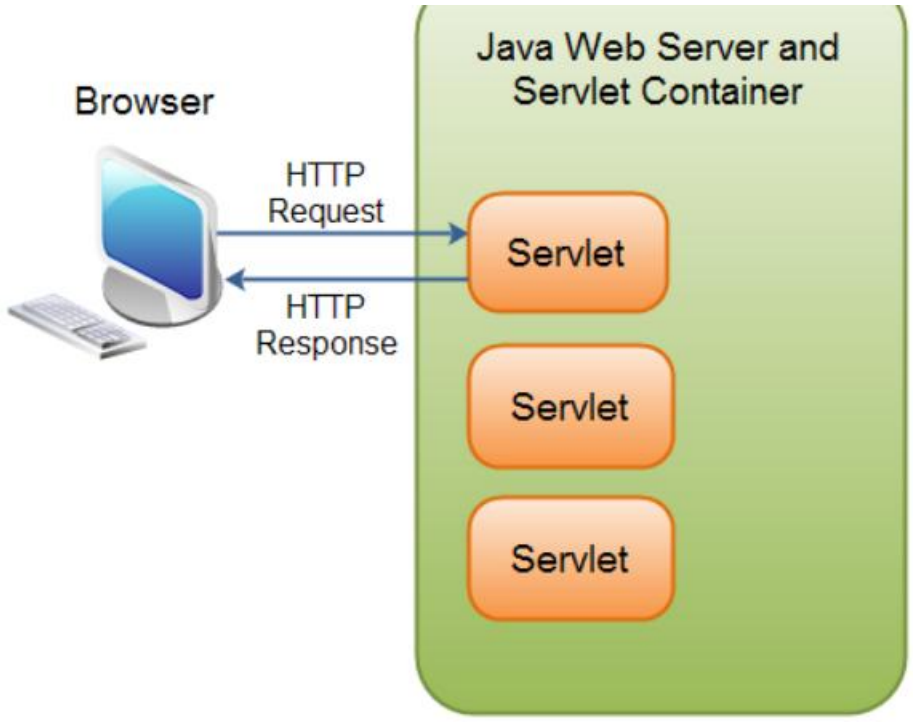
JAVA Servlets

Java Servlets is a web technology for Java. It was the first web technology for Java and many new web technologies have arrived since. Still, Java Servlets are very useful, both to know, and for certain use cases.
Java Servlets are part of the Java Enterprise Edition (Java EE). You will need to run your Java Servlets inside a Servlet compatible "Servlet Container" (e.g. web server) in order for them to work.

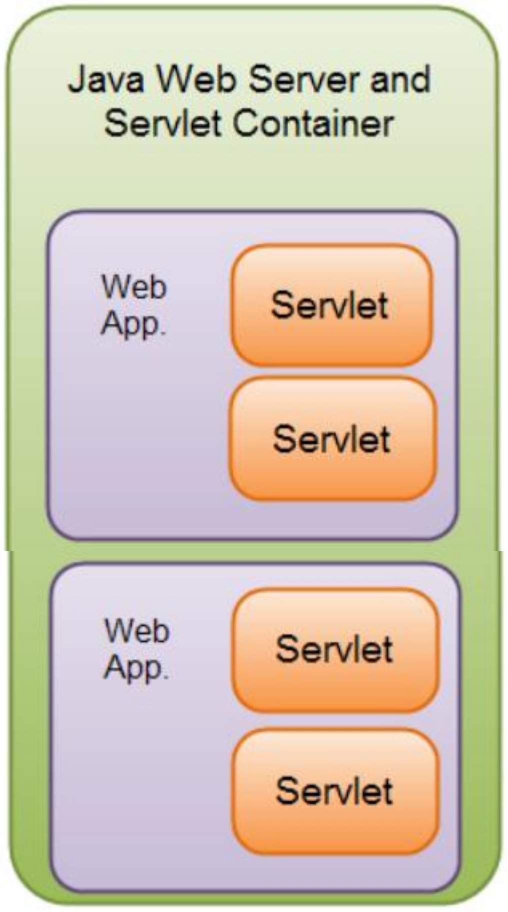
❖ What is a Servlet?

A Java Servlet is a Java object that responds to HTTP requests. It runs inside a Servlet container.



Servlets inside a Java Servlet Container

A Servlet is part of a Java web application. A Servlet container may run multiple web applications at the same time, each having multiple servlets running inside.



Web applications with multiple servlets inside a Java Servlet container

A Java web application can contain other components than servlets. It can also contain Java Server Pages (JSP), Java Server Faces (JSF) and Web Services.

- ## HTTP Request and Response

The browser sends an HTTP request to the Java web server. The web server checks if the request is for a servlet. If it is, the servlet container is passed the request. The servlet container will then find out which servlet the request is for, and activate that servlet. The servlet is activated by calling the Servlet.service() method.
Once the servlet has been activated via the service() method, the servlet processes the request, and generates a response. The response is then sent back to the browser.

- ## Servlet Containers

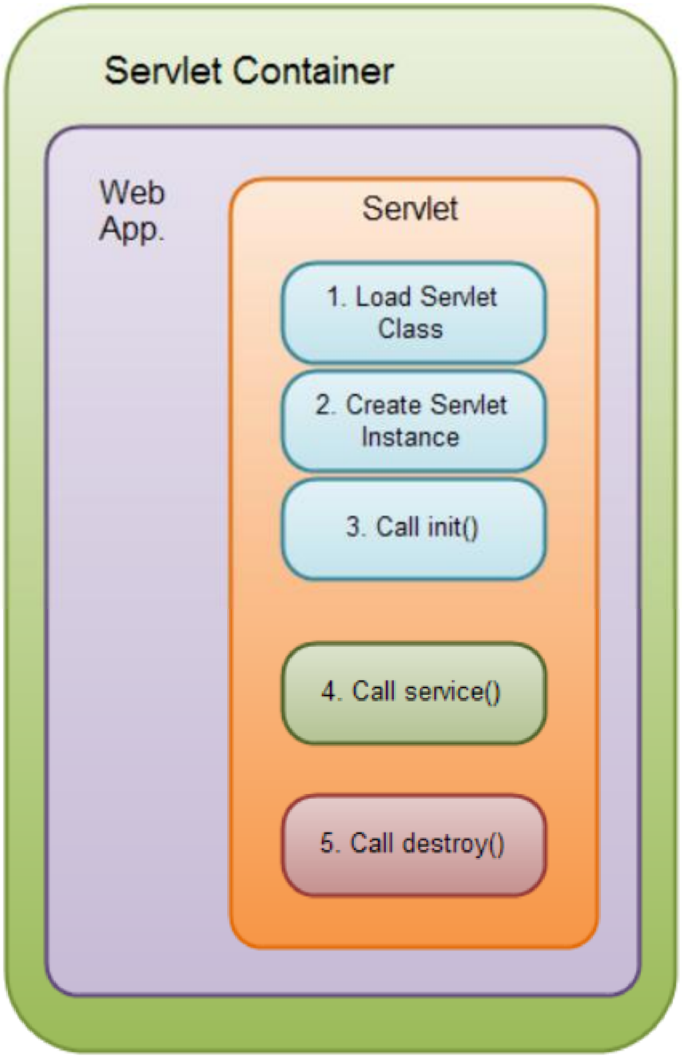
Java servlet containers are usually running inside a Java web server. A few common well known, free Java web servers are: Jetty & Tomcat

❖ Servlet Life Cycle

A servlet follows a certain life cycle. The servlet life cycle is managed by the servlet container. The life cycle contains the following steps:

1. Load Servlet Class.
2. Create Instance of Servlet.
3. Call the servlets `init()` method.
4. Call the servlets `service()` method.
5. Call the servlets `destroy()` method.

Step 1, 2 and 3 are executed only once, when the servlet is initially loaded. By default the servlet is not loaded until the first request is received for it. You can force the container to load the servlet when the container starts up though.
Step 4 is executed multiple times - once for every HTTP request to the servlet.
Step 5 is executed when the servlet container unloads the servlet.



The Java Servlet life cycle

1. Load Servlet Class

Before a servlet can be invoked the servlet container must first load its class definition. This is done just like any other class is loaded.

2. Create Instance of Servlet

When the servlet class is loaded, the servlet container creates an instance of the servlet.
Typically, only a single isntance of the servlet is created, and concurrent requests to the servlet are executed on the same servlet instance. This is really up to the servlet container to decide, though. But typically, there is just one instance.

3. Call the Servlets init() Method

When a servlet instance is created, its init() method is invoked. The init() method allows a servlet to initialize itself before the first request is processed.
You can specify init parameters to the servlet in the web.xml file.

4. Call the Servlets service() Method

For every request received to the servlet, the servlets service() method is called. For HttpServlet subclasses, one of the doGet(), doPost() etc. methods are typically called.
As long as the servlet is active in the servlet container, the service() method can be called. Thus, this step in the life cycle can be executed multiple times.

5. Call the Servlets destroy() Method

When a servlet is unloaded by the servlet container, its destroy() method is called. This step is only executed once, since a servlet is only unloaded once.
A servlet is unloaded by the container if the container shuts down, or if the container reloads the whole web application at runtime.

❖ JAVA Servlet Example

A Java Servlet is just an ordinary Java class which implements the interface

```
javax.servlet.Servlet;
```

The easiest way to implement this interface is to extend either the class GenericServlet or HttpServlet.

```
import javax.servlet.GenericServlet;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import java.io.IOException;


public class SimpleServlet extends GenericServlet {

  public void service(ServletRequest request, ServletResponse response)
        throws ServletException, IOException {

       // do something in here
  }
}
```

When an HTTP request arrives at the web server, targeted for your Servlet, the web server calls your Servlet's service() method.
The service() method then reads the request, and generates a response which is sent back to the client (e.g. a browser).
Here is an example service() implementation:

```
public void service(ServletRequest request, ServletResponse response)
        throws ServletException, IOException {

  String yesOrNoParam = request.getParameter("param");

  if("yes".equals(yesOrNoParam) ){

      response.getWriter().write(
        "<html><body>You said yes!</body></html>");
  }

  if("no".equals(yesOrNoParam) ){

      response.getWriter().write(
        "<html><body>You said no!</body></html>");
  }
}
```

This service() method first reads the request parameter "param". Then it checks if the param is equal to the text "yes" or "no", and writes an HTML response back to the browser.

❖ HttpServlet

The javax.servlet.http.HttpServlet class is a slightly more advanced base class than the GenericServlet shown in the Simple Servlet example.
The HttpServlet class reads the HTTP request, and determines if the request is an HTTP GET, POST, PUT, DELETE, HEAD etc. and calls one the corresponding method.
To respond to e.g. HTTP GET requests only, you will extend the HttpServlet class, and override the doGet() method only. Here is an example:

```
public class SimpleHttpServlet extends HttpServlet {

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

      response.getWriter().write("<html><body>GET response</body></html>");
  }
}
```

The HttpServlet class has methods you can override for each HTTP method (GET, POST etc.). Here is a list of the methods you can override:

- doGet()
- doPost()
- doHead()
- doPut()
- doDelete()
- doOptions()
- doTrace()

Most often you just want to respond to either HTTP GET or POST requests, so you just override these two methods.
If you want to handle both GET and POST request from a given servlet, you can override both methods, and have one call the other. Here is how:

```
public class SimpleHttpServlet extends HttpServlet {

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

      doPost(request, response);
  }

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

      response.getWriter().write("GET/POST response");
    }
}
```

I would recommend you to use the HttpServlet instead of the GenericServlet whenever possible. HttpServlet is easier to work with, and has more convenience methods than GenericServlet.

❖ HttpRequest

The HttpServlet class request processing methods take two parameters.

1. javax.servlet.http.HttpRequest
2. javax.servlet.http.HttpResponse

For instance, here is the signature of the HttpServlet.doGet() method:

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

}
```

The purpose of the HttpRequest object is to represent the HTTP request a browser sends to your web application. Thus, anything the browser may send, is accessible via the HttpRequest.

➢ Parameters

The request parameters are parameters that are sent from the browser along with the request. Request parameters are typically sent as part of the URL (in the "query string"), or as part of the body of an HTTP request. For instance:

```
http://jenkov.com/somePage.html?param1=hello¶m2=world
```

Notice the "query string" part of the URL: ?param1=hello¶m2=world This part contains two parameters with parameter values:

```
param1=hello
param2=world
```

You can access these parameters from the HttpRequest object like this:

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

    String param1 = request.getParameter("param1");
        String param2 = request.getParameter("param2");

}
```

In general, if the browser sends an HTTP GET request, the parameters are included in the query string in the URL. If the browser sends an HTTP POST request, the parameters are included in the body part of the HTTP request.

➢ Headers

The request headers are name, value pairs sent by the browser along with the HTTP request. The request headers contain information about e.g. what browser software is being used, what file types the browser is capable of receiving etc. In short, at lot of meta data around the HTTP request.
You can access the request headers from the HttpRequest object like this:

```
String contentLength = request.getHeader("Content-Length");
```

This example reads the Content-Length header sent by the browser.

The Content-Length header contains the number of bytes sent in the HTTP request body, in case the browser sends an HTTP POST request. If the browser sends an HTTP GET request, the Content-Length header is not used, and the above code will return null.

➢ InputStream

If the browser sends an HTTP POST request, request parameters and other potential data is sent to the server in the HTTP request body. It doesn't have to be request parameters that is sent in the HTTP request body. It could be pretty much any data, like a file or a SOAP request (web service request).
To give you access to the request body of an HTTP POST request, you can obtain an InputStream pointing to the HTTP request body. Here is how it is done:

```
InputStream requestBodyInput = request.getInputStream();
```

You will have to call this method before calling any getParameter() method, because calling the getParameter() method on an HTTP POST request will cause the servlet engine to parse the HTTP request body for parameters. Once parsed, you cannot access the body as a raw stream of bytes anymore.

➢ Session

It is possible to obtain the session object from the HttpRequest object too.

The session object can hold information about a given user, between requests. So, if you set an object into the session object during one request, it will be available for you to read during any subsequent requests within the same session time scope.
Here is how you access the session object from the HttpRequest object:

```
HttpSession session = request.getSession();
```

➢ ServletContext

You can access the ServletContext object from the HttpRequest object too. The ServletContext contains meta information about the web application. For instance, you can access context parameters set in the web.xml file, you can forward the request to other servlets, and you can store application wide parameters in the ServletContext too.
Here is how you access the ServletContext object from the HttpRequest object:

```
ServletContext context = request.getSession().getServletContext();
```

❖ HttpResponse

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

}
```

The purpose of the HttpResponse object is to represent the HTTP response your web application sends back to the browser, in response to the HTTP request the browser send to your web application.

Writing HTML

To send HTML back to the browser, you have to obtain the a PrintWriter from the HttpResponse object. Here is how:

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

writer.write("<html><body>GET/POST response</body></html>");
```

Headers

Just like the request object, the HttpRequest can contain HTTP headers. Headers must be set before any data is written to the response. You set a header on the response object like this:

```
response.setHeader("Header-Name", "Header Value");
```

Content-Type

The Content-Type header is a response header that tells the browser the type of the content you are sending back to it. For instance, the content type for HTML is text/html. Similarly, if what you send back to the browser is plain text, you use the content type text/plain.

```
response.setHeader("Content-Type", "text/html");
```

Writing Text

```
response.setHeader("Content-Type", "text/plain");

PrintWriter writer = response.getWriter();
writer.write("This is just plain text");
```

First the Content-Type header is set to text/plain. Then a plain text string is written to the writer obtained from the response object.

Content-Length

The Content-Length header tells the browser how many bytes your servlet is sending back. If you are sending binary data back you need to set the content length header. Here is how:

```
response.setHeader("Content-Length", "31642");
```

➢ Writing Binary Data

You can also write binary data back to the browser instead of text. For instance, you can send an image back, a PDF file or a Flash file or something like that.
Again, you will first have to set the Content-Type header to the type matching the data you are sending back. For instance, the content type for a PNG image is image/png.
In order to write binary data back to the browser you cannot use the Writer obtained from response.getWriter(). After all, Writer's are intended for text.
Instead you have to use the OutputStream obtained from the response.getOutputStream() method. Here is how:

```
OutputStream outputStream = response.getOutputStream();

outputStream.write(...);
```

➢ Redirecting to a Different URL

You can redirect the browser to a different URL from your servlet. You cannot send any data back to the browser when redirecting. Here is how you redirect:

```
response.sendRedirect("http://google.com");
```

❖ HttpSession

The HttpSession object represents a user session. A user session contains information about the user across multiple HTTP requests.
When a user enters your site for the first time, the user is given a unique ID to identify his session by. This ID is typically stored in a cookie or in a request parameter.

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

    HttpSession session = request.getSession();
}
```

You can store values in the session object, and retrieve them later. First, let's see how you can store values in the session object:

```
session.setAttribute("userName", "theUserName");
```

This code sets an attribute named "userName", with the value "theUserName".

To read the value again, you do this:

```
String userName = (String) session.getAttribute("userName");
```

➢ Sessions and Clusters

If you have an architecture with 2 web servers in a cluster, keep in mind that values stored in the session object of one server, may not be available in the session object on the other server. So, if a user's requests are divided evenly between the two servers, sometimes session values may be missing.
The solution to this problem would be one of:

1. Do not use session attributes.
2. Use a session database, into which session attributes are written, and from which it is read.
3. Use sticky session, where a user is always sent to the same server, throughout the whole session.

❖ RequestDispatcher

The RequestDispatcher class enables your servlet to "call" another servlet from inside another servlet. The other servlet is called as if an HTTP request was sent to it by a browser.
You can obtain a RequestDispatcher from the HttpServletRequest object, like this:

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

  RequestDispatcher requestDispatcher =
    request.getRequestDispatcher("/anotherURL.simple");
}
```

The above code obtains a RequestDispatcher targeted at whatever Servlet (or JSP) that is mapped to the URL /anotherUrl.simple.

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

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

By calling either the include() or forward() method the servlet container activates whatever Servlet is mapped to the URL the RequestDispatcher.
The activated servlet has access to the same request as the servlet calling it, and will write to the same response as your current servlet. That way you can merge the output of servlets into a single repsonse.
The forward() method intended for use in forwarding the request, meaning after the response of the calling servlet has been committed. You cannot merge response output using this method.
The include() method merges the response written by the calling servlet, and the activated servlet. This way you can achieve "server side includes" using the include().

❖ ServletContext

The ServletContext is an object that contains meta informaton about your web application. You can access it via the HttpRequest object, like this:

```
ServletContext context = request.getSession().getServletContext();
```

➢ Context Attributes

Just like in the session object you can store attributes in the servlet context. Here is how:

```
context.setAttribute("someValue", "aValue");
```

You can access the attributes again like this:

```
Object attribute = context.getAttribute("someValue");
```

The attributes stored in the ServletContext are available to all servlets in your application, and between requests and sessions. That means, that the attributes are available to all visitors of the web application. Session attributes are just available to a single user.

❖ Web.xml Servlet Configuration

For a Java servlet to be accessible from a browser, you must tell the servlet container what servlets to deploy, and what URL's to map the servlets to. This is done in the web.xml file of your Java web application.

➢ Configuring and Mapping a Servlet

To configure a servlet in the web.xml file, you write this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <servlet>
    <servlet-name>controlServlet</servlet-name>
    <servlet-class>com.jenkov.butterfly.ControlServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>controlServlet</servlet-name>
    <url-pattern>*.html</url-pattern>
  </servlet-mapping>
</web-app>
```

First you configure the servlet. This is done using the <servlet> element. Here you give the servlet a name, and writes the class name of the servlet.
Second, you map the servlet to a URL or URL pattern. This is done in the <servlet-mapping> element. In the above example, all URL's ending in .html are sent to the servlet.

Other possible servlet URL mappings are:

```
/myServlet

/myServlet.do

/myServlet*
```

The * is a wild card, meaning any text. As you can see, you can either map a servlet to a single, specific URL, or to a pattern of URL's, using a wild card (*). What you will use depends on what the servlet does.

➢ Servlet Init Parameters

You can pass parameters to a servlet from the web.xml file. The init parameters of a servlet can only be accessed by that servlet. Here is how you configure them in the web.xml file:

```
<servlet>
    <servlet-name>controlServlet</servlet-name>
    <servlet-class>com.jenkov.butterfly.ControlServlet</servlet-class>

        <init-param>
        <param-name>myParam</param-name>
        <param-value>paramValue</param-value>
        </init-param>
</servlet>
```

Here is how you read the init parameters from inside your servlet - in the servlets init() method:

```
public class SimpleServlet extends GenericServlet {

  protected String myParam = null;

  public void init(ServletConfig servletConfig) throws ServletException{
    this.myParam = servletConfig.getInitParameter("myParam");
  }

  public void service(ServletRequest request, ServletResponse response)
        throws ServletException, IOException {

    response.getWriter().write("<html><body>myParam = " +
            this.myParam + "</body></html>");
  }
}
```

A servlets init() method is called when the servlet container loads the servlet for the first time. No one can access the servlet until the servlet has been loaded, and the init() method has been called successfully.

➢ Servlet Load-on-Startup

The <servlet> element has a subelement called <load-on-startup> which you can use to control when the servlet container should load the servlet. If you do not specify a

<load-on-startup> element, the servlet container will typically load your servlet when the first request arrives for it.
By setting a <load-on-startup> element, you can tell the servlet container to load the servlet as soon as the servlet container starts. Remember, the servlets init() method is called when the servlet is loaded.
Here is an example <load-on-startup> configuration:

```
<servlet>
    <servlet-name>controlServlet</servlet-name>
    <servlet-class>com.jenkov.webui.ControlServlet</servlet-class>
    <init-param><param-name>container.script.static</param-name>
                <param-value>/WEB-INF/container.script</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

➢ Context Parameters

You can also set some context parameters which can be read from all servlets in your application. Here is how you configure a context parameter:

```
<context-param>
    <param-name>myParam</param-name>
    <param-value>the value</param-value>
</context-param>
```

Here is how you access the parameter from inside an HttpServlet subclass:

```
String myContextParam =
        request.getSession()
               .getServletContext()
               .getInitParameter("myParam");
```

❖ Cookies and Servlets

HTTP Cookies are little pieces of data that a web application can store on the client machine of users visiting the web application. Typically up to 4 kilo bytes of data.

➢ Java Cookie Example

You can write cookies using the HttpServletResponse object like this:

```
Cookie cookie = new Cookie("myCookie", "myCookieValue");

response.addCookie(cookie);
```

As you can see, the cookie is identified by a name, "myCookie", and has a value, "myCookieValue". Thus, you can add many different cookies with different identifies (names). It's a bit like a Hashtable.
Whenever the the browser accesses the web application it submits the cookies stored on the client machine to the web application. Only cookies stored by the accessed web application are submitted.

- ## Reading Cookies Sent From the Browser

You can read the cookies via the HttpServletRequest like this:

```
Cookie[] cookies = request.getCookies();
```

Note: the getCookies() method may return null!

Now you can iterate through the array of cookies and find the cookies you need. Unfortunately there is no way to obtain a cookie with a specific name. The only way to find that cookie again is to iterate the Cookie[] array and check each cookie name. Here is an example:

```
Cookie[] cookies = request.getCookies();

String userId = null;
for(Cookie cookie : cookies){
    if("uid".equals(cookie.getName())){
        userId = cookie.getValue();
    }
}
```

- ## Cookie Expiration

One important Cookie setting is the cookie expiration time. This time tells the browser receiving the cookie how long time it should keep the cookie before deleting it.
You set the cookie expiration time via the setMaxAge() method. This method takes the number of seconds the cookie is to live as parameter. Here is an example:

```
Cookie cookie = new Cookie("uid", "123");

cookie.setMaxAge(24 * 60 * 60);  // 24 hours.

response.addCookie(cookie);
```

- ## Removing Cookies

Sometimes you may want to remove a cookie from the browser. You do so by setting the cookie expiration time. You can set the expiration time to 0 or -1. If you set the expiration time to 0 the cookie will be removed immediately from the browser. If you set the expiration time to -1 the cookie will be deleted when the browser shuts down. Here is an example:

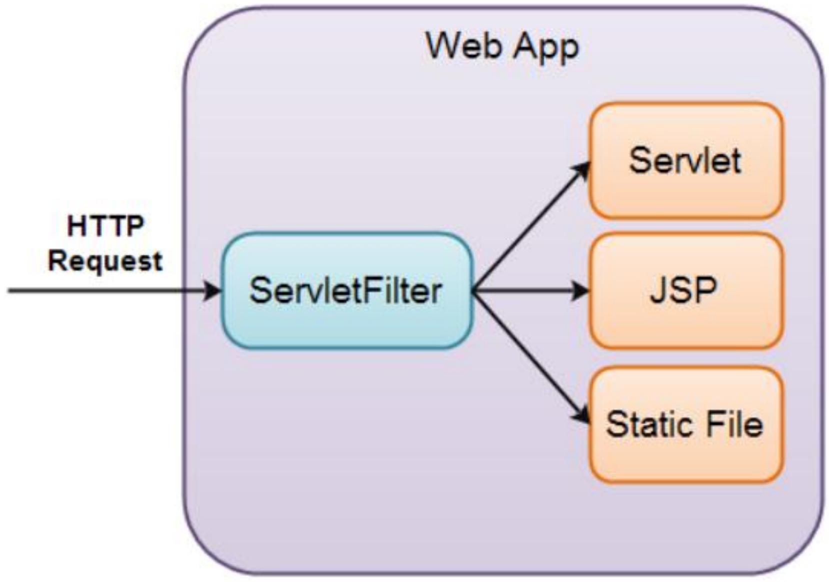
```
Cookie cookie = new Cookie("uid", "");

cookie.setMaxAge(0);

response.addCookie(cookie);
```

- ## Cookie Use Cases

Cookies are most often used to store user specific information, like e.g. a unique user ID (for anonymous users which do not login), a session ID, or user specific setttings you do not want to store in your web applications database (if it has one).

❖ Servlet Filters

A Servlet filter is an object that can intercept HTTP requests targeted at your web application.
A servlet filter can intercept requests both for servlets, JSP's, HTML files or other static content, as illustrated in the diagram below:



A Servlet Filter in a Java Web Application

In order to create a servlet filter you must implement the javax.servlet.Filter interface. Here is an example servlet filter implementation:

```
import javax.servlet.*;
import java.io.IOException;

/**

 */
public class SimpleServletFilter implements Filter {

    public void init(FilterConfig filterConfig) throws ServletException {
    }

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

    }

    public void destroy() {
    }
}
```

When the servlet filter is loaded the first time, its init() method is called, just like with

servlets.
When a HTTP request arrives at your web application which the filter intercepts, the filter can inspect the request URI, the request parameters and the request headers, and based on that decide if it wants to block or forward the request to the target servlet, JSP etc.
It is the doFilter() method that does the interception. Here is a sample implementation:

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

    String myParam = request.getParameter("myParam");

    if(!"blockTheRequest".equals(myParam)){
        filterChain.doFilter(request, response);
    }
}
```

Notice how the doFilter() method checks a request parameter, myParam, to see if it equals the string "blockTheRequest". If not, the request is forwarded to the target of the request, by calling the filterChain.doFilter() method. If this method is not called, the request is not forwarded, but just blocked.
The servlet filter above just ignores the request if the request parameter myParam equals "blockTheRequest". You can also write a different response back to the browser. Just use the ServletResponse object to do so, just like you would inside a servlet.

➢ Configuring the Servlet Filter in web.xml

You need to configure the servlet filter in the web.xml file of your web application, before it works. Here is how you do that:

```
<filter>
    <filter-name>myFilter</filter-name>
    <filter-class>servlets.SimpleServletFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>myFilter</filter-name>
    <url-pattern>*.simple</url-pattern>
</filter-mapping>
```

❖ GZip Servlet Filters

A GZip Servlet Filter can be used to GZip compress content sent to a browser from a Java web application. This text will explain how that works, and contains a GZip Servlet Filter you can use in your own Java web applications.

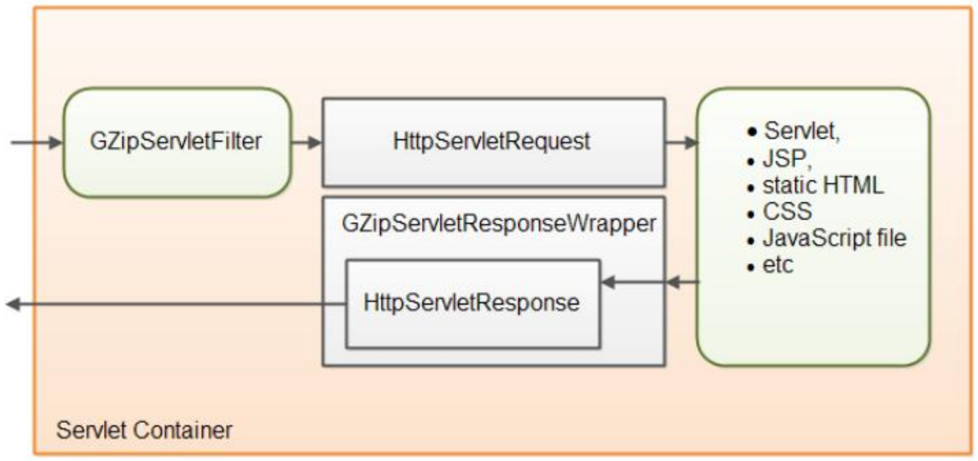
➢ Why GZip Compress Content?

GZip compressing HTML, JavaScript, CSS etc. makes the data sent to the browser smaller. This speeds up the download. This is especially beneficial for mobile phones where internet bandwidth may be limited. GZip compressing content adds a CPU overhead on the server and browser, but it is still speeding up the total page load compared to not GZip compressing.

- ## Why a GZip Servlet Filter?

The smart thing about a GZip Servlet filter is that it is executed before and after any Servlet, JSP, or even static files. That way you can create a single servlet filter that enables GZip compression for all content that needs it. The Servlets, JSPs etc. don't even know that the content is being compressed, because it happens in the Servlet filter. The GZip Servlet filter enables GZip compression, sets the right HTTP headers, and makes sure that content written by Servlets, JSPs etc. is compressed.

- ## GZip Servlet Filter Design



The GZip Servlet Filter design.

First you need a Servlet filter class. That class is mapped to a set of URL's in the web.xml file.
When an HTTP request arrives at the Servlet container which is mapped to the filter, the filter intercepts the request before it is handled by the Servlet, JSP etc. which the request is targeted at. The GZip servlet filter checks if the client (browser) can accept GZip compressed content. If yes, it enables compression of the response.
GZip compression of the response is enabled by wrapping the HttpServletResponse object in a GZipServletResponseWrapper. This wrapper is passed to the Servlet, JSP etc. which handles the request. When the Servlet, JSP etc. writes output to be sent to the browser, it does so to the response wrapper object. The Servlet, JSP etc.

- ## GZip Servlet Filter Code

The code consists of 3 classes. A GZipServletFilter, a GZipServletResponseWrapper and a GZipServletOutputStream.
The GZipServletOutputStream is what compresses the content written to it. It does so by using a GZIPOutputStream internally, which is a standard Java class.
When the GZipServletResponseWrapper gives back an OutputStream or PrintWriter to a Servlet or JSP, it is either a GZipServletOutputStream or a PrintWriter that writes to the GZipServletOutputStream which is returned.
The GZipServletFilter is what intercepts the requests, checks if the client accepts compression or not, and enables compression if it does. It does so by wrapping the HttpServletResponse in a GZipServletResponseWrapper before passing it down the

filter chain.

```
public class GZipServletFilter implements Filter {

  @Override
  public void init(FilterConfig filterConfig) throws ServletException {
  }

  @Override
  public void destroy() {
  }

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

    HttpServletRequest  httpRequest  = (HttpServletRequest)  request;
    HttpServletResponse httpResponse = (HttpServletResponse) response;

    if ( acceptsGZipEncoding(httpRequest) ) {
      httpResponse.addHeader("Content-Encoding", "gzip");
      GZipServletResponseWrapper gzipResponse =
        new GZipServletResponseWrapper(httpResponse);
      chain.doFilter(request, gzipResponse);
      gzipResponse.close();
    } else {
      chain.doFilter(request, response);
    }
  }

  private boolean acceptsGZipEncoding(HttpServletRequest httpRequest) {
      String acceptEncoding =
        httpRequest.getHeader("Accept-Encoding");

      return acceptEncoding != null &&
             acceptEncoding.indexOf("gzip") != -1;
  }
}
```

```
class GZipServletResponseWrapper extends HttpServletResponseWrapper {

  private GZipServletOutputStream gzipOutputStream = null;
  private PrintWriter             printWriter      = null;

  public GZipServletResponseWrapper(HttpServletResponse response)
          throws IOException {
      super(response);
  }

  public void close() throws IOException {

      //PrintWriter.close does not throw exceptions.
      //Hence no try-catch block.
      if (this.printWriter != null) {
          this.printWriter.close();
      }

      if (this.gzipOutputStream != null) {
          this.gzipOutputStream.close();
      }
  }
```

```
/**
 * Flush OutputStream or PrintWriter
 *
 * @throws IOException
 */

@Override
public void flushBuffer() throws IOException {

  //PrintWriter.flush() does not throw exception
  if(this.printWriter != null) {
    this.printWriter.flush();
  }

  IOException exception1 = null;
  try{
    if(this.gzipOutputStream != null) {
      this.gzipOutputStream.flush();
    }
  } catch(IOException e) {
      exception1 = e;
  }

  IOException exception2 = null;
  try {
    super.flushBuffer();
  } catch(IOException e){
    exception2 = e;
  }

  if(exception1 != null) throw exception1;
  if(exception2 != null) throw exception2;
}

@Override
public ServletOutputStream getOutputStream() throws IOException {
  if (this.printWriter != null) {
    throw new IllegalStateException(
      "PrintWriter obtained already - cannot get OutputStream");
  }
  if (this.gzipOutputStream == null) {
    this.gzipOutputStream = new GZipServletOutputStream(
      getResponse().getOutputStream());
  }
  return this.gzipOutputStream;
}
```

```
  @Override
  public PrintWriter getWriter() throws IOException {
     if (this.printWriter == null && this.gzipOutputStream != null) {
       throw new IllegalStateException(
         "OutputStream obtained already - cannot get PrintWriter");
     }
     if (this.printWriter == null) {
       this.gzipOutputStream = new GZipServletOutputStream(
         getResponse().getOutputStream());
       this.printWriter      = new PrintWriter(new OutputStreamWriter(
       this.gzipOutputStream, getResponse().getCharacterEncoding()));
     }
     return this.printWriter;
  }


  @Override
  public void setContentLength(int len) {
    //ignore, since content length of zipped content
    //does not match content length of unzipped content.
  }
}
```

```
class GZipServletOutputStream extends ServletOutputStream {
  private GZIPOutputStream    gzipOutputStream = null;

  public GZipServletOutputStream(OutputStream output)
        throws IOException {
    super();
    this.gzipOutputStream = new GZIPOutputStream(output);
  }

  @Override
  public void close() throws IOException {
    this.gzipOutputStream.close();
  }

  @Override
  public void flush() throws IOException {
    this.gzipOutputStream.flush();
  }

  @Override
  public void write(byte b[]) throws IOException {
    this.gzipOutputStream.write(b);
  }


  @Override
  public void write(byte b[], int off, int len) throws IOException {
    this.gzipOutputStream.write(b, off, len);
  }

  @Override
  public void write(int b) throws IOException {
     this.gzipOutputStream.write(b);
  }
}
```

➢ GZip Servlet Filter web.xml Configuration

In order to activate the GZip Servlet filter in your Java web application, you need the configuration below. Remember to replace the class name with the fully qualified name of your own GZip Servlet filter class. The filter mappings determine what

HTTP requests the filter is activated for.

```
<filter>
  <filter-name>GzipFilter</filter-name>
  <filter-class>com.myapp.GZipServletFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>GzipFilter</filter-name>
  <url-pattern>*.js</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>GzipFilter</filter-name>
  <url-pattern>*.css</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>GzipFilter</filter-name>
  <url-pattern>*.html</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>GzipFilter</filter-name>
  <url-pattern>*.jsp</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name>GzipFilter</filter-name>
  <url-pattern>/</url-pattern>
</filter-mapping>
```

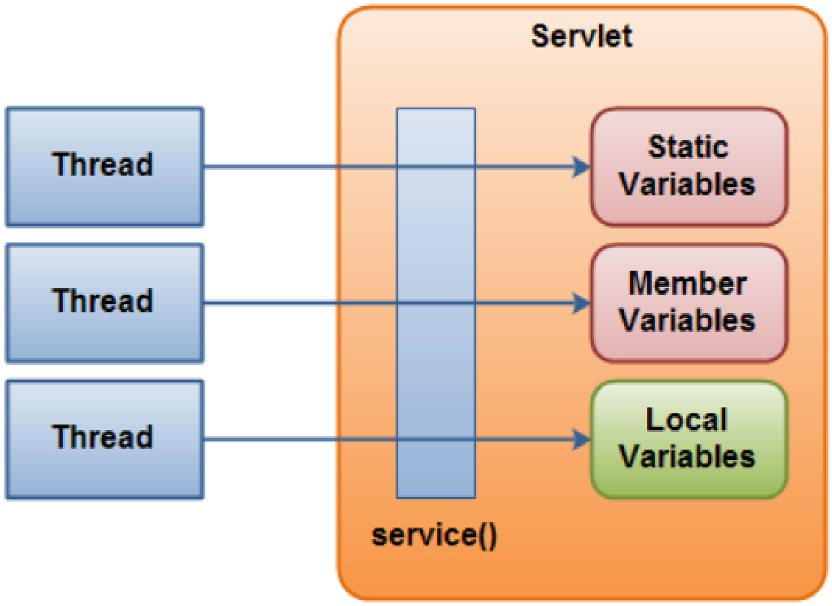
❖ Servlet Concurrency

A Java servlet container / web server is typically multithreaded. That means, that multiple requests to the same servlet may be executed at the same time. Therefore, you need to take concurrency into consideration when you implement your servlet.
To make sure that a servlet is thread safe, there are a few basic rules of thumb you must follow:

1. Your servlet service() method should not access any member variables, unless these member variables are thread safe themselves.
2. Your servlet service() should not reassign member variables, as this may affect other threads executing inside the service() method. If you really, really need to reassign a member variable, make sure this is done inside a synchronized block.
3. Rule 1 and 2 also counts for static variables.
4. Local variables are always thread safe. Keep in mind though, that the object a local variable points to, may not be so. If the object was instantiated inside the method, and never escapes, there will be no problem. On the other hand, a local variable pointing to some shared object, may still cause problems. Just because you assign a shared object to a local reference, does not mean that object automatically becomes thread safe.

The request and response objects are of course thread safe to use. A new instance of these are created for every request into your servlet, and thus for every thread executing in your servlet.
Here is a diagram which illustrates the servlet concurrency rules / issues mentioned above. The red boxes represent state (variables) that your servlet's service() method

should be careful about accessing.



Java Servlet Concurrency Rules.

➢ Other Shared Resources

Of course it is not only the member variables and static variables inside the servlet class itself, that you need to be careful about accessing. Static variables in any other class which are accessed by your servlet, must also be thread safe. The same is true for member variables of any shread objects accessed by your servlet.

➢ Code Example

```
public class SimpleHttpServlet extends HttpServlet {

  // Not thread safe, static.
  protected static List list = new ArrayList();

  // Not thread safe
  protected Map map = new HashMap();

  // Thread safe to access object, not thread safe to reassign variable.
  protected Map map = new ConcurrentHashMap();

  // Thread safe to access object (immutable), not thread safe to reassign variab
  protected String aString = "a string value";
  protected void doGet( HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {


    // Not thread safe, unless the singleton is 100% thread safe.
    SomeClass.getSomeStaticSingleton();


    // Thread safe, locally instantiated, and never escapes method.
    Set set = new HashSet();

  }
}
```