

Práctica 2:Filters, Sessions and Listeners

Morales Flores Víctor Leonel

Marzo 2019

1 Introducción

Cuando hablamos del API de servlets es común escuchar conceptos como la clase Servlet, los objetos de la clase HttpRequest y HttpResponse pero muy rara vez se escucha sobre los 'Filters' o los 'Listener'. Estos últimos también pertenecen al API de Servlets y el conocer sobre sus alcances y limitaciones, así como la forma en que trabajan, nos permitirán crear aplicaciones más robustas y con códigos más simples y limpios.

Los Filters son clases de java que como su nombre nos lo indica, nos permiten filtrar solicitudes en nuestro sistema. Para crear un filtro nosotros tenemos que importar de la clase javax.servlet.Filter y definir una clase que implemente Filter. La potencia de los filtros surge porque como programadores podemos decidir todas las peticiones que lleguen a nuestro sistema o peticiones a rutas muy específicas y decidir cómo debe actuar el filtro(dejar pasar la petición o bloquearla) en base a nuestras reglas del negocio, por ejemplo.

Por otra parte, los listener son clases que nos permiten tener conocimiento de todo lo que está pasando en nuestro sistema. Los listener son levantados cuando nuestro web container habilita nuestra aplicación para recibir peticiones y son bastante versátiles ya que podemos decidir qué tipo de eventos queremos filtrar. Entre todos los eventos posibles, algunos de ellos son los cambios en el ciclo de vida o atributos de nuestro ServletContext, el ciclo de vida y los cambios en los atributos de nuestras sesiones, el ciclo de vida y el cambio de los atributos en nuestros objetos request, etc.

A lo largo de la práctica se desarrollarán ejercicios para que se pueda comprender cómo es que funcionan estas clases y se le empezará a dar aplicaciones prácticas a estas clases para ver el potencial que tienen y cómo nos pueden simplificar algunas cuestiones a lo largo del desarrollo de nuestras aplicaciones web.

2 Desarrollo

2.1 Archivo pom.xml

Para realizar esta práctica utilizamos el IDE de Eclipse y generamos un proyecto Maven para agregar nuestras dependencias sin tener que descargarlas. Por ello, para ejecutar nuestro proyecto es necesario tener acceso a Internet.

Debido a que para el proyecto se usó un código fuente base proporcionado por el profesor que usar el framework de Hibernate es que tendremos que agregar varias dependencias para su correcto funcionamiento. EL código en nuestro archivo pom.xml debe tener lo siguiente:

```
<properties>
    <!-- Versiones usadas de los frameworks -->
    <java.version>1.8</java.version>
    <spring.version>4.3.8.RELEASE</spring.version>
    <hibernate.version>5.2.10.Final</hibernate.version>
```

```

<jetty.version>8.1.8.v20121106</jetty.version>
<testng.version>6.11</testng.version>
<!-- Define el formato de fecha con el que debe ser generada la estampa
      de tiempo en cada ejecución de construcción de maven -->
<maven.build.timestamp.format>yyyyMMddHHmm</maven.build.timestamp.format>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
<build>
  <finalName>tarea6</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>9.4.1212</version>
  </dependency>
  <!-- ***** -->
  <!-- Dependencias de Hibernate -->
  <!-- ***** -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${hibernate.version}</version>

```

```

</dependency>
<dependency>
    <groupId>org.antlr</groupId>
    <artifactId>antlr-runtime</artifactId>
    <version>3.5.2</version>
</dependency>
<dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jta_1.1_spec</artifactId>
    <version>1.1.1</version>
</dependency>
<dependency>
    <groupId>org.hibernate.common</groupId>
    <artifactId>hibernate-commons-annotations</artifactId>
    <version>5.0.1.Final</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>${hibernate.version}</version>
</dependency>
<dependency>
    <groupId>org.hibernate.javax.persistence</groupId>
    <artifactId>hibernate-jpa-2.1-api</artifactId>
    <version>1.0.0.Final</version>
</dependency>
<dependency>
    <groupId>org.jboss</groupId>
    <artifactId>jandex</artifactId>
    <version>2.0.0.Final</version>
</dependency>
<dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>3.1</version>
</dependency>
<dependency>
    <groupId>com.mchange</groupId>
    <artifactId>c3p0</artifactId>
    <version>0.9.5.2</version>
</dependency>
<dependency>
    <groupId>javax.transaction</groupId>
    <artifactId>jta</artifactId>
    <version>1.1</version>
</dependency>

```

```

        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>42.2.5</version>
        </dependency>
    </dependencies>

```

2.2 Un Filter básico

2.2.1 Código java

Una vez agregadas nuestras dependencias comenzaremos por ver cómo es que funciona un filtro. Para ello primero crearemos una clase donde importaremos las siguientes librerías y que implemente a Filter. EL ejemplo se proporciona a continuación:

```

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

```

```

public class Filter1 implements Filter
{

}

```

Lo siguiente es implementar el método que se encarga de cumplir la tarea de filtrado. Este método se llama `doFilter()` y es el que será invocado cuando se intente hacer una petición hacia la ruta o rutas donde queremos estar filtrando.

```

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
throws IOException, ServletException {

    chain.doFilter(request, response);

}

```

2.2.2 Configuración del Deployment Descriptor

Este es el código más simple para hacer un Filter pero aún no hace nada. Con `chain.doFilter(...)` permitimos que la petición continúe hacia el recurso que solicitó el cliente pero el filtro aún no está del todo listo. Para que nuestro Filter se levante junto con nuestra aplicación web es necesario colocarlo en nuestro archivo `web.xml` (deployment descriptor). Para cumplir con esta tarea las etiquetas que deberemos agregar al archivo xml son las siguientes:

```
<filter>
    <filter-name>Filter1</filter-name>
    <display-name>Filter1</display-name>
    <description></description>
    <filter-class>mx.ipn.escom.wad.filters.Filter1</filter-class>
</filter>

<filter-mapping>
    <filter-name>Filter1</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

En el primer bloque le asignaremos el nombre de nuestra Filtro y la clase a la que está haciendo referencia. En el `filter-mapping` indicamos el nombre de nuestro filtro (debe coincidir con el nombre que le colocamos en el primer bloque) y el recurso que queremos que filtre. En el código de ejemplo `(/*)` estamos indicando al filtro que queremos que filtre todas las peticiones que lleguen a nuestra aplicación.

Hasta aquí el filtro ya estará funcionando y se levantará junto con nuestra aplicación web pero aún no veremos nada. Para ver qué está haciendo el filtro agregaremos la siguiente línea de código a nuestro método `doFilter()`.

```
HttpServletRequest req=(HttpServletRequest)request;
Date fecha =new Date();
DateFormat df=new SimpleDateFormat("dd/MM/yy | HH:mm:ss");
String url=req.getRequestURL().toString();
System.out.println("Filter: "+req.getRemoteAddr()+" | "+df.format(fecha)+
" | "+req.getMethod()+" | "+url);
```

Ahora, cada vez que se reciba una petición el filtro nos imprimirá por consola la dirección remota de la que se está intentando acceder, la fecha y hora de la petición, el método por el que se está haciendo la petición y el recurso que está siendo solicitado.

Para este ejemplo hay que tomar en cuenta que el parámetro que recibe el método `doFilter(...)` es un objeto `request` de la clase `ServletRequest` por lo que para poder recuperar algunos datos del ejemplo que se muestra arriba hay que hacer un cast a un objeto de la clase `HttpServletRequest`.

NOTA: Tras cada modificación en el código fuente es necesario que reinicies el servicio para poder ver los cambios.

2.3 Un filtro más complejo

2.3.1 Modificaciones a nuestra clase Filter1

Ahora que se hemos visto que cada vez que hagamos una petición hacia nuestro sitio, antes de acceder pasará por nuestro y filtro y este simplemente deja pasar la petición, haremos modificación en el código para bloquear algunos recursos de nuestra aplicación web.

La forma en que haremos esto será a través de la URL que solicita el cliente. Para este caso decidimos bloquear todos los recursos que se encuentren en nuestra ruta `'/3cm6-practica2-mfvl/privado'`.

Primero recuperaremos en una cadena la URL a la que quiere acceder el cliente a través de nuestro objeto `Request(HttpRequest)` y crearemos una cadena que tenga la ruta descrita anteriormente.

Con una condición verificaremos si el URL al que se quiere dirigir contiene la ruta que queremos bloquear. Si es así asignaremos a nuestra variable booleana `'permitido'` el valor de falso, en caso contrario el valor será verdadero. Para verificar si hay o no una sesión iniciada crearemos un objeto de nuestra clase `HttpSession` y recuperaremos la sesión de nuestro objeto `request`.

Posteriormente recuperaremos el atributo de sesión `'authenticated_user'` y verificaremos que este objeto recuperado de la sesión no sea nulo y que además sea una instancia de la clase `'Usuario'`. Si se cumplen estas dos condiciones entonces el filtro dejará pasar la petición del usuario, en caso contrario responderá con un error `'666'`.

En cualquiera de los casos nos debe mostrar en consola la petición que se realizó y si el recurso es un recurso al que no esté permitido acceder, deberá mostrar `'NOT ALLOWED'`.

El código que se colocará en nuestro método `doFilter(...)` de nuestro `Filter1` será el siguiente:

```
String recursoPrivado="/3cm6-practica2-mfvl/privado";
boolean permitido=false;
HttpSession session=req.getSession();
Object o=session.getAttribute(NombreObjetosSession.USER);

if(url.indexOf(recursoPrivado)!=-1)//Retorna -1 si no lo contiene
{
    permitido=false;
}
else
{
    permitido=true;
}
if(o!= null && o instanceof Usuario)
{
    System.out.println("Filter: "+req.getRemoteAddr()+
        " | "+df.format(fecha)+" | "+req.getMethod()+
```

```

        " | "+req.getRequestURL());
        chain.doFilter(request, response);
    }
    else
    {
        System.out.println("Filter: "+req.getRemoteAddr()+
            " | "+df.format(fecha)+" | "+req.getMethod()+
            " | "+req.getRequestURL()+" | NOT ALLOWED");
        HttpServletResponse resp=(HttpServletResponse)response;
        if(permitido)
            chain.doFilter(request, response);
        else
            resp.sendError(666);
    }
}

```

Es importante que `chain.doFilter(...)` se encuentre dentro del condicional para que este cumpla su cometido, si se coloca fuera de los condicionales veremos que el comportamiento de filtrado no es el esperado.

2.3.2 Creando página de error HTML

Para evitar que al generar un error nos regrese una página sin mucho estilo podemos crear nosotros nuestras propias páginas de error para que nuestro deployment descriptor nos dirija hacia ella cuando se genere el error.

En este caso la página no tendrá mucho formato pero el objetivo del ejercicio es ver cómo puede generar el programador estas páginas de error. Posteriormente el programador puede colocarle el diseño que desee.

En este ejemplo usaremos un archivo JSP llamado `error.jsp` con el siguiente código:

```

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title>Recurso no encontrado</title>
</head>
<body>
    <h1>Disculpe las molestias, el recurso al que usted desea acceder
    no se encuentra disponible sin sesión.</h1>
</body>
</html>

```

2.3.3 Definiendo nuestro código de error(666) en el Deployment Descriptor

Para que este ejercicio funcione debemos indicarle a nuestro Deployment Descriptor hacia dónde debe redirigirnos cuando se presente este error(666). La

forma de hacerlo es muy sencillo y nos da la opción de crear páginas mucho más personalizadas y alineadas con la filosofía del proyecto en cuestión.

Las líneas de código que se deben agregar a nuestro DeploymentDescriptor(web.xml) son las siguientes:

```
<error-page>
    <error-code>666</error-code>
    <location>/error.jsp</location>
</error-page>
```

2.4 Subiendo a nuestro usuario al objeto de Sesión

En este punto de la práctica si todo ha funcionado correctamente, todos los recursos que estén en nuestra ruta '/3cm6-practica2-mfv1/privado' serán inaccesibles si no tienes una sesión activa. Para desarrollar esta parte de la práctica se hizo uso de los códigos realizados a lo largo de la práctica 1 y la tarea 6. En ellas se desarrollaron los puntos para acceder a la base de datos, autenticar al usuario y hacemos una consulta a la base de datos para generar nuestra tabla dinámica.

El 'Servlet5' que genera nuestra tabla dinámica tras hacer la consulta en la base de datos lo colocaremos como un recurso privado.

La clase en la que subiremos a nuestro usuario a la sesión será 'LoginCtrl' en nuestro paquete 'mx.ipn.escom.wad.tarea6.controller'.

En esta clase ya se habían realizado las consultas necesarias par autenticar al usuario en la práctica anterior, lo único que se hará en esta ocasión es agregar a los atributos de nuestra sesión el objeto Usuario que se instanció.

La parte importante de esta sección es aprender a subir objetos a nuestros atributos de sesión. Para ello hay que usar el método setAttribute(..) en nuestro objeto de sesión(instancia de HttpSession)

```
usuario = LoginBs.login(user, pswd);
HttpSession session = request.getSession();
session.setAttribute(NombreObjetosSession.USER, usuario);
if(usuario!=null)
{
    RequestDispatcher rd = request.getRequestDispatcher("../privado/Home");
    rd.forward(request, response);
}
else
{
    response.sendRedirect("../?error=true");
}
```

A través de nuestro objeto LoginBs nosotros obtenemos una instancia de la clase Usuario. Si el usuario no existe en nuestra base de datos, entonces LoginBs retornará un objeto nulo. Si el usuario existe y la contraseña coincide nos dará la instancia de Usuario con los datos del usuario que solicitó el acceso.

Nosotros podemos subir cualquier objeto a los atributos de sesión. Para subir un objeto hay que recordar que se requiere de un par clave-valor. En este ejemplo nuestro valor es el objeto Usuario y nuestra clave para poder recuperarla en el futuro es la cadena estática 'authenticated_user', la cual está definida en la clase NombreObjetosSession y se obtiene a través de 'NombreObjetosSession.USER'.

Como se puede observar en el código, si nuestra instancia de la clase Usuario es distinta de nula nos redigirá a la ruta '/3cm6-practica2-mfvl/privado/Home' pero si es un objeto nulo nos regresará a la página de inicio para intentar autenticarnos otra vez.

Una vez autenticado puedes verificar que ahora puedes acceder a los recursos que eran privados sin ningún problema.

2.5 Los Listener

Como ya se mencionó anteriormente, los listener pueden ser muy potentes si sabemos usarlos adecuadamente. En esta sección crearemos una clase Listener que implementará a ServletContextListener, ServletContextAttributeListener, HttpSessionListener, HttpSessionAttributeListener, ServletRequestAttributeListener, ServletRequestListener. Por cada implementación que haga se creará un método que será invocado cuando se ejecute ese evento en nuestro sistema.

2.5.1 El código básico para un Listener

Para poder crear un Listener debemos importar algunas librerías, así como definir la clase de la forma correcta. La sintaxis básica para crear una clase que nos sirva como Listener es la siguiente:

```
import javax.servlet.ServletContextAttributeEvent;
import javax.servlet.ServletContextAttributeListener;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletRequestAttributeEvent;
import javax.servlet.ServletRequestAttributeListener;
import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionAttributeListener;
import javax.servlet.http.HttpSessionBindingEvent;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

public class Listener1 implements ServletContextListener,
ServletContextAttributeListener, HttpSessionListener,
HttpSessionAttributeListener, ServletRequestAttributeListener,
ServletRequestListener {

}
```

El número de clases que importemos, así como el número de métodos dependerá de cuál es la tarea que cumpla el listener. En este caso solo imprimiremos en consola cada que se ejecute un evento.

2.5.2 Implementando a las clases

Debido a que la tarea que debe cumplir el Listener tras cada evento depende del objetivo del listener, en esta práctica solo colocaremos la sintaxis de cada uno de los métodos que se deben implementar y una descripción sobre lo que hace

```
public void sessionCreated(HttpSessionEvent se) { /*Codigo*/ }
```

Este método se invocará cuando se cree una sesión en nuestra aplicación.

```
public void attributeRemoved(ServletContextAttributeEvent event) { /*Codigo*/ }
```

attributeRemoved se ejecutará cuando eliminemos algún atributo de nuestro ServletContext.

```
public void attributeAdded(ServletRequestAttributeEvent srae) { }
```

Cada que agreguemos un atributo a nuestro Servlet Request este método de nuestro Listener será invocado.

```
public void attributeReplaced(HttpSessionBindingEvent event) { }
```

Si reemplazamos un atributo de nuestra sesión este método se ejecuta.

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

Método ejecutado cuando se inicializa el ServletContext(al levantar el servicio).

```
public void attributeAdded(ServletContextAttributeEvent event) { }
```

Método invocado cuando agregamos un atributo a nuestro ServletContext

```
public void requestDestroyed(ServletRequestEvent sre) { }
```

Método ejecutado cuando se destruye un objeto request.

```
public void attributeRemoved(ServletRequestAttributeEvent srae) { }
```

El método es invocado cuando se elimina un atributo de un objeto request.

```
public void requestInitialized(ServletRequestEvent sre) { }
```

Se ejecuta este método cuando se inicializa un objeto request.

```
public void sessionDestroyed(HttpSessionEvent se) { }
```

En cuanto se genere un evento que elimine nuestra sesión este método será invocado.

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

```
public void attributeReplaced(ServletRequestAttributeEvent srae) { }
```

Método invocado cuando un atributo de nuestro objeto request es remplazado.

```
public void attributeAdded(HttpSessionBindingEvent event) { }
```

Método que es invocado si agregamos un atributo a nuestra sesión.

```
public void attributeRemoved(HttpSessionBindingEvent event) { }
```

Método que es invocado si eliminamos un atributo a nuestra sesión.

```
public void attributeReplaced(ServletContextAttributeEvent event) { }
```

Se ejecuta el bloque de código que se encuentre al interior del método cuando se reemplace un atributo de nuestro ServletContext.

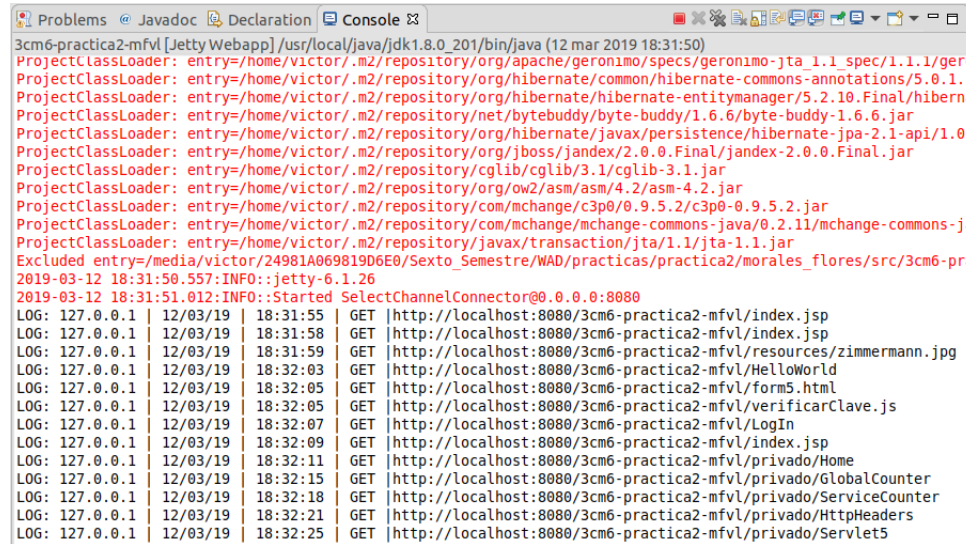
Los filtros tienen muchos eventos que nos podrían ser de utilidad para desarrollar aplicaciones más robustas y poder realizar acciones cuando alguna de ellas ocurra. Con ello podríamos automatizar muchas tareas en la aplicación.

2.5.3 Definiendo el Listener en el Deployment Descriptor

Para que nuestro Listener se levante junto con nuestro servicio debe ser definido en nuestro Deployment Descriptor. Para hacer esto la sintaxis es la siguiente:

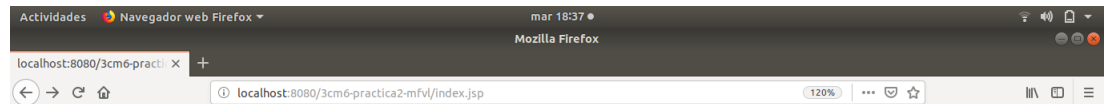
```
<listener>
  <listener-class>
    mx.ipn.escom.wad.listeners.Listener1
  </listener-class>
</listener>
```

3 Pruebas



```
3cm6-practica2-mfvl [Jetty Webapp] /usr/local/java/jdk1.8.0_201/bin/java (12 mar 2019 18:31:50)
ProjectClassLoader: entry=/home/victor/.m2/repository/org/apache/geronimo/specs/geronimo-jta_1.1_spec/1.1.1/geronimo-jta-1.1.1.jar
ProjectClassLoader: entry=/home/victor/.m2/repository/org/hibernate/common/hibernate-commons-annotations/5.0.1.Final/hibernate-commons-annotations-5.0.1.Final.jar
ProjectClassLoader: entry=/home/victor/.m2/repository/org/hibernate/hibernate-entitymanager/5.2.10.Final/hibernate-entitymanager-5.2.10.Final.jar
ProjectClassLoader: entry=/home/victor/.m2/repository/net/bytebuddy/byte-buddy/1.6.6/byte-buddy-1.6.6.jar
ProjectClassLoader: entry=/home/victor/.m2/repository/org/hibernate/javax/persistence/hibernate-jpa-2.1-api/1.0.0.Final/hibernate-jpa-2.1-api-1.0.0.Final.jar
ProjectClassLoader: entry=/home/victor/.m2/repository/org/jboss/jandex/2.0.0.Final/jandex-2.0.0.Final.jar
ProjectClassLoader: entry=/home/victor/.m2/repository/org/ow2/asm/asm/4.2/asm-4.2.jar
ProjectClassLoader: entry=/home/victor/.m2/repository/com/mchange/c3p0/0.9.5.2/c3p0-0.9.5.2.jar
ProjectClassLoader: entry=/home/victor/.m2/repository/com/mchange/mchange-commons-java/0.2.11/mchange-commons-java-0.2.11.jar
ProjectClassLoader: entry=/home/victor/.m2/repository/javax/transaction/jta/1.1/jta-1.1.jar
Excluded entry=/media/victor/24981A069819D6E0/Sexto_Semestre/WAD/practicas/practica2/morales_flores/src/3cm6-practica2-mfvl
2019-03-12 18:31:50.557:INFO::jetty-6.1.26
2019-03-12 18:31:51.012:INFO::Started SelectChannelConnector@0.0.0.0:8080
LOG: 127.0.0.1 | 12/03/19 | 18:31:55 | GET | http://localhost:8080/3cm6-practica2-mfvl/index.jsp
LOG: 127.0.0.1 | 12/03/19 | 18:31:58 | GET | http://localhost:8080/3cm6-practica2-mfvl/index.jsp
LOG: 127.0.0.1 | 12/03/19 | 18:31:59 | GET | http://localhost:8080/3cm6-practica2-mfvl/resources/zimmermann.jpg
LOG: 127.0.0.1 | 12/03/19 | 18:32:03 | GET | http://localhost:8080/3cm6-practica2-mfvl/HelloWorld
LOG: 127.0.0.1 | 12/03/19 | 18:32:05 | GET | http://localhost:8080/3cm6-practica2-mfvl/form5.html
LOG: 127.0.0.1 | 12/03/19 | 18:32:05 | GET | http://localhost:8080/3cm6-practica2-mfvl/verificarClave.js
LOG: 127.0.0.1 | 12/03/19 | 18:32:07 | GET | http://localhost:8080/3cm6-practica2-mfvl/LogIn
LOG: 127.0.0.1 | 12/03/19 | 18:32:09 | GET | http://localhost:8080/3cm6-practica2-mfvl/index.jsp
LOG: 127.0.0.1 | 12/03/19 | 18:32:11 | GET | http://localhost:8080/3cm6-practica2-mfvl/privado/Home
LOG: 127.0.0.1 | 12/03/19 | 18:32:15 | GET | http://localhost:8080/3cm6-practica2-mfvl/privado/GlobalCounter
LOG: 127.0.0.1 | 12/03/19 | 18:32:18 | GET | http://localhost:8080/3cm6-practica2-mfvl/privado/ServiceCounter
LOG: 127.0.0.1 | 12/03/19 | 18:32:21 | GET | http://localhost:8080/3cm6-practica2-mfvl/privado/HttpHeaders
LOG: 127.0.0.1 | 12/03/19 | 18:32:25 | GET | http://localhost:8080/3cm6-practica2-mfvl/privado/Servlet5
```

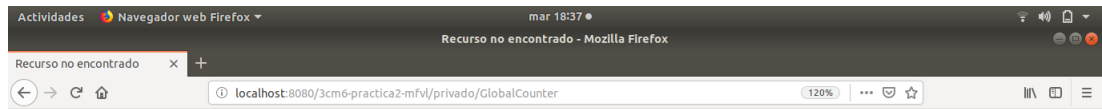
Figure 1: Filtro a todas las rutas de la aplicación



Practica 2

- Recursos públicos:
 - [Imagen](#)
 - [HelloWorld](#)
 - [Form 5](#)
 - [Log In](#)
- Recursos privados:
 - [Home](#)
 - [GlobalCounter](#)
 - [Service Counter](#)
 - [HttpHeadres](#)
 - [Servlet 5](#)

Figure 2: Recursos privados y públicos



Disculpe las molestias, el recurso al que usted desea acceder no se encuentra disponible sin sesión.

Figure 3: Mensaje de error(666) cuando se dirige a un recurso privado

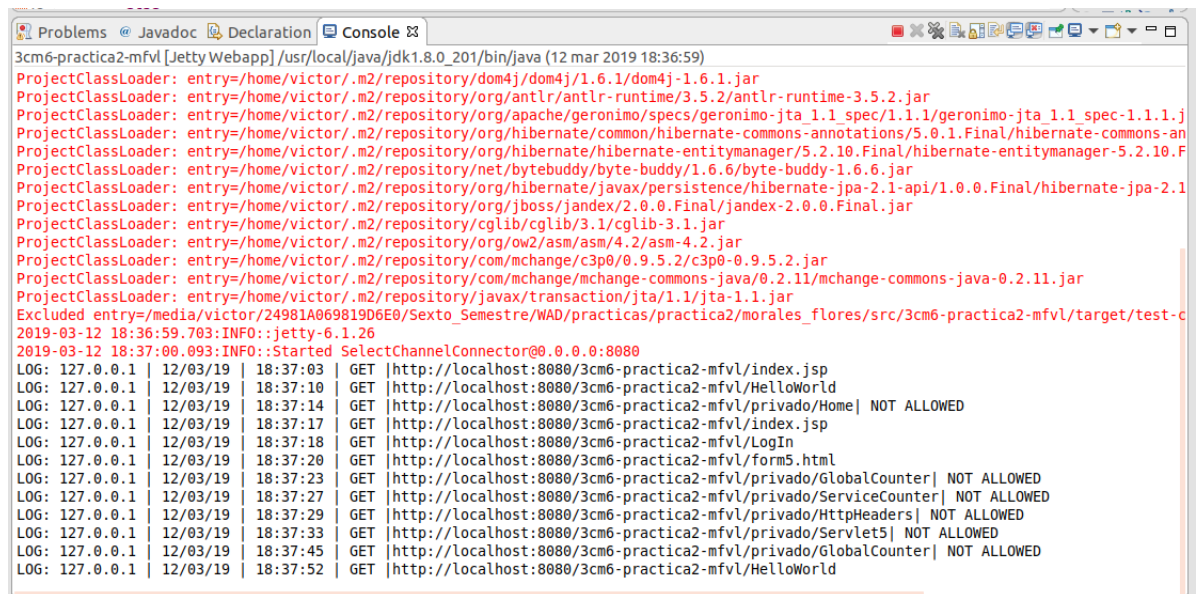


Figure 4: Filter1 bloqueando solicitudes a recursos privados(salidas en consola)

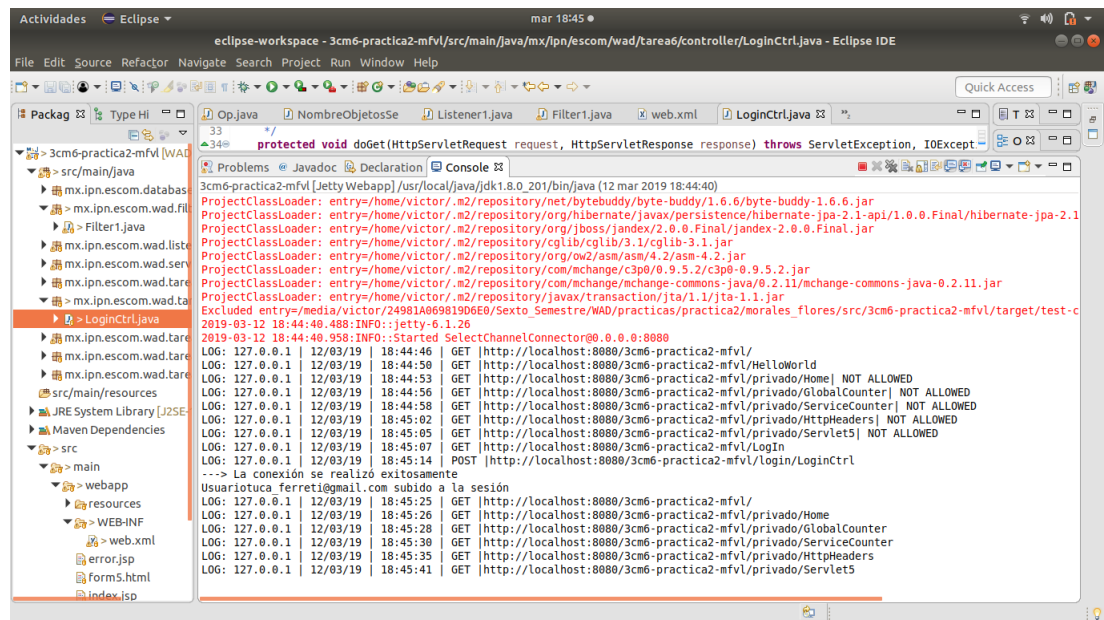


Figure 5: Filter1 bloqueando peticiones antes de inicio sesión y permitiendo el paso después de inicio de sesión

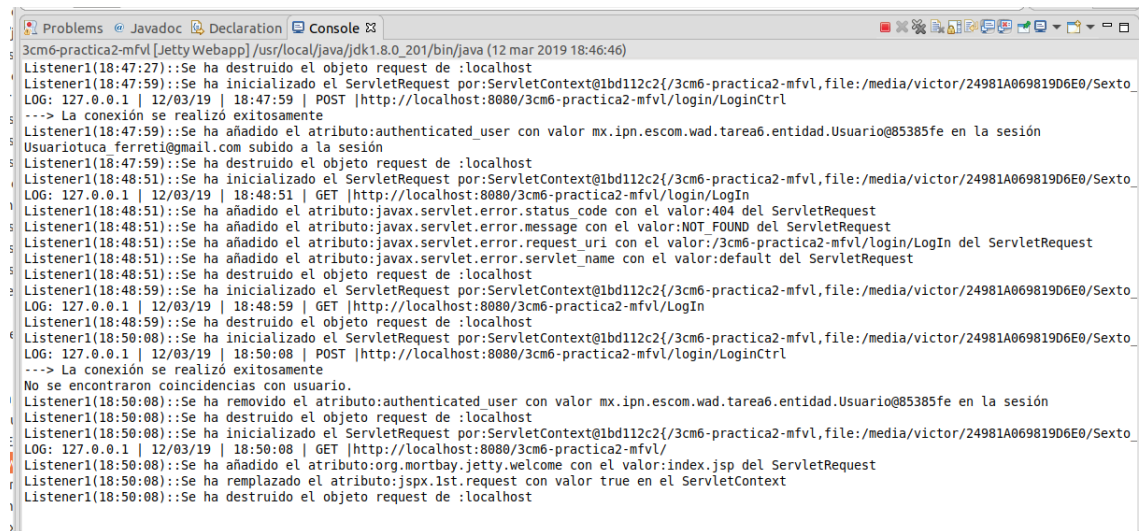


Figure 6: Listener ejecutando métodos durante la navegación por la aplicación

4 Conclusiones

Cuando se empiezan a desarrollar aplicaciones con el API de Servlets usualmente se habla mucho de los objetos `HttpRequest`, `HttpResponse`, las sesiones y los Servlets en sí, pero casi no se habla de los filtros o listeners que también son parte del API. El saber cuál es el alcance de estas clases que se implementan pueden ayudarnos a ahorrarnos código, así como dotar a la aplicación web de mayor seguridad para el contenido sensible y poder saber en todo momento qué está pasando en nuestra aplicación.

Las sesiones son un tema que sí es abordado comúnmente y esto es porque con las sesiones solucionamos el problema del alcance de nuestros objetos `Request`. A través de las sesiones podemos recuperar valores u objetos creados desde cualquier parte de nuestra aplicación siempre y cuando hayamos subido ese objeto a la sesión.