api de Servlets se agrega una nueva posibilidad para la configuración de los Servlets y esta posibilidad no requiere la utilización del archivo de configuración de despliegue web.xml, sino que directamente nos permite configurar nuestro servlet mediante una anotación en la propia clase del servlet.

Por ejemplo:

```
17  import javax.json.Json;
18  import javax.json.JsonWriter;
19
20  /**...4 lines */
24  @WebServlet(name = "EchoJson", urlPatterns = {"/echojson"})
25  public class EchoServletJson extends HttpServlet {
26
27      /** Processes requests for both HTTP <code>GET</code> and <code>POST</code> ...9 lines */
36      protected void processRequest(HttpServletRequest request, HttpServletResponse response)
37              throws ServletException, IOException {...15 lines }
52
53      // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click on the + sign on the left to edit the code.">
54      /** Handles the HTTP <code>GET</code> method ...8 lines */
62      @Override
        protected void doGet(HttpServletRequest request, HttpServletResponse response)
64              throws ServletException, IOException {
65                  JsonObjectBuilder resultBuilder = Json.createObjectBuilder();
```

En este caso utilizamos la anotación @WebServlet (como vemos en la línea 24) para indicar que la clase es un Servlet y mediante el parámetro name de la anotación le damos un nombre de componente y mediante el parámetro urlPatterns de la anotación le configuramos el patrón url de mapeo para el servlet.

Notar que aquí el servidor tendrá el trabajo de inspeccionar cada clase durante el despliegue para generar la misma configuración que gracias al archivo web.xml y la trazabilidad luego ante una petición del recurso …/echojson será igual a la que antes estudiamos solo que basada en la configuración mediante la anotación. Vale decir también que en el caso que en el web.xml incluyéramos una definición distinta para esta clase, la configuración del web.xml pisa o redefine la configuración por anotación.

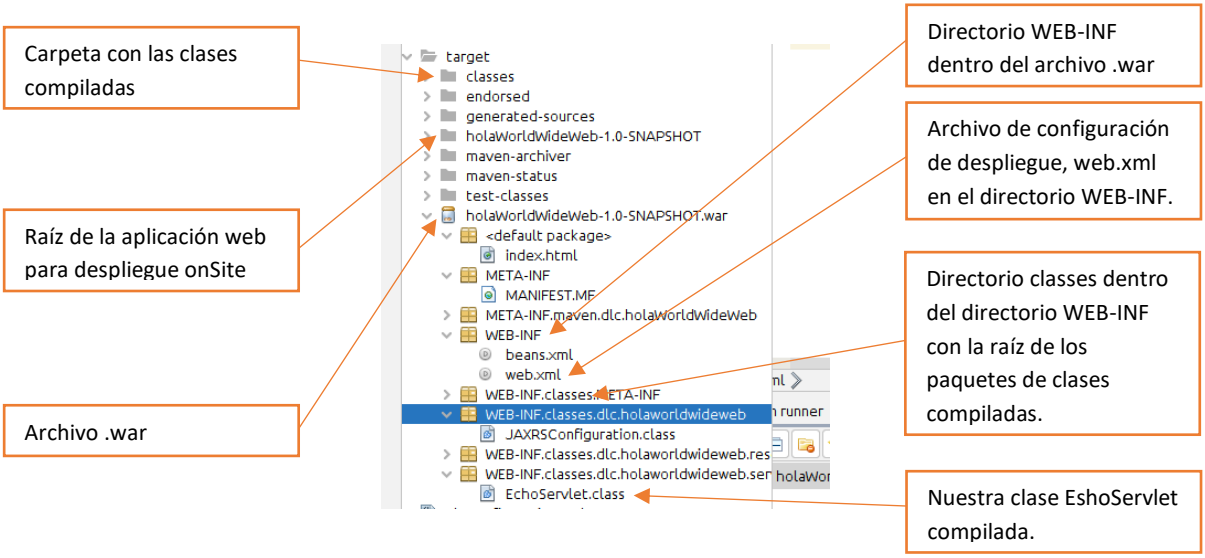
Desplegando y probando nuestro primer Servlet

Habiendo llegado hasta aquí solo nos falta empaquetar y probar nuestro primer servlet, para generar nuestro archivo .war debemos pedirle a maven que compile, testee (en el caso que hubiera tests unitarios) y empaquete, este último paso es el que genera el archivo .war.

En NetBeans podemos instalar el plugin MavenRunner para hacerlo a la usanza de maven o utilizar la opción “Clean and Build” (por limpiar y construir) del menú contextual del proyecto, que además del goal package de maven, ejecuta también el goal install de maven lo que nos despliega el proyecto en el servidor que estuviera configurad, pero no nos adelantemos tanto. Antes de esto revisemos un poco que hizo el empaquetador.

Si revisamos en la estructura de archivos del proyecto en el sistema de archivos, vamos a encontrar una nueva carpeta o directorio, target, que es la ubicación por defecto para los archivos generados o de salida.

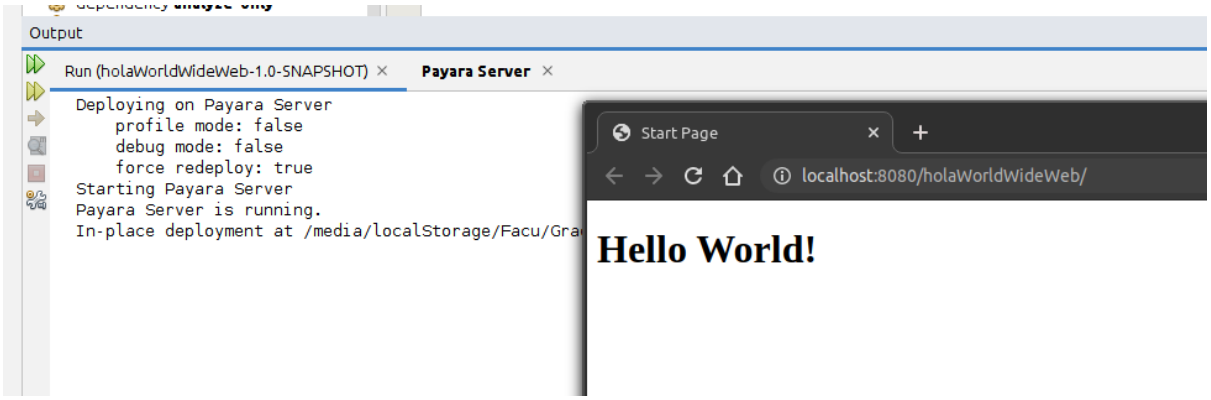
En este directorio vamos a encontrar varias cosas que sería bueno revisar en el siguiente esquema:



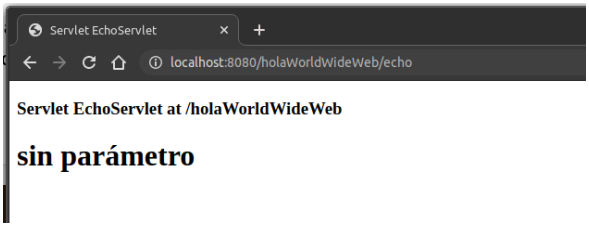
Notar que no están los archivos .java de nuestro proyecto, pero si los recursos como, por ejemplo, el archivo index.html que crea NetBeans con el proyecto por defecto cuando creamos el proyecto mismo.

Bien, hecho esto, despleguemos el proyecto en el servidor para poder probarlo, desde NetBeans es tan simple como hacer click en el triángulo verde, del mismo modo que si estuviéramos ejecutando una aplicación Java SE de consola.

NetBeans se encargará de iniciar nuestro servidor y realizar el despliegue de nuestra aplicación en el servidor e incluso iniciará un navegador con la petición por defecto para nuestra aplicación que hace referencia al mencionado archivo index.html.



En la ventana de resultados de NetBeans vemos dos pestañas, una con la consola del servidor identificada como “Payara Server” y otra con el resultado de la ejecución de nuestra aplicación. En el browser vemos lo que muestra el archivo index.html… sí y podemos modificarlo si lo queremos comprobar.

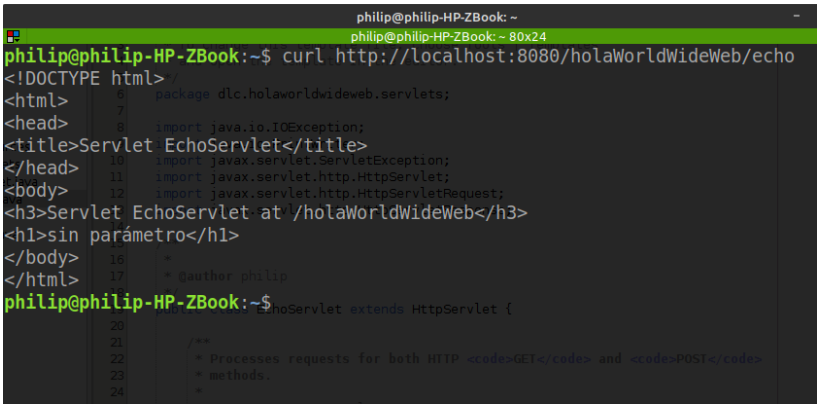


Ahora bien, si cambiamos la petición para invocar a nuestro servlet, primero sin ningún valor extra más allá del patrón de url que definimos, el resultado será el texto html que generamos con el Servlet para cuando no recibe valor en el parámetro.

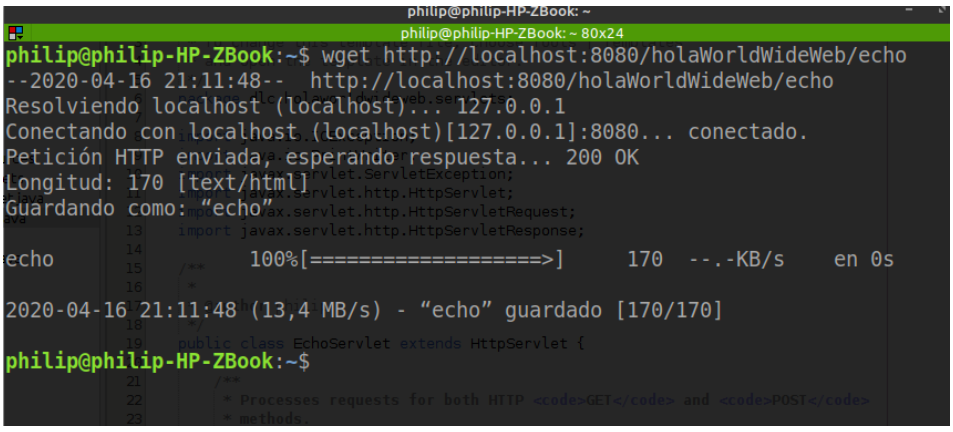
Si en cambio lo ejecutamos, enviando el patrón url y agregando el parámetro con el valor “Texto enviado”, el resultado será ahora el que podemos observar en la imagen.

Notar el “%20” que divide la palabre “Texto” de la palabra “enviado” en el valor del parámetro, es la forma de enmascarar el espacio para las peticiones HTTP.

Visto así como hasta ahora, pareciera que es obligatorio utilizar un navegador de internet y que por arte de magia en lugar de texto, como vinimos comentando, estamos recibiendo en el cliente, formatos y ventanas. Lo último que nos queda por comprar entonces, es que realmente podemos enviar una petición al cliente y recibir exclusivamente texto. Si usamos la herramienta curl de línea de comandos podemos obtener el mismo texto que recibe el navegador solo que lo veremos sin formateo ni la ventana del propio navegador:



Y finalmente si usamos wget en lugar de curl, la salida va a parar a un archivo sin embargo tenemos una breve vista previa de la comunicación con el servidor:



Bueno pues, hemos inicializado un entorno de trabajo para proyectos cliente servidor con Java EE, Payara Server y NetBeans, hemos codificado nuestro primer Servlet, hemos configurado su despliegue, hemos generado el empaquetado instalable en el servidor, lo hemos desplegado (o instalado que es igual) en el servidor y hemos probado nuestro Servlet.

Se puede decir que, logramos nuestro ***HOLA MUNDO WEB!.***

Ahora nos queda por delante comprender como controlar en los componentes lo que en las aplicaciones Java SE controlamos manejando nosotros mismos el main, que en los componentes lo haremos mediante el llamado ***Ciclo de Vida del Componente*** y finalmente revisar las posibilidades que tenemos para manejar el ***estado entre peticiones*** que como ya vimos, HTTP por si solo no es capaz de manejar.

7.] Ciclo de Vida de los Servlets.

Cuando programamos aplicaciones Java SE y tenemos el control del main de dichas aplicaciones, somos amos de la vida de nuestras aplicaciones, nosotros decidimos cuando inicializar un recurso, nosotros decidimos cuando crear instancias de los objetos con los que nos vamos a comunicar, nosotros decidimos cuando destruir o liberar para el garbage collector esas instancias y de una u otra manera nosotros manejamos cuándo finaliza la aplicación. Todo esto que parece tan elemental y básico se vuelve de cabeza cuando comenzamos a programar componentes del servidor, puesto que lo último que decidimos, el último control que tenemos directamente nosotros es cuando desplegamos la aplicación y en el mejor de los casos cuando iniciamos o detenemos el servidor, pero no podemos manejar cuando se va a instanciar nuestra clase, o cuantas instancias de ellas van a existir o cuando se va a disponer para el garbage collector.

Esto se vuelve un poco tenebroso ¿no?, al menos al principio para los que nos tocó hacer el cambio de venir muy acostumbrados al control, dejar ese control cuesta. Y en ese momento nos encontramos buscando puntos a los que anclarnos para poder migrar nuestros conceptos previos al nuevo esquema, bueno, en la programación de componentes, esos puntos de anclaje los encontramos al conocer, comprender y manejar el ciclo de vida de los componentes. Y los Servlets no son la excepción.

Naturaleza multihilos de los servidores

Antes de entrar de lleno en el ciclo de vida de los Servlets, hay un elemento que no podemos dejar de mencionar, y es la naturaleza multihilos de los servidores de aplicaciones. Sin entrar aquí en los detalles de la programación de redes y la necesidad de los hilos para que un mismo servidor escuchando en un solo puerto pueda dar respuesta a “n” clientes, vamos solamente a decir que cada cliente va a iniciar su petición en un hilo independiente en el servidor y por esta razón una de las cuestiones a tener en cuenta en los Servlets es la seguridad a la hora de responder peticiones multihilos sobre todo cuando hay acceso a memoria compartida por los “n” hilos que se ejecutan de manera concurrente en el servidor. Situación que, por cierto, es probablemente imposible de probar en desarrollo y muy difícil de probar en testing, pero que puede causar graves problemas cuando llevamos nuestra aplicación a un entorno productivo.

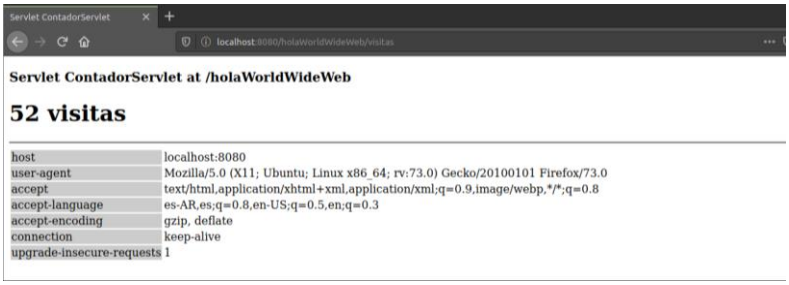
La API de Servlets logra manejar esta situación al ***crear una y solo una instancia*** de cada clase/Servlet en la aplicación y una vez creada esa instancia, da servicio, es decir ejecuta el método doXxxx que corresponda para el método de petición, con esa misma instancia, para todos los clientes conectados independientemente del hilo por el cual llegue la petición.

En el ejemplo que acompaña a la presente ficha agregamos un servlet que se llama Contador y que está implementado en la clase ContadorServlet, en este servlet agregamos un atributo entero de instancia en la clase, y, cada vez que el servlet responde a una petición incrementa el valor del atributo y luego retorna un bloque de texto html, mostrando el valor del atributo en conjunto con algunos datos de la petición.

Podemos ver entonces una petición desde Chrome:

En este servlet, además, se ejemplifica el ciclo de vida que veremos a continuación, y por esa razón como vamos a detallar en el siguiente apartado, la primera visita devuelve el contador 51.

E inmediatamente a continuación otra petición desde Firefox, que evidentemente tiene que haber iniciado otro hilo en el servidor, ya que los distintos navegadores ejecutan en procesos separados y por lo tanto conectaron de forma separada al servidor, el resultado que arroja es:



Queda demostrado pues, en el contador incrementado, que cada navegador inició un hilo independiente en el servidor (como veremos más adelante lo podemos observar en el nombre del navegador), pero ambos hilos fueron procesados por la misma instancia del Servlet ya que el atributo de instancia se incrementó y contó las visitas de hilos separados de los dos navegadores en el mismo contador, de ahí la secuencia.

Fases del ciclo de vida

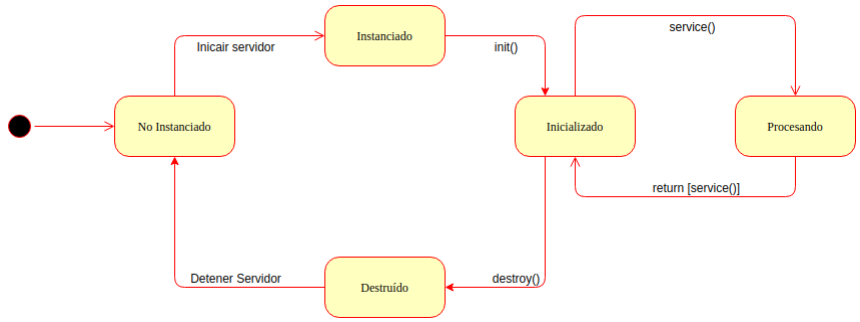
Ahora sí, entonces, podemos centrarnos en los mecanismos para intervenir en el control del componente en el servidor y en la investigación de lo que llamamos ciclo de vida del componente que no es más que un conjunto de métodos en la clase que lo implementa y en el conocer, en qué orden y momento, el servidor ejecuta cada uno de esos métodos. Con ese conocimiento en la mochila, solo restará programar en cada segmento/método el código necesario para que nuestro componente funcione como el proceso que implementa necesita que lo haga. Y de esa forma pues, volvemos a tener el control.

El ciclo de vida está compuesto por 3 o 4 fases de acuerdo con la documentación que consultemos, la razón de la diferencia está en que cuando se habla de 4 fases 2 de esas 4 fases se ejecutan de manera conjunta e indivisible. Aquí nos vamos a referir a 3 fases mencionando en la primera las 2 operaciones que se ejecutan.

Estas fases son:

- ***Inicialización***, que incluye la creación de la instancia y la inicialización propiamente dicha
- ***Servicio***, que implica la ejecución del método que corresponda para dar respuesta a la petición del cliente
- y finalmente ***destrucción***, que brinda un lugar a la programación del cierre de recursos que fueran necesarios.

Así podemos entenderlo como un diagrama de estados de la siguiente manera:



Notar que donde interviene el método service(), el mismo representa a cualquiera de los métodos doXxx() para el caso de los HttpServlets.

Entonces, cuando se inicial el servidor, el Class Loader carga la clase, pero aún no crea ninguna instancia, cuando llega la primera petición que el servidor necesita mapear a nuestro servlet se produce la fase de ***Inicialización***, primero se ejecuta el constructor en el caso que se hubiera definido, sino se definió la clase tendrá el constructor por defecto y será ejecutado este. En nuestro ejemplo lo creamos solo para realizar una impresión en el log del servidor y poder trazar luego su funcionamiento.

Una vez ejecutado el constructor, inmediatamente después, todavía en la fase de inicialización y por única vez se ejecuta el método init(). El método init presenta al menos dos versiones diferentes en la API de Servlets, una que no recibe parámetros y otra que recibe como parámetro una instancia de una clase que implementa la interfaz ServletConfig (revisar nuevamente el diagrama de clases) esta interfaz nos brinda acceso a los parámetros de configuración que podemos agregar para la inicialización del servlet en el archivo de configuración de despliegue o en la anotación. En nuestro ejemplo hemos agregado un parámetro de configuración para mostrar la inicialización del contador mediante un valor definido en la configuración del servlet.

```
10  import javax.servlet.ServletConfig;
11  import javax.servlet.ServletException;
12  import javax.servlet.http.HttpServlet;
13  import javax.servlet.http.HttpServletRequest;
14  import javax.servlet.http.HttpServletResponse;
15
16  /**...4 lines */
20  public class ContadorServlet extends HttpServlet
21  {
22      private int contador;
23
24      public ContadorServlet()
25      {
26          System.err.println("Ejecutando constructor().....................................");
27      }
28
29      @Override
31      public void init(ServletConfig config) throws ServletException
32      {
33          super.init(config);
34          contador = Integer.parseInt(config.getInitParameter("initValue"));
35          System.err.println("Ejecutando init().....................................");
36
37      }
38
39  //    @Override
40  //    public void init() throws ServletException
41  //    {
42  //        contador = 10;
43  //        System.err.println("Ejecutando init().....................................");
44  //    }
45
```

En el archivo de configuración será necesario configurar el parámetro “initValue” para que no falle al intentar recuperarlo.

```
7       <servlet>
8           <servlet-name>Contador</servlet-name>
9           <servlet-class>dlc.holaworldwideweb.servlets.ContadorServlet</servlet-class>
10          <init-param>
11              <param-name>initValue</param-name>
12              <param-value>50</param-value>
13          </init-param>
14      </servlet>
```

Ahora vamos comprendiendo por qué la primera ejecución en el ejemplo anterior arrojó 51 visitas 😉.

Bien, luego de haber inicializado nuestro Servlet no olvidemos que lo que disparó todo este proceso fue una primera petición de un cliente, y por lo tanto hay que dar servicio a esa petición, con lo que entramos en la fase de ***Servicio***, en este ejemplo hemos dejado la plantilla de Servlets que NetBeans nos crea por defecto, en la clase resultante NetBeans agrega un método protected processRequest(…) con idéntica firma a los métodos doXxx(…) y dentro de estos últimos, de los que también agrega en la plantilla el doGet(…) y el doPost(…) codifica la llamada al processRequest(…) que mencionamos antes, logrando así, que nuestro servlet responda de la misma forma independientemente del método de la petición.

Tenemos entonces, en nuestro Servlet el código del método processRequest y de los métodos doGet y doPost:

```
46
47      /** Processes requests for both HTTP ...10 lines */
57      protected void processRequest(HttpServletRequest request, HttpServletResponse response)
58              throws ServletException, IOException
59      {
60
61          System.err.print("Ejecutando service().....................................");
62          response.setContentType("text/html;charset=UTF-8");
63          contador++;
64          try (PrintWriter out = response.getWriter())
65          {
66              /* TODO output your page here. You may use following sample code. */
67              out.println("<!DOCTYPE html>");
68              out.println("<html>");
69              out.println("<head>");
70              out.println("<title>Servlet ContadorServlet</title>");
71              out.println("</head>");
72              out.println("<body>");
73              out.println("<h3>Servlet ContadorServlet at " + request.getContextPath() + "</h3>");
74              out.println("<h1>" + contador + " visitas</h1>");
75              out.println("<hr/>");
76              out.println("<table border=0>");
77              Enumeration<String> e = request.getHeaderNames();
78              while (e.hasMoreElements()) {
79                  String headerName = (String)e.nextElement();
80                  String headerValue = request.getHeader(headerName);
81                  out.println("<tr><td bgcolor=\"#CCCCCC\">");
82                  out.println(headerName);
83                  out.println("</td><td>");
84                  out.println(headerValue);
85                  out.println("</td></tr>");
86              }
87              out.println("</table>");
88              out.println("</body>");
89              out.println("</html>");
90          }
91      }
92
93      // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click on the + sign on the left to edit the code.">
94      /** Handles the HTTP ...9 lines */
103     @Override
105     protected void doGet(HttpServletRequest request, HttpServletResponse response)
106             throws ServletException, IOException
107     {
108         processRequest(request, response);
109     }
110
     /**
```

Notar el incremento del atributo contador en el código del método que lleva a cabo el proceso, tenemos entonces un atributo, declarado de instancia, inicializado en el método init con el valor obtenido del parámetro de configuración del servlet, e incrementado luego cada vez que un usuario vuelve a hacer una petición, y como respuesta desde clientes distintos la secuencia perfecta de incrementos. La suma de estos elementos aclara la demostración del apartado anterior y evidencia que la instancia de la clase ContadorServlet en memoria es siempre la misma para dar servicio a todas las peticiones que llegan.

También se evidencia que el objeto va a quedar en un ciclo entre brindar servicio y estar inicializado mientras el servidor siga funcionando.

Finalmente, cuando se decida detener el servidor, este invocará el método destroy() de nuestro servlet antes de descargar la instancia del objeto de memoria y finalizar todos los procesos de la máquina virtual cumpliendo así con la fase de ***Destrucción*** del servlet.

```
136
137     @Override
        public void destroy()
139     {
140         System.err.println("Ejecutando destroy().....................................");
141         super.destroy(); //To change body of generated methods, choose Tools | Templates.
142     }
143
```

Y de esta forma finaliza el ciclo de vida del servlet, llegando nuevamente al estado final, cabe mencionar que en situaciones particulares cuando un servlet quedara mucho tiempo sin recibir ninguna petición el servidor podrá descargarlo para liberar recursos cerrando así el ciclo de vida de forma independiente a la detención del propio servidor, igualmente, en este caso, también invocará el método destroy() antes de desreferenciar al objeto.

Veamos un ejemplo de como se visualizan las llamadas al System.err.println(…) que programamos en cada una de las fases desde la consola del servidor evidenciando aún más lo que venimos sosteniendo.

```
Initializing Soteria 1.1-b01 for context '/holaWorldWideWeb'|#]
Loading application [holaWorldWideWeb-1.0-SNAPSHOT] at [/holaWorldWideWeb]|#]
holaWorldWideWeb-1.0-SNAPSHOT was successfully deployed in 502 milliseconds.|#]
Ejecutando constructor()......................................|#]
Ejecutando init()......................................|#]
Ejecutando service()......................................|#]
Ejecutando service()......................................|#]
Ejecutando service()......................................|#]
Server shutdown initiated|#]
FileMonitoring shutdown|#]
JMXStartupService: Stopped JMXConnectorServer: null|#]
JMXStartupService and JMXConnectors have been shut down.|#]
Ejecutando destroy()......................................|#]
The web application [] created a ThreadLocal with key of type [java.lang.ThreadLocal] (value [java.lang.Thre
The web application [] created a ThreadLocal with key of type [java.lang.ThreadLocal] (value [java.lang.Thre
```

8.] Seguimiento de Estado a través de las distintas Peticiones del Cliente.

Como ya dijimos HTTP es un protocolo que no mantiene información acerca de las peticiones anteriores de un cliente, por lo tanto, en el caso de querer almacenar información de las peticiones de un cliente deberemos arbitrar los medios desde nuestro código para lograrlo.

En frente tenemos dos caminos para lograr el cometido, cada uno con sus ventajas y desventajas, por un lado, estará mantener el estado en la memoria del servidor, o lo que comúnmente se conoce como del lado del servidor, que podría implicar la memoria, el sistema de archivos o incluso una base de datos. O, en oposición, mantener el estado del lado del cliente.

Si elegimos mantener el estado del lado del servidor, tendremos como principal ventaja el hecho de tener el dominio del tiempo de vida del estado almacenado y la capacidad de disponer del mismo en cualquier momento y enfrentaremos el costo que implicará multiplicar la necesidad de almacenamiento asociado al estado por la cantidad de conexiones concurrentes al servidor lo que representa una merma en la posibilidad de escalabilidad de la aplicación al utilizar esta estrategia.

Por otro lado, si optamos por el camino de mantener el estado del lado del cliente no tendremos problemas de espacio ya que la necesidad de almacenamiento se divide entre todos los clientes y cada cliente solo almacena su propio estado, pero incrementamos el peso de las peticiones que tendrán que enviar y recibir dicho estado en cada petición y dejamos un canal de ataque en el cliente que puede ser utilizado de forma maliciosa.

Hecha la aclaración, le tocará al desarrollador decidir lo que convenga implementar en cada caso y acto seguido vamos a revisar las distintas alternativas para las opciones planteadas.

Mantención de Estado del Lado del Servidor

Si bien, esta alternativa permite ser implementada en un sin número de formas, nos interesa en este punto centrarnos en las posibilidades de la memoria del servidor y la visión y acceso que tenemos a esta desde el código de los Servlets. Esto nos brindará un panorama completo acerca de la

posibilidad de interacción entre el código que escribimos en un componente, los Servlets en este caso, y las capacidades administradas por el servidor.

A la hora de almacenar estado de las peticiones en la memoria en el servidor, debemos decidir cuánto tiempo queremos que dicho estado permanezca disponible y si queremos que ese estado sea compartido por todas las peticiones realizadas al servidor o que cada conexión mantenga su estado de forma independiente, lo que en cierta forma también tiene que ver con el tiempo de vida.

En este punto entonces tenemos, la posibilidad de mantener el estado a nivel de la petición, es decir desde que llega la petición y hasta que la respuesta es enviada al cliente. Esto no parece tan revolucionario ya que como estamos respondiendo a toda la petición del cliente en un solo método en nuestros ejemplos no hay redirección de la petición en el servidor y por lo tanto no existe la necesidad de compartir los datos de la petición entre un componente y otro. Sin embrago, esto si es necesario en escenarios reales muchas veces, por ejemplo, si tuviéramos un servlet que actúa de filtro o interceptor, que no vimos en esta ficha pero que existen y que luego de realizar las validaciones correspondientes le pasa la posta a quien realmente procesa la petición.

O podríamos necesitar mantener el estado a lo largo de toda la sesión de un cliente, es decir desde que el cliente realiza la primera petición de una conexión, por ejemplo, cuando se autentica con sus credenciales, a través de todas las peticiones que este mismo cliente realice y hasta que el cliente decida cerrar su sesión y abandonar el servidor. En este caso además deberemos asegurar que cada cliente mantenga un estado independiente del resto de los clientes conectados al servidor y que dichos estados no se mezclen entre llamada y llamada de cada uno de los clientes de forma independiente.

O incluso podríamos requerir mantener el estado entre todas las peticiones que se realicen al servidor independientemente de qué cliente las haya realizado, como por ejemplo para mantener valores de configuración general de la aplicación que requieren estar disponibles para todos los clientes para todas las peticiones, desde que inicie el servidor y llegue la primera petición y hasta que el servidor se detenga o un proceso decida eliminar dicho estado del servidor.

En resumen, podríamos requerir mantener los datos a nivel de una ***Petición***, de todas las Peticiones de una ***Sesión*** o de todas las peticiones mientras esté disponible la aplicación en este último caso llamo en la API de Servlets mientras esté vivo el ***Contexto***.

Analizaremos pues, las herramientas existentes en el servidor de aplicaciones, para dar soporte a cada una de estas tres alternativas.

Mantenimiento de estado a nivel de Petición

Como dijimos antes, no es fácil visualizar este caso en los ejemplos triviales que manejamos en esta ficha de estudio ya que todas las peticiones son resueltas por el primer componente que las recibe y por lo tanto no hay nada que compartir o trasmitir a ningún otro componente.

Sin embrago vale mencionar que el request mediante los métodos setAttribute(nombre, valor) y getAttribute(nombre) pone a disposición un diccionario (implementado en una tabla de hash) en la que podemos almacenar objetos y recuperarlos en cualquier momento de la petición desde que esta llega al servidor y hasta que se genera la respuesta y se retorna la misma al cliente.

Cabe aclarar también que la colección o diccionario de atributos del request, prevé como nombre un String, pero el valor es un Object, con lo cual podemos almacenar allí los que nos haga falta.

Nota: el ejemplo típico de esto se implementa, cuando se desarrolla un ServletFilter, que no implementaremos en este curso puesto que los Servlets no son el foco principal de nuestro estudio, sino una herramienta para comprender el modelo cliente servidor y su funcionamiento en los servidores de aplicaciones Java EE, se deja pues, planteada la incógnita para que quien lo desee tome la posta y continúe su estudio.

Mantenimiento de estado a nivel de Sesión

La idea aquí es mantener el estado a través de las distintas peticiones de cada cliente de forma que cada quién tenga su propio estado e independiente del estado de otro cliente que pueda estar conectado al mismo tiempo.

Para lograr este esquema de mantenimiento de estado nos vamos a apoyar en la estructura de sesiones del servidor, cada vez que un cliente establece una conexión con el servidor, se agrega en la tabla de sesiones una nueva entrada identificada por un Id de Sesión, y a la cual se asocia una tabla de atributos similar a la tabla que contiene el request y que revisamos en el apartado anterior, pero persistente entre una petición y la siguiente siempre y cuando el cliente mantenga establecida la conexión con el servidor.

En nuestro código tenemos acceso a este repositorio a través del método getSession() del request (revisar el diagrama de clases de la API) que nos devuelve una instancia de una Implementación del servidor de la interfaz javax.servlet.http.HttpSession.

El objeto de la clase HttpSession provee los métodos esenciales para interactuar con el diccionario de valores almacenados, de la misma forma que el request lo hace para los del nivel de petición, es decir, provee un método getAttribute(nombre) que devuelve Object con el valor del atributo almacenado a partir de un nombre de tipo String , y que fue asociado a ese valor mediante el llamado al método setAttribute(nombre, valor) que permite almacenar en la sesión el Object indexado por el nombre antes mencionado.

Además, la clase HttpSession provee otros métodos como:

- getId(): devuelve el identificador de la sesión del usuario
- isNew(): que devuelve true si el cliente nunca ha visto la sesión, normalmente porque acaba de ser creada y false en caso contrario.
- getCreationTime(): que devuelve el instante de creación de la sesión
- getLasAccessedTime(): que devuelve el instante de la última petición de la sesión del cliente.
- invalidate(): que permite eliminar la sesión de un cliente con lo cual se creará una nueva sesión la próxima vez que realice una petición.
- removeAttribute(String): Elimina el objeto asociado al nombre suministrado como parámetro.

Si retomamos nuestro ejemplo del contador de visitas (ver página siguiente), pero intentando que el contador sea independiente para cada cliente que se conecte al servidor, podemos observar el código del método processRequest(…) de ContadorSesionServlet que realiza tal tarea, mostrando además alguna información extra acerca de la sesión.

Solo para remarcar en la línea 46, tomamos una referencia a la sesión a partir del método getSession() del request que nos devuelve un objeto de la clase del servidor que implementa la interfaz HttpSession. Y luego en las líneas 47, 48 y 52 utilizamos los métodos getAttribute(…) y setAttribute(…) para acceder al diccionario de objetos almacenados en la sesión del cliente.

Mantenimiento de estado a nivel de Contexto (Aplicación)

Ahora, en cambio, nos proponemos mantener el estado, pero de forma que sea compartido todos los clientes conectados. Nuevamente apelaremos a un lugar en la memoria del servidor que nos permita mantener valores, al igual que lo hicimos con el request o la sesión, pero compartidos por cualquier petición que se realice a la aplicación desde cualquiera de los clientes conectados en un momento determinado.

```
36      protected void processRequest(HttpServletRequest request, HttpServletResponse response)
37              throws ServletException, IOException
38      {
39
40          response.setContentType("text/html;charset=UTF-8");
41          int contador;
42          try (PrintWriter out = response.getWriter())
43          {
44              /* TODO output your page here. You may use following sample code. */
45              // Usando la Sesion
46              HttpSession session = request.getSession();
47              if (session.getAttribute("contador") != null)
48                  contador = ((Integer) session.getAttribute("contador"));
49              else
50                  contador = 10;
51              contador++;
52              session.setAttribute("contador", contador);
53              if (contador >= 20)
54                  session.invalidate();
55  //                session = request.getSession();
56  //                session.removeAttribute("contador");
57
58              out.println("<!DOCTYPE html>");
59              out.println("<html>");
60              out.println("<head>");
61              out.println("<title>Servlet ContadorServlet</title>");
62              out.println("</head>");
63              out.println("<body>");
64              out.println("<h3>Servlet ContadorServlet at " + request.getContextPath() + "</h3>");
65              out.println("<h1>" + contador + " visitas</h1>");
66              out.println("<hr/>");
67              out.println("<ul>");
68              out.println("<li>Id de sesión: ");
69              out.println(session.getId());
70              out.print("</li><li>Creada: ");
71              out.println(session.getCreationTime());
72              out.print("</li><li>Ultimo acceso: ");
73              out.println(session.getLastAccessedTime());
74              out.println("</li>");
75              out.println("</ul>");
76              out.println("</body>");
77              out.println("</html>");
78          }
79      }
```

Es evidente aquí que el request ya no podrá ser el camino para tomar contacto con este espacio de memoria, porque justamente estamos diciendo que queremos un espacio que sea independiente de la petición y que se mantenga en la memoria desde que lo agreguemos allí y hasta que lo borremos o el servidor se detenga.

El objeto que nos va a dar acceso a este espacio de memoria va a ser nuestro propio servlet que a través un método heredado de la clase GenericServlet, el método getServletContext(), este método no quedó en el diagrama de clase pero se define en la clase GenericServlet y es utilizado por cualquier servlet para tener acceso al contexto del servidor.

Este método nos retorna una instancia de la implementación de la interfaz javax.servlet.ServletContext para el servidor que estemos utilizando.

Del mismo modo que el request y que la sesión, el ServletContext dispone de la colección de atributos que pueden ser administrados a través de los métodos setAttribute(nombre, valor) y getAttribute(nombre) y removeAttribute(nombre). Pero además, también dispone de una colección de parámetros que son accedidos a través de los métodos getInitParameter(nombre) y setInitParameter(nombre, valor) y que brinda acceso a los parámetros de configuración de la aplicación especificados en el archivo de configuración de despliegue de la aplicación.

Cabe aclarar aquí que los parámetros, a diferencia de los atributos, son cadenas de caracteres y no pueden almacenar objetos.

Si volvemos nuevamente a nuestro ejemplo de contador de visitas, ahora volvemos a tener un contador de visitas global ya que el valor guardado en la memoria del contexto de los Servlets será compartido por todos los clientes que ejecuten el servlet y por lo tanto el resultado será similar al que obtuvimos al poner el contador como atributo de instancia de la clase, sin embargo, es importante que comprendamos que dada la naturaleza multihilos de los Servlets, dejar el valor almacenado en la colección segura para multihilos de la memoria del contexto es preferible que mantener un atributo de instancia de la clase que ante el acceso concurrente podría verse afectado si no hacemos nada al respecto.

```
34      protected void processRequest(HttpServletRequest request, HttpServletResponse response)
35              throws ServletException, IOException
36      {
37
38          response.setContentType("text/html;charset=UTF-8");
39          int contador;
40          try (PrintWriter out = response.getWriter())
41          {
42              /* TODO output your page here. You may use following sample code. */
43              // Usando el contexto
44              ServletContext contexto = getServletContext();
45              System.err.println("Clase del ServletContext: " + contexto.getClass().getName());
46              if (contexto.getAttribute("contador") != null)
47                  contador = ((Integer) contexto.getAttribute("contador"));
48              else
49                  contador = Integer.parseInt(contexto.getInitParameter("contador-inicial"));
50              contador++;
51              contexto.setAttribute("contador", contador);
52
53              out.println("<!DOCTYPE html>");
54              out.println("<html>");
55              out.println("<head>");
56              out.println("<title>Servlet ContadorServlet</title>");
57              out.println("</head>");
58              out.println("<body>");
59              out.println("<h3>Servlet ContadorServlet at " + request.getContextPath() + "</h3>");
60              out.println("<h1>" + contador + " visitas</h1>");
61              out.println("</body>");
62              out.println("</html>");
63          }
64      }
```

La lógica es similar al ejemplo con la sesión, a partir de la línea 44 podemos observar la interacción con el contexto de Servlets a partir de la referencia contexto de tipo ServletContext que recibe el objeto como resultado de la invocación al método getServletContext() utilizado sin referencia ya que es un método propio por herencia desde la clase GenericServlet.

Mantenimiento de estado del lado del Cliente

En el caso de mantener el estado del lado del servidor, tenemos del mismo modo 2 opciones, una en la que desde el servidor no tenemos control alguno más que asumir que en las peticiones vienen enviados como datos extra de la petición la información asociada al estado y que quedará a cargo del desarrollador del cliente arbitrar los medios para que esto pase, o la otra en la que sí tendremos la tarea de administrar lo que el cliente almacena y el envío y recepción del estado se hará igualmente en cada una de las peticiones pero administrador y definido por el código del servidor.

Esta segunda opción es la que nos ocupa y vamos a introducir brevemente a continuación para cerrar la ficha y completar las alternativas de manejo de estado administrado por el código del servidor.

Mantenimiento de estado en el cliente por Cookies

El mecanismo por el cual, el servidor puede almacenar estado en el cliente, asegurando que el cliente lo envíe con cada petición y así disponer de la información durante el procese de la siguiente petición es a través de la utilización de ***cookies***.

Las Cookies, son porciones de datos que el servidor envía al cliente y este las almacena de alguna forma en su cache (en principio como pequeños archivos de texto), según rumores reciben ese nombre por las migras de pan de la historia de Hansel y Gretel, los rastros de nuestra navegación quedan almacenados en cookies en nuestro navegador. Luego el cliente cada vez que realiza una petición al servidor, incluye en la petición todas las cookies no caducas que estén asociadas a ese servidor y de esa forma cuando el servidor recibe los datos de la petición recibe también las cookies asociadas a la misma.

En el servidor están representadas por la clase javax.servlet.http.Cookie, esencialmente esta clase tiene un constructor que recibe dos parámetros de tipo String, el nombre y el valor y por otro lado podemos establecer algunos valores extra como el tiempo de expiración, el dominio o el path asociado pero escapan al objetivo del apartado de la mantención de estado.

En la última versión del contador de visitas que se adjunta en el ejemplo asociado a la ficha, implementado en la clase ContadorCookieServlet, utilizamos cookies para mantener el estado del contador, cabe reiterar que en este caso el contador será independiente para cada cliente conectado y que a diferencia del funcionamiento con la sesión el mismo va a sobrevivir a la desconexión de nuestro cliente al servidor y va a reiniciarse solo cuando en el navegador (el cliente) solicitemos se borren las cookies locales.

```
38      protected void processRequest(HttpServletRequest request, HttpServletResponse response)
39              throws ServletException, IOException
40      {
41
42          response.setContentType("text/html;charset=UTF-8");
43          int contador;
44          try (PrintWriter out = response.getWriter())
45          {
46              /* TODO output your page here. You may use following sample code. */
47              // Usando las cookies
48              Cookie[] cookies = request.getCookies();
49              Optional<Cookie> optional = Arrays.stream(cookies)
50                                      .filter(c -> "cuenta".equals(c.getName()))
51                                      .findFirst();
52
53              if(optional.isPresent())
54                  contador = Integer.parseInt(optional.get().getValue());
55              else
56              {
57                  ServletContext contexto = getServletContext();
58                  contador = Integer.parseInt(contexto.getInitParameter("contador-inicial"));
59              }
60              contador++;
61
62              Cookie respCookie = new Cookie("cuenta", "" + contador);
63              response.addCookie(respCookie);
64
65              out.println("<!DOCTYPE html>");
66              out.println("<html>");
67              out.println("<head>");
68              out.println("<title>Servlet ContadorServlet</title>");
69              out.println("</head>");
70              out.println("<body>");
71              out.println("<h3>Servlet ContadorServlet at " + request.getContextPath() + "</h3>");
72              out.println("<h1>" + contador + " visitas</h1>");
73              out.println("</body>");
74              out.println("</html>");
75          }
76      }
```

La particularidad que podemos notar en la línea 48, es que el request mediante el método getCookies() nos retorna un array con todas las cookies que envió el servidor y por lo tanto la primera tarea a realizar será buscar nuestra cookie por nombre en dicho array, he agregado el código de la búsqueda aquí mismo entre las líneas 49 y 53, en el caso de encontrar la cookie tomo de allí el valor del contador y si no estuviera la sigo tomando del parámetro de configuración de la aplicación mediante el contexto de Servlets.

Vale hacer notar también que cuando probemos este modelo la cookie va a sobrevivir incluso a un reinicio del servidor y de esta forma podemos ver y analizar los alcances de las diferentes técnicas de mantención de estado en las aplicaciones Cliente – Servidor.

9.] Referencias.

- Java EE 8 Platform Documentation: https://javaee.github.io/glassfish/documentation
- Payara Foundation: https://www.payara.org y https://www.payara.fish
- CodeJava.net: https://www.codejava.net
- Wikipedia: https://es.wikipedia.org y https://en.wikipedia.org