Java EE

- Inter-servlet communication
  - Servlet Redirection (sendRedirect() method)
  - Request Dispatcher
  - *include()* and *forward()* methods
- Session Handling

Inter-Servlet Communication

Inter-Servlet Communication is possible through:

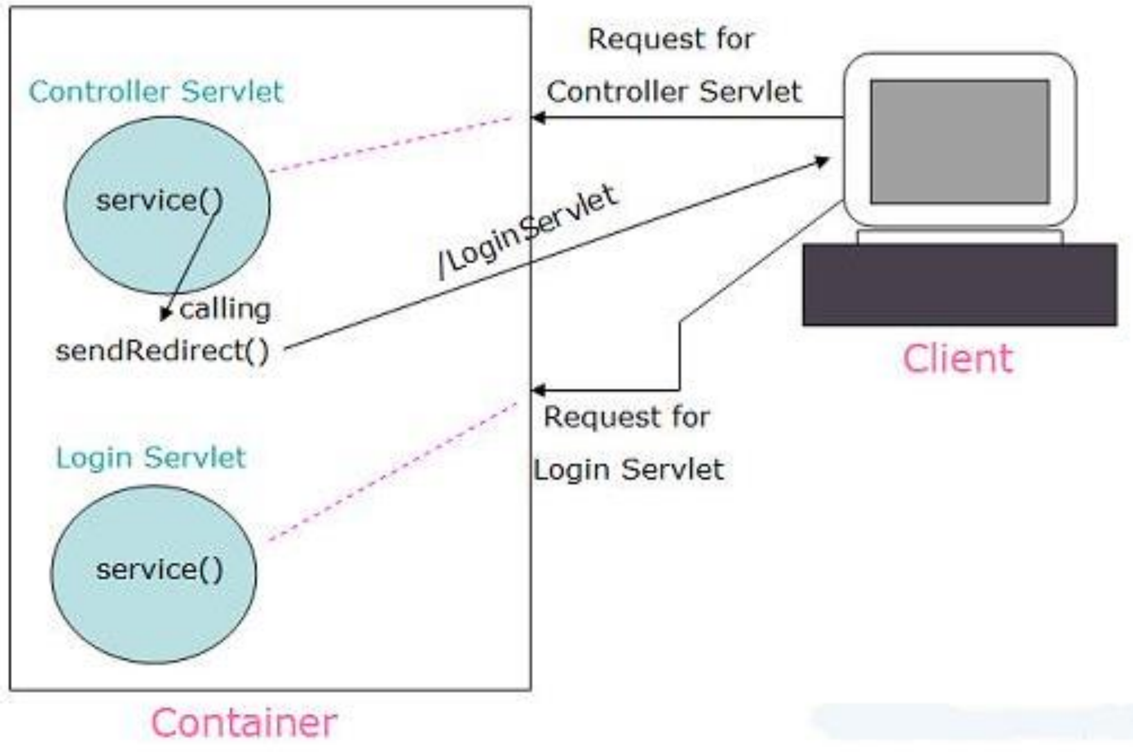
- **Request Dispatcher - direct invocation of a web resource calling include() or forward() methods of RequestDispatcher**
- **Servlet Redirection - indirect invocation of web resource using sendRedirect() method of HttpServletResponse**

RequestDispatcher's include()/forward() method transfers the control of the request to another servlet or jsp without informing anything about this request dispatch to the client browser. Therefore client browser has no knowledge about the returned resource is from another servlet/jsp. **Only one request/response cycle is required.**

sendRedirect ()method stops further processing of the request and sends http status code "302" and URL of the location to be redirected **to the client browser** in the response header.

Client browser looks at http status 302 and then it knows that it should send a new request to the url in "Location“ specified in the http header which is set by server and Client browser sends a new request to the new URL and it will be processed by the server as another normal request.
In this case, Two request/response cycles are required.

SendRedirect Example



```
@WebServlet("/Controller")
public class ControllerServlet extends HttpServlet {
                ................
               // read parameters
                if(valid_credentials){
                  response.sendRedirect("LoginServlet");
                }
                else{
response.sendError(HttpServletResponse.SC_NOT_FOUND, "Invalid
Credentials");    }
.......
}
```

Observe URI

```
@WebServlet("/LoginServlet")
public class LoginServlet extends HttpServlet {
@Override
protected void service(HttpServletRequest request,
HttpServletResponse response)
                                        throws ServletException, IOException{
                            PrintWriter out = response.getWriter();
                            out.println("Welcome to  Web Application");
              }
}
```

login.html

```
<form action= "Controller" method="POST">
      <p>Enter user name <input type="text"
      name="username"  size="20"></p>
      <p>Enter password <input type="password"
      name= " password"  size="20"></p>
      <p><input type="submit" value="Submit"> </p>
</form>
```

RequestDispatcher

The **javax.servlet.RequestDispatcher** interface of Java Servlet API provides methods to dispatch the request from one Web component to another Web resource in the context.

Dispatching the Request:

1. Getting a **RequestDispatcher** object
2. Using **include()** or **forward()** methods of RequestDispatcher

Getting a RequestDispatcher object:

The RequestDispatcher object can be obtained by using the following methods:

- The **getRequestDispatcher(java.lang.String path)** method of **ServletContext**
- The **getRequestDispatcher(java.lang.String path)** method of **ServletRequest**

```
RequestDispatcher requestDispatcher = getServletContext().getRequestDispatcher("url");
```

or

```
RequestDispatcher requestDispatcher = request.getRequestDispatcher("url");
```

Dispatching request/response objects:

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

or

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

Relative Path Vs Absolute Path

The getRequestDispatcher() method of ServletContext

The getRequestDispatcher() method of ServletContext takes a String argument describing the path to locate the resource to which the request is to be dispatched.
When this method is called , the container locates the resource with the given path **where path should start with /**

The getRequestDispatcher() method of ServletRequest

The getRequestDispatcher() method of ServletRequest takes a String argument describing the **path relative to the current request**.

The Servlet container uses the information in the request object to build a complete path and locates the resource using the path.

Note : ServletRequest.getRequestDispatcher takes a relative URL while ServletContext.getRequestDispatcher takes absolute path.

The getRequestDispatcher() Methods

Let us consider an example to demonstrate the use of the getrequestDispatcher() method in different cases.

Servlet Name	URL	URL Pattern
Servlet1	http://localhost:8080/mycntxt/myser1	/myser1
Servlet2	http://localhost:8080/mycntxt/myser2	/myser2
Servlet3	http://localhost:8080/mycntxt/myservlets/myser3	/myservlets/myser3
Servlet4	http://localhost:8080/mycntxt/myservlets/myser4	/myservlets/myser4

Now, to forward the request from the Servlet1 to Servlet4, we can call getRequestDispatcher() method using the following methods:

```
ServletContext context = getServletContext();
RequestDispatcher requestDispatcher= context.getRequestDispatcher("/myservlets/myser4");
                                   or
RequestDispatcher requestDispatcher = request.getRequestDispatcher("myservlets/myser4");
```

How do we forward the request from the Servlet4 to Servlet1 using both the methods ?

SendRedirect method Vs RequestDispatcher

Replace the following statement in the ControllerServlet

```
response.sendRedirect("LoginServlet");
```

with the following statements

```
RequestDispatcher requestDispatcher= getServletContext().getRequestDispatcher("/LoginServlet");
requestDispatcher.forward(request, response);
```

Now observe the URI

include() method

An included web component has access to the request object but is limited in what it can do with the response object.

•It can write to the body of the response.

•It cannot set headers or call any method, such as setCookie(), that affects the headers of the response.

The response generated from the calling servlet is also included to the final response.

The method prototype for include() method is as follows:

```
public void include(ServletRequest request, ServletResponse response)  throws ServletException,   java.io.IOException
```

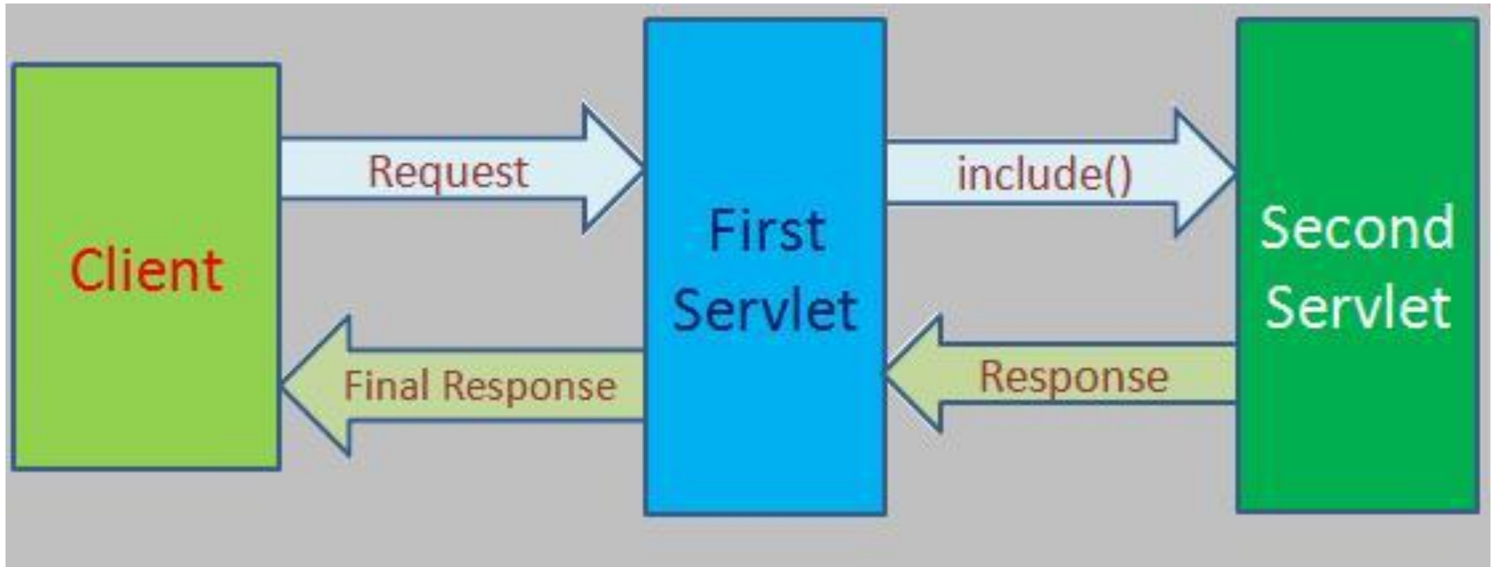


Fig: InterCommunication using include( )

forward() Method

forward(): The forward() method of **RequestDispatcher** interface is responsible for simply forwarding the request from one resource to another.

1. The response generated from the calling servlet is not added to the final response.
2. The forward() method allows only the target Servlet to set the response headers.

public void **forward(ServletRequest request, ServletResponse response) throws ServletException, java.io.IOException**

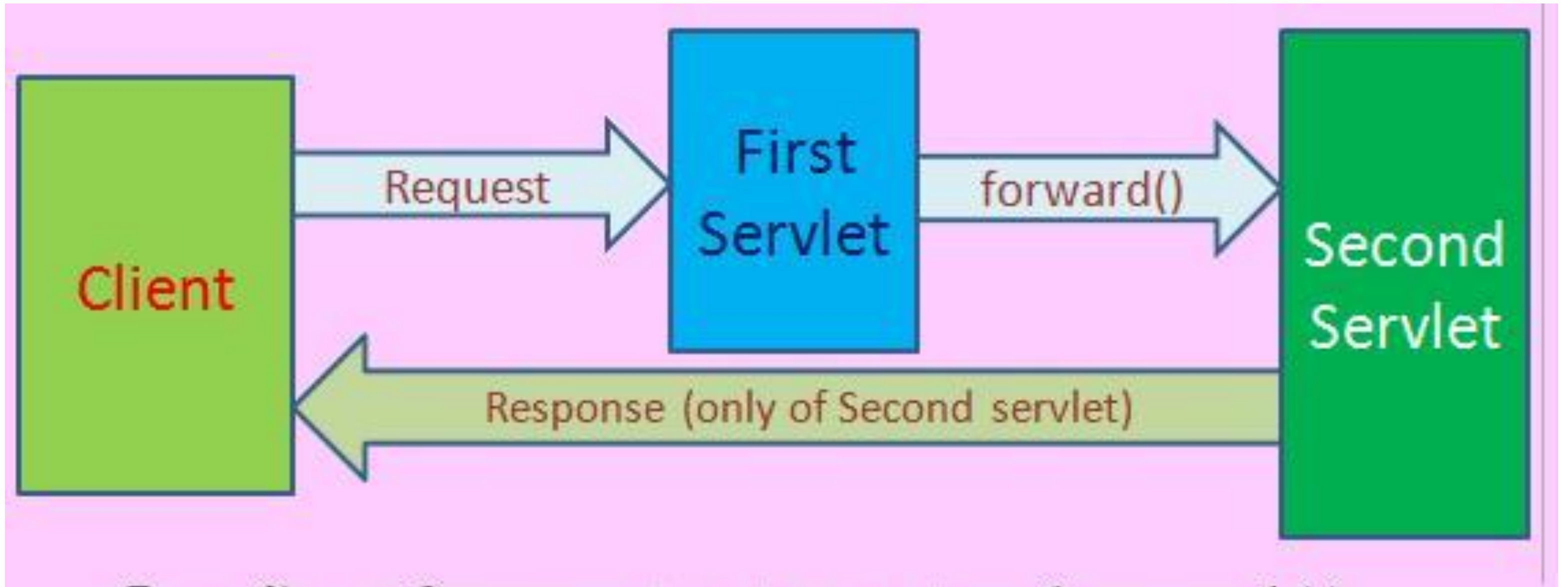


Fig: InterCommunication using forward( )

Comparing include() and forward() Methods

1. The forward() method allows the target Servlet to set the response headers. However, it cannot be done with the include() method.

2. In the forward() method, the source Servlet cannot generate the response content, but it can be done with the include() method.

   Note : In the case of the forward() method, after the target Servlet completes the execution of its service() method and before the control is given back to the source Servlet, the RequestDispatcher object ***closes the response stream.***

   Due to this reason, the source servlet data is not added to the response buffer.

Java EE

Sharing data between web components

- Request and Application object attributes

Session Handling

Using Scope Objects

- There are four scope objects in servlets & JSPs which enables sharing information between web components.

- Collaborating web components share information by means of objects that are maintained as attributes of four scope objects.

- We can access these attributes by using *getAttribute()* and *setAttribute()* methods of the class representing the scope.

The scope objects and their corresponding Java classes are listed below:

Scope Object	Class	Accessible from
Web context/ application	javax.servlet.ServletContext	Web components within a web context/web application.
Session	javax.servlet.http.HttpSession	Web components handling a request that belongs to the session.
Request	Subtype of javax.servlet.ServletRequest	Web components handling the request.
Page	javax.servlet.jsp.PageContext	The JSP page that creates the object.

Object Visibility

Included or forwarded objects do not share Java variables with the originating resource, but they do share attributes.

Thus if you include the following code

object.setAttribute("attributeName", "attributeValue");

Within the doGet method of another web component, we can retrieve the value of that attribute with a call to ***object.getAttribute("username");***

Note: JSP supports the equivalent action via the *jsp:include, jsp:forward, and jsp:param.*

Request Object Attributes

- The Servlet container supports an ability to dispatch requests from one Servlet to another within the context. This is required if you want to make some information available from the source Servlet to the target Servlet or vice-versa.

- The javax.servlet.ServletRequest object manages the data within the request scope to meet this requirement.

- The ServletRequest object has methods to set and get name-value pairs into the request scope. Such name-value pairs are called as attributes.

Methods:

void setAttribute(String,Object)- Sets an attribute with the current request. If the given String name is already bound with an object, it is replaced by the new object.

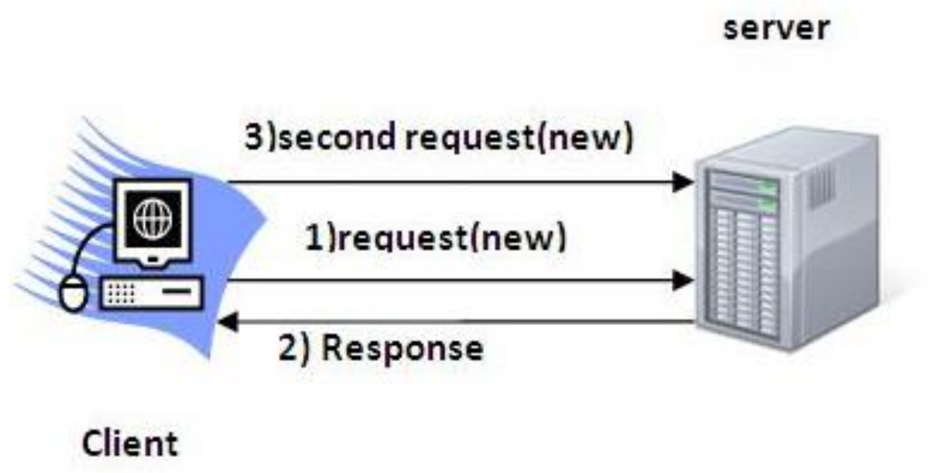
Object getAttribute(String) – Returns the value of an attribute located with the given name or returns null value if an attribute with the given name is not found.
Enumeration getAttributeNames()-Returns all the attribute names in the current request as an **Enumeratio**n of String or returns empty **Enumeration**, if there are no attribute in the request.
void removeAttribute(String) – Removes an attribute by the given name from the request.

SESSION HANDLING

WHAT IS SESSION HANDLING

The protocol used for communication between a browser and the Web server is generally HTTP which is a stateless protocol.

Each time user sends a request to the web server, the server treats the request as a new request. This means that the Web server does not remember the previous interaction with the browser(client).



Session tracking is a way to **maintain state of a HTTP client** (browser) between multiple request-response cycles between the browser and the Server.

Online Web Applications: Reservation Systems, Shopping portals, Bank Transactions etc.

COOKIES

There are three primary ways of implementing session handling in your applications:

1. Session handling using cookies— mostly used technology for session tracking.

- Cookie is a small text file that contains information in form of key value pairs, sent by the server to the browser. The browser saves the cookie in the client's machine.
- So, whenever the browser sends a request to the same server, it sends the cookie along with it. The server can identify the client through this cookie.

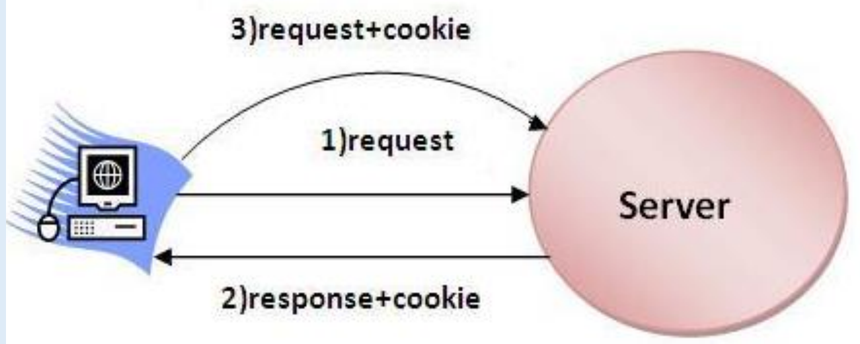
Disadvantage:

Users can opt to disable cookies using their browser preferences. In such case, the browser will not save the cookie at client's machine and session tracking fails.

There are two types of cookies:

Session cookies - are temporary cookie files, which are erased when we close the browser.

Persistent cookies – These files stay in one of our browser's subfolders until we delete them manually or the browser deletes them based on the duration period contained within the persistent cookie's file.



Sending Cookies to the Client

- **Create a Cookie object.**
  - Call the Cookie constructor with a cookie name and a cookie value, both of which are Strings.

    Cookie cookie = new Cookie("name", "value");

    Ex. ***Cookie cookie_name = new Cookie("username“, "Smith");***

- **Add the cookie** to the *Set-Cookie* response header by means of the addCookie method of HttpServletResponse.

  response.addCookie(cookie_name);

  Ex. response.addCookie(cookie);

  Note : If you forget this step, no cookie is sent to the browser!

- Call **request.getCookies() method** that returns an array of Cookie objects in the Servlet.

- Loop down the array, calling ***getName() & getValue()*** on each entry.

Setting age limit for a cookie object

<u>To set the age limit for a cookie object</u>

public void setMaxAge(int expiry)

- Sets the maximum age of the cookie in seconds.
- A ***positive value*** indicates that the cookie will expire after that many seconds have passed.
- A ***negative value*** means that the cookie is not stored persistently and will be deleted when the Web browser exits(session cookie).
- A ***zero value*** causes the cookie to be deleted.

Note: If you don't set this, the cookie will last only for the current session.

URL RE-WRITING

2. URL Rewriting

Original URL: **http://server:port/servlet/ServletName**

Rewritten URL: **http://server:port/servlet/ServletName?jsessionid=DA32242SSGE2**

When a request is made, additional parameter is appended to the ***url***. Generally the added parameter will be ***jsessionid*** or sometimes the ***userid***.

Disadvantage:

We have to be careful that every URL returned to the user (even via indirect means like Location fields in server redirects) has the extra information appended.

And, if the user leaves the session and comes back **via a bookmark or a hyperlink**, the session information can be lost.

Hidden form fields

3. Hidden form fields.

HTML forms have an entry that looks like the following:

<INPUT TYPE="HIDDEN" NAME="session" VALUE="...">.

This means that, when the form is submitted, the specified name and value are included in the GET or POST data. This can be used to store information about the session.

Disadvantage

Works only if every page is dynamically generated,

HttpSession

Session Handling: HttpSession API

State maintenance between the client browser and the servlet is done by the server's setting a **cookie** in the client browser.

But unlike the plain cookie method of session handling, where all the session data is stored in the cookie as name-value strings in the client’s machine, the onus of maintaining the session is undertaken by the Web Server.

Here the web server creates an session object with a unique identifier and stores the session information in it and sends only the session-id to the client and the browser stores the session-id in form of a cookie in the client’s machine.

Since this data is stored on the server side and only the identifier is passed back and forth between the client browser and the web server as a cookie, this method is an example of server-side session handling.

The server maintains a mapping of session identifiers generated for different clients and the data stored for each session.

HttpSession

When the Servlet container creates a session object(by using getSession() method), it creates a unique identity and sets it into the cookie with the name ***jsessionid****.*

However, in some situations, a browser may not accept cookies, which means that session tracking with cookies is not possible.

In such cases, session identity is appended to the URLs by using URL rewriting.

Most Servlet containers use URL rewriting when a session is new, even if the browser accepts cookies, because the server cannot determine during the first visit of a session, whether the browser accepts cookies or not.

If the cookies are **not disabled** by the browser, from the next trip onwards cookie which contains the session-id travels between the server and the client to maintain the state.

In case browser **has disabled cookies**, then **URL rewriting technique** is implemented to maintain the state between the server and the client.

HttpSession

Using sessions in servlet is quite straightforward, and involves :

1. Looking up the session object associated with the current request,
2. creating a new session object when necessary, looking up information associated with a session,
3. storing information in a session, and
4. discarding completed or abandoned sessions.

Creating a session object

getSession() or getSession(true) methods of HttpServletRequest .

When the above methods are called, the Web Container tries to locate the session object and returns it. If unable to locate the session object , it creates a new session and associates it with the client request.

The above methods perform the following operation when an session object is created:

1. Creates a new session object provided there is no existing session object associated with the request.
2. Prepares a unique identity for the new session
3. Sets this identity into a cookie with the name ***jsessionid***
4. Returns a session object

getSession(false)

The web container tries to locate and if found returns the session object, if web container is **unable** to locate the session, **it returns null**.

Storing information in a session

Session objects have a built-in data structure that allows you to store any number of ***key-value pairs***.

public void setAttribute(String name, Object value)
where ***name*** *identifies the* ***object*** *that is bound to the session.*

So the above method binds an object to this session, using the name specified.

Note: The **setAttribute()** method replaces the attribute value if the attribute already exists in the session, i.e. if the given name is already mapped to a value.

In this case, If the given value is null then this method acts similar to **removeAttribute()** method.

Note: If the object bound to an session object is an *HashMap* or *Hashtable*, then it can be used to hold the data in form of key-value pairs between the requests for the same user and session.

Retrieving the attributes:

- **public Object getAttribute(String name)**
- Enumeration **getAttributeNames**()

Removing the attribute from the Session object:

- **public void removeAttribute(String name)**

Attaching the session Identity to the URLs

As we have discussed earlier, if the browser does not accept cookies, we need to append the session identity to the URLs by using **URL rewriting**.

We can use the ***getId()*** method of the HttpSession interface to get the session identity and add it to the path of all the URLs returned to the clients.

Most Servlet containers use URL rewriting when a session is new, even if the browser accepts cookies, because the server cannot determine during the first visit of a session, whether the browser accepts cookies or not.

The **encodeURL()** method determines whether or not the URL needs to be rewritten and , if necessary, includes the session ID in the URL.

```
……
HttpSession session = request.getSession();
out.println("<a href='"+response.encodeURL("nextPage")+"'>");
out.println("Go to Next page </a>");
```

Maximum Inactive Interval Time

By default every web server will have a configuration set for expiry of session objects.

Generally it will be some X seconds of inactivity i.e. when the user has not sent any request to the server for the past X seconds then the session will expire.

When the user sends a request after the session has expired, server will treat it as a new user and creates a new session.

In Tomcat Server: the default maximum inactive interval is 30 minutes.

We can change this by following entry into web.xml.

```
<session-config>
        <session-timeout>10</session-timeout>
</session-config>
```

Note: 10 indicates 10 minutes not seconds

- **setMaxInactiveInterval**(-1) means the session never times out
- **setMaxInactiveInterval(0)** means session times out immediately

We can programmatically set the maximum inactive interval time by calling the method, **setMaxInactiveInterval()** *where argument to this method is in seconds.*

Invalidating a session object

To invalidate(inaccessible) session object:

session.invalidate();

Thank You!