
# Java Technologies

## Servlets

# The Context

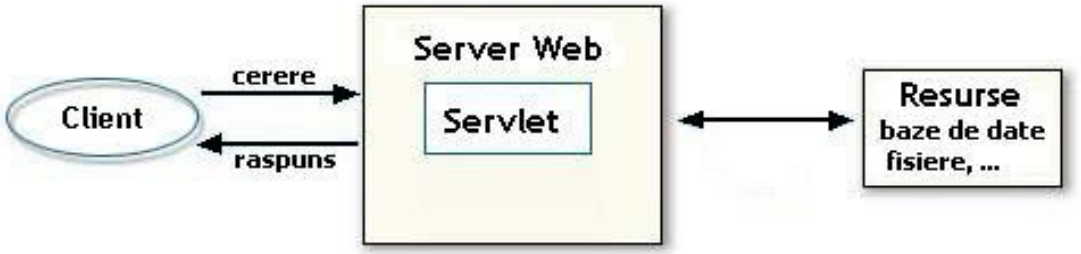
- We are in the context of developing a **distributed** application - *a software system in which components communicate over the network and coordinate their actions in order to achieve a common goal.*
- What is the most common way of communication?

  → *Message Passing*
- What kind of messages are the components passing?

  → Requests / Responses
- *There should be a standard way to implement a component that receives a request and returns a response over a network protocol.*

# What is a Servlet?

- A **servlet** is a component that offers a **service** over the network using a **request-response** programming model.
- At specification level, a servlet can respond to **any type of request.** However, for Web applications Java Servlet technology defines **HTTP-specific** classes.
- In fact, it is a simple way *to extend the capabilities of a Web server*, in order to:
  - generate dynamic content (pages),
  - control various aspects of a Web flow.



# “Nobody” uses servlets anymore...

- Java Server Faces – FacesServlet

  “Manages the request processing lifecycle for web applications that are utilizing JavaServer Faces to construct the user interface.”

- Spring - DispatchServlet

  “Central dispatcher for HTTP request handlers/controllers, e.g. for web UI controllers or HTTP-based remote service exporters. Dispatches to registered handlers for processing a web request, providing convenient mapping and exception handling facilities.”

- Jersy (REST services) – ServletContainer

  “Responsible for deploying root resource classes.”

- Freemarker (template engine)

  “FreeMarker MVC View servlet that can be used similarly to JSP views. That is, you put the variables to expose into HTTP servlet request attributes, then forward to an FTL file (instead of to a JSP file) that's mapped to this servet”

- ...

# The Servlet Interface

Defines methods that all servlets must implement.

```
package javax.servlet;

public interface Servlet {

    public void init(ServletConfig config) throws ServletException;

    public void service(ServletRequest req, ServletResponse res)
        throws ServletException, IOException;

    public void destroy();

    public ServletConfig getServletConfig();

    public String getServletInfo();
}
```

```
public abstract class GenericServlet implements Servlet { … }
```

```
public abstract class HttpServlet extends GenericServlet { … }
```

# The HttpServlet Class

Provides an abstract class to be subclassed in order to create an HTTP servlet .

```
package javax.servlet.http;

public abstract class HttpServlet extends GenericServlet {

  protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
      String method = req.getMethod();
      if (method.equals(METHOD_GET)) {
        doGet(req, resp);
      } else if (method.equals(METHOD_POST)) {
        doPost(req, resp);
      } …
  }

  //Called by the server (via the service method) to allow a servlet to
  //handle a GET request. Override this method to support a GET request.
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    }

  protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException { … }
...
}
```

# *HelloWorld* Servlet

```
package demo;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HelloWorldServlet extends HttpServlet {

  @Override
  public void doGet( HttpServletRequest request,
     HttpServletResponse response ) throws IOException {

        response.setContentType("text/html");

        PrintWriter out = new PrintWriter(response.getWriter());

        out.println("<html><head><title>Hello</title></head>");
        out.println("<body>Hello World at " + new java.util.Date());
        out.println("</body></html>");

        out.close();
    }
}
```

← What is **MIME type**?

# Running the Servlet

- Create a mapping: *class – name – url pattern*
  - Using *@WebServlet* annotation

```
@WebServlet(name = "HelloWorld", urlPatterns = {"/hello"})
public class HelloWorldServlet extends HttpServlet {
...
}
```

What is an URL pattern?

  - Using *web.xml*

```
<servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>demo.HelloWorldServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>HelloWorld</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

Why do we need a servlet name?

- *http://localhost:8080/MyApplication/hello*

# URL-Pattern specification

- **Path mapping**:  strings beginning with a '/' character and ending with a '/*' suffix: /hello/*
- **Extension mapping**: strings beginning with a '*.' prefix is used as an extension mapping: *.hello
- **Default mapping**: a string containing only the '/' character indicates the "default" servlet of the application. In this case the servlet path is the request URI minus the context path and the path info is null.
- **Exact mapping**: all other strings are used for exact matches only

# Sending Request Parameters

```
<html>
<head>
  <title>Register</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>

<body>
  <form method="GET" action="RegisterServlet">

    Name:
    <input type="text" name="name" size="20" value=""/> <br/>

    Email address:
    <input type="text" name="mail" size="20" value=""/> <br/>

    <input type="submit" name="submit" value="Submit">
  </form>
</body>
</html>
```

"action" indicates the Web component that receives the request when the form is submitted; **absolute** or **relative** URL

http://localhost:8080/MyApplication/RegisterServlet? name='John'& mail='john@yahoo.com'

# Receiving Request Parameters

```
@WebServlet(name = "RegisterServlet",urlPatterns = {"/RegisterServlet"})
public class RegisterServlet extends HttpServlet {
  private PrintWriter writer = null;
  @Override public void init() throws ServletException {
    String filename =
      getServletContext().getRealPath("database.txt");
    try {
      writer = new PrintWriter(new FileWriter(filename));
    } catch (IOException e) {
      throw new UnavailableException(e.getMessage());
  }

  public void doGet(HttpServletRequest request,HttpServletResponse response)
      throws IOException {
    String name = request.getParameter("name");
    String mail = request.getParameter("mail");
    writer.println(name + "\t" + mail); writer.flush();

    //send confirmation
    PrintWriter out = new PrintWriter(response.getWriter());
    out.println(...);
  }

  @Override public void destroy() {
    writer.close();
  }
}
```

When do you use POST
and when do you use GET

# Initialization Parameters

**web.xml**

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>demo.HelloWorldServlet</servlet-class>

  <init-param>
    <param-name>message</param-name>
    <param-value>Hello World</param-value>
  </init-param>

</servlet>
```

```
public class HelloWorldServlet extends HttpServlet {

  public void doGet( HttpServletRequest request,
     HttpServletResponse response ) throws IOException {
        ...
        ServletConfig config = getServletConfig();
        String message = config.getInitParameter("message");
        out.println(message);
        ...
    }
}
```

A servlet configuration object used by a servlet container to pass information to a servlet during initialization.

# Using AJAX to Invoke a Servlet

**HEAD**

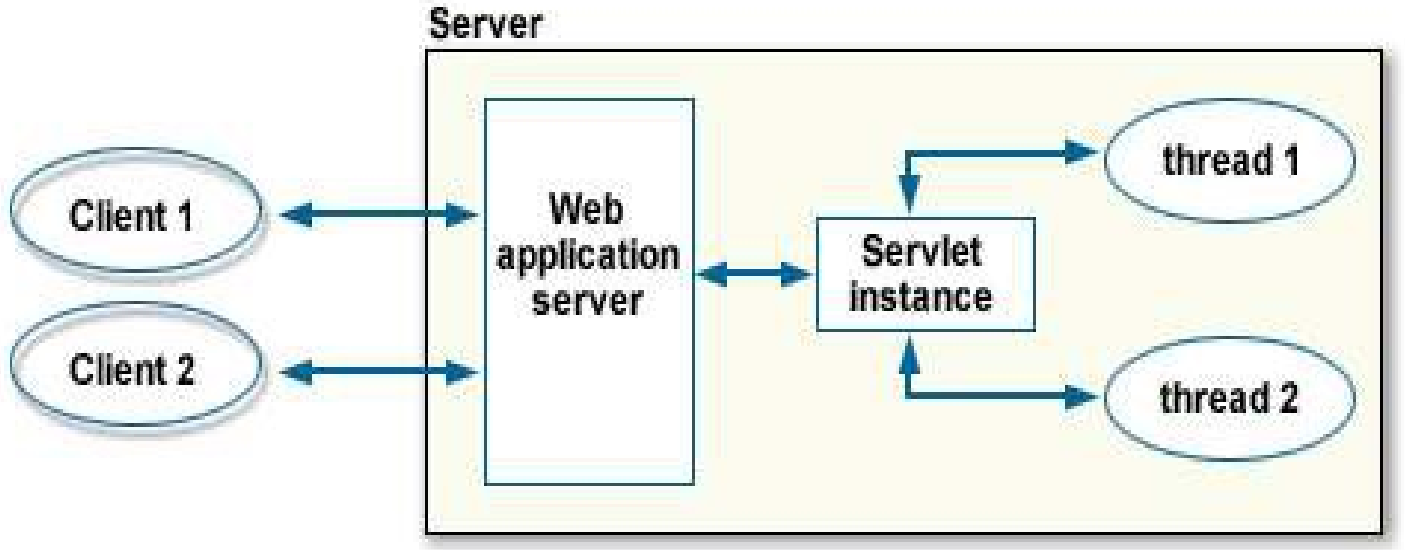
```
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js"></script>

<script type="text/javascript">
  $(document).ready(function () {
    $('#button1').on('click', function () {
      $.get("helloServlet", function (response) {
        $('#someDiv').html(response);
    });
  });

  var params = {param1: "value1", param2: "value2"};
  $('#button2').on('click', function () {
    $.post("helloServlet", $.param(params), function (response) {
      $('#someDiv').html(response);
    });
  });
});
</script>
```

**BODY**

```
<input type="button" id="button1" value="Invoke the servlet using GET. "/>
<input type="button" id="button2" value="Invoke the servlet using POST. "/>
<br/>
The response from the servlet will be displayed below:
<br/>
<div id="someDiv"></div>
```

In the servlet, you may check if the header field *X-Requested-With* has the value *XMLHttpRequest*

# Servlet Concurrency




- The servlet container is multithreaded.
- Multiple requests to the same servlet may be executed *at the same time*.
- Servlets are not thread-safe by default.
- The request and response objects are thread safe to use. Why?
- You need to take concurrency into consideration when you access shared resources from the service method of a servlet.

# Concurrency Example

```
public class ThreadSafeTestServlet extends HttpServlet {

  // Shared member variable
  private String notThreadSafe;

  public void doGet(HttpServletRequest request, HttpServletResponse res)
    throws ServletException, IOException {

    // Wrong
    notThreadSafe = request.getParameter("param");
    System.out.println(notThreadSafe);

    // OK
    String threadSafe = request.getParameter("param");
   }
}
```

# Th Servlet Lifecycle

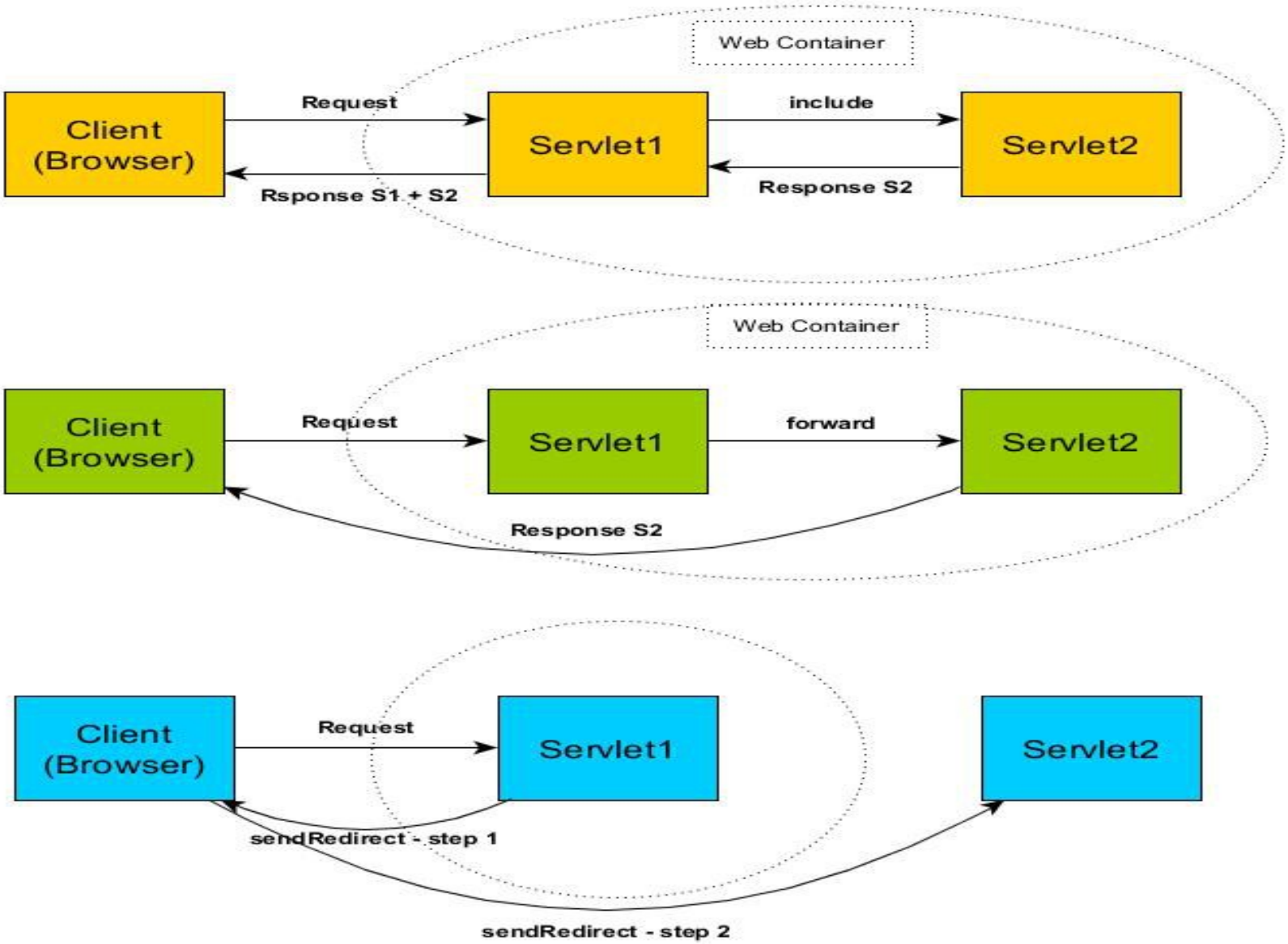
- A HTTP request to a specific servlet is received.
- The servlet container loads the servlet class in memory, creates an instance of it and invokes the init method. This may happen at application start-up if the servlet is annotated with: *@WebServlet(... loadOnStartup=1)*
- The servlet container calls the init method exactly once after instantiating the servlet.
- The servlet container creates a new thread and then invokes the service method of that single instance of the servlet from the newly created thread.
- The servlet container calls the destroy method to indicate to a servlet that the servlet is being taken out of service. This method is called only once at the end of the life cycle of a servlet, usually when the application is stopped / undeployed.

# Servlet Communication

*A servlet receives a request... What can it do?*

- Ignore the request … *not really*
- Create a response **on its own** … *not usually*
- Use **some help** from other resources, **including**
  - static pages, templates, etc.
  - the responses of other dynamic Web components
- **Redirect** the request to another Web component
  - from the same application as the servlet
  - from elsewhere, anywhere

# Include / Forward / Redirect



# Using the *RequestDispatcher*

- Forward

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/ThankYouServlet");
dispatcher.forward(request, response);
//no output here, the response is created by the forwarded servlet
```

- Include

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/HelperServlet");
PrintWriter out = response.getWriter();
out.println("..."); //this servlet starts to output response

//it includes the response from the helper
dispatcher.include(request, response);

out.println("..."); //it continues the creation of the response
out.close();
```

# Using the *RequestDispatcher*

- Forward

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/ThankYouServlet");
dispatcher.forward(request, response);
//no output here, the response is created by the forwarded servlet
```

- Include

```
RequestDispatcher dispatcher =
    getServletContext().getRequestDispatcher("/HelperServlet");
PrintWriter out = response.getWriter();
out.println("..."); //this servlet starts to output response

//it includes the response from the helper
dispatcher.include(request, response);

out.println("..."); //it continues the creation of the response
out.close();
```

# Using *sendRedirect*

- A **client side redirect** is a two step process, where the web application instructs the browser to fetch a second URL, which differs from the original
- **response.sendRedirect ( someUrl )**
- It requires two browser requests, not one
- The new request will be visible in the client browser
- Objects placed in the original request scope are not available to the second request

You may want to read “Forward versus redirect” at
http://www.javapractices.com/

# Attributes

- Servlets are able to communicate with other components via the **message passing** paradigm.
- What about using a **shared memory**?
- An **attribute** is an object that can be *set*, *get* or *removed* from a specified *scope*: *request, session, application.*

```
request.setAttribute("message", "Hello World");
session.setAttribute("shoppingCart",new ShoppingCart());
servletContext.setAttribute("version", "0.0.1")
```

- Make sure the attributes have unique names in their scope, in order to avoid conflicts

*What is a session ?*

What is a *servletContext* ?

# The ServletContext Object

- Defines a set of methods that a servlet uses to **communicate with its servlet container,** for example, to get the MIME type of a file, dispatch requests, or write to a log file.

```
ServletContext context = this.getServletContext();
  //this is a GenericServlet, usually a HttpServlet
context.log(context.getServerInfo());
context.getAttribute("version");
context.getInitParameter("resourcesPath");
...
```

- There is one context per "web application" per JVM Machine

  In the case of a web application marked "distributed" in its deployment descriptor, there will be one context instance for each virtual machine. In this situation, the context cannot be used as a location to share global information (because the information won't be truly global). Use an external resource like a database instead.
- The ServletContext object is contained within the ServletConfig object, which the Web server provides the servlet when the servlet is initialized.

# Web Sessions / The Context

- The *user* interacts with a *web application*
- During the communication **information** about their *dialogue* may be **stored on the server**.
  - where and for how long?
- The user expects **to be remembered**
  - after the authentication and during the dialogue
    - server side
  - later on, when he resumes the dialogue
    - client side

# Internet Protocols

Every “dialogue” over a network uses some kind of protocol:

- **Stateful**
  - a state is maintained on the server
  - *start – communication – stop*
  - TCP, FTP, etc.
- **Stateless**
  - no information is kept on the server
  - *request – response*
  - UDP, **HTTP**

# Web Session

- *Web Session* = the “dialogue” between an user and a web application
  - *login → buy stuff → logout*
- *Session Data* = information stored on the server specific to a certain “dialogue”
  - *user info, shopping cart,etc.*
- Over HTTP, sessions are **maintained by the application server** (on demand) using a:
  - *session ID,* usually an UUID

    `jsessionid=1A530637289A03B07199A44E8D531427`

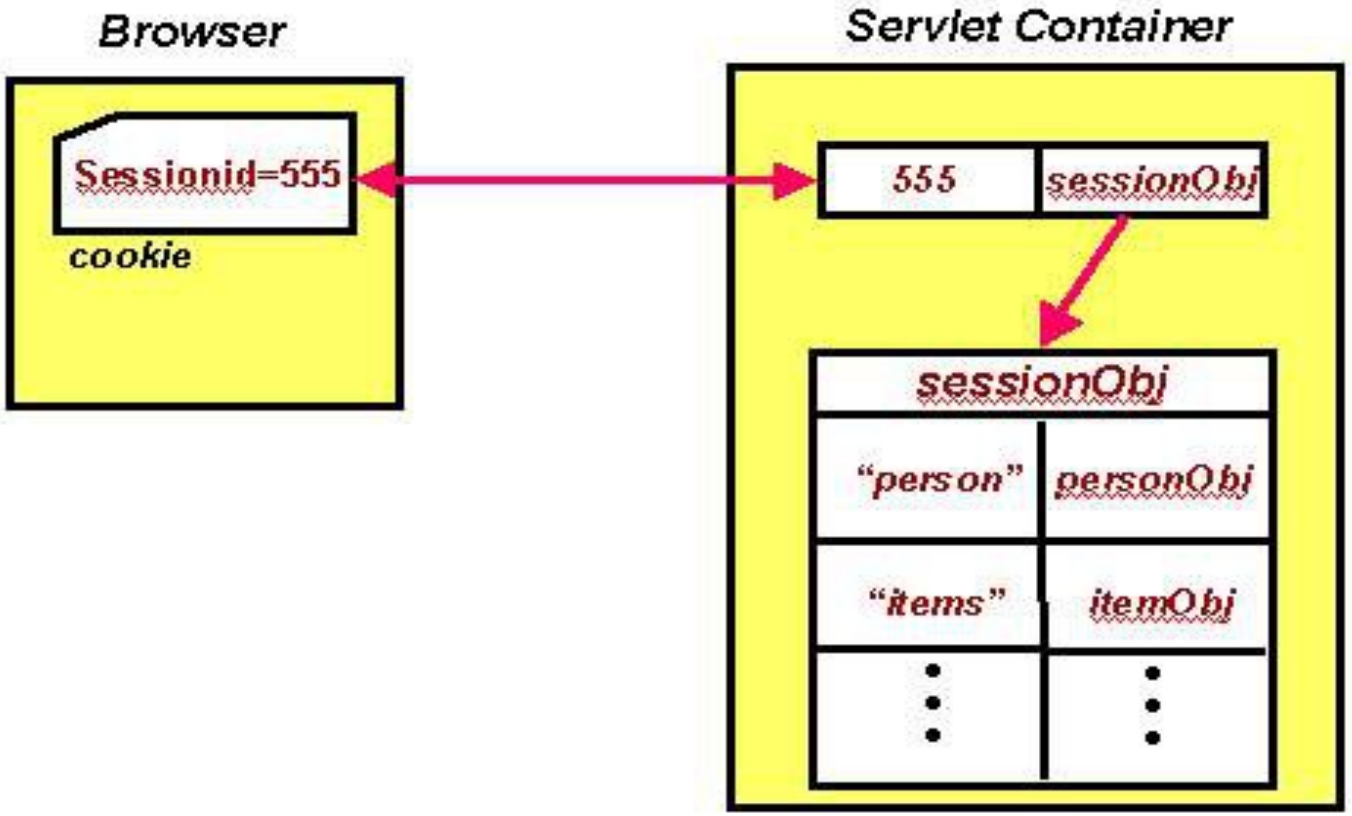
# Session Management over HTTP

A server maintains a session using either:

- **Cookies (jsessionid)**
  - *first HTTP request*: the server creates a **jsessionid cookie** and sends it to the server (inside the response)
  - *subsequent requests*: the cookie is included inside the client requests – the server reads the id and identifies the session
- **URL rewriting**

```
https://localhost:8080/myapp/
/somerequest.html;jsessionid=1A530637289A03B07199A44E8D531427
```

# Session Data




Session data is similar to a `Map<String, Object>`

# javax.servlet.http.HttpSession

```
//Example
//The client sends one number per request, the servlet computes their sum

//create the session
HttpSession session = request.getSession(true);

double number; //get the parameter from request
number = Double.parseDouble(request.getParameter("number"));

if (session.isNew())
  sum = new Double(0);
else
  sum = (Double)session.getAttribute("MyServlet.sum");}
}
sum = new Double(sum.doubleValue() + number);
session.putAttribute("MyServlet.sum", sum);

if (number == 0) {
  session.invalidate();
  // or use session.setMaxInactiveInterval
}

// Create the response
...
```

# URL Rewriting

## The HTML forms must be generated dynamically

```
...
String serlvetUrl = "http://localhost:8080/servlet/MyServlet";
//Generate the form
out.println("<HTML><FORM METHOD=GET" +
  " ACTION=" + response.encodeURL(servletUrl)} + " >");
...
out.println("</FORM>"</HTML>);
out.close();
return;
...
```

**http://localhost:8080/servlet/MyServlet**
**;jsessionid=935244D5D07441895052477146C371D3**
**?param1 = value1 & param2 = value2** …

**encodeURL**: encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged. The implementation of this method includes the logic to determine whether the session ID needs to be encoded in the URL. For example, if the browser supports cookies, or session tracking is turned off, URL encoding is unnecessary.
For robust session tracking, all URLs emitted by a servlet should be run through this method. Otherwise, URL rewriting cannot be used with browsers which do not support cookies. (occasionally, read the API documentation)

# Cookie API

```
//Example
//The client send one number per request, the servlet computes their sum
//Get the cookie
sum = 0;
Cookie theCookie=null;
Cookie cookies[] = request.getCookies();
if (cookies != null) {
  for (int i=0; i<cookies.length; i++) {
    theCookie = cookies[i];
    if (theCookie.getName().equals("CookieServlet.sum")) {
      sum = Double.parseDouble(theCookie.getValue());
      break;
    }
  }
}
...
//Save the cookie
theCookie = new Cookie("CookieServlet.sum", Double.toString(sum));
theCookie.setComment("the sum");}
theCookie.setMaxAge(30*60); //in seconds

response.setContentType("text/html");
response.addCookie(theCookie);
...
```