Java WEB programing

Introduction to WEB programing

HTML

- Hypertext Markup Language
- Defines the meaning and structure of web content

Tags

- <html> <body>
- <h1>   <p> <a>
-
- <img> <video> <audio>
- <form> <input> <button> <label>

Element

Element with opening and closing tegs:

<html> opening tag
content
</html> closing tag

Element without closing tag:
<hr>

Nested elements

```
<p>
        Some <strong>text</strong>
</p>

<div>Demo Text, with <span>some other</span> text.</div>
```

Attributes

<img src=‘picture.jpg’ alt=‘0’>

There are can be more than one attribute.

Page example

```
<!DOCTYPE html>
<html>
<head>
        <meta charset="utf-8">
        <title>Page title</title>
</head>
<body>
        <!-- Content-->
</body>
</html>
```

Another page example

```
<!DOCTYPE html>
<html>
<head>
        <meta charset="utf-8">
        <title>Title</title>
        <link rel="stylesheet" href="style.css">
        <script src="scripts.js"></script>
</head>
<body>
        <h1>Topic</h1>
        <p>some text</p>
</body>
</html>
```

HTML Specification

https://www.w3.org/TR/html53/

https://html.spec.whatwg.org/multipage/

Useful sections:
https://www.w3.org/TR/html53/dom.html#dom

https://www.w3.org/TR/html53/semantics.html#semantics

Code convention: https://codeguide.co/

HTML Validation

https://validator.w3.org/nu/

CSS

Style sheet language used for describing the presentation of a document written in a markup language like HTML

CSS

A style sheet consists of a list of rules.

Each rule or rule-set consists of one or more selectors, and a declaration block.

Selectors

https://www.w3.org/TR/selectors/

https://en.wikipedia.org/wiki/CSS

CSS Specification

https://www.w3.org/Style/CSS/current-work

CSS Code Convention

https://google.github.io/styleguide/htmlcssguide.html

Rule combination (cascading)

CSS:

```
.title { color: red; }
p { font-weight: bold; }
```

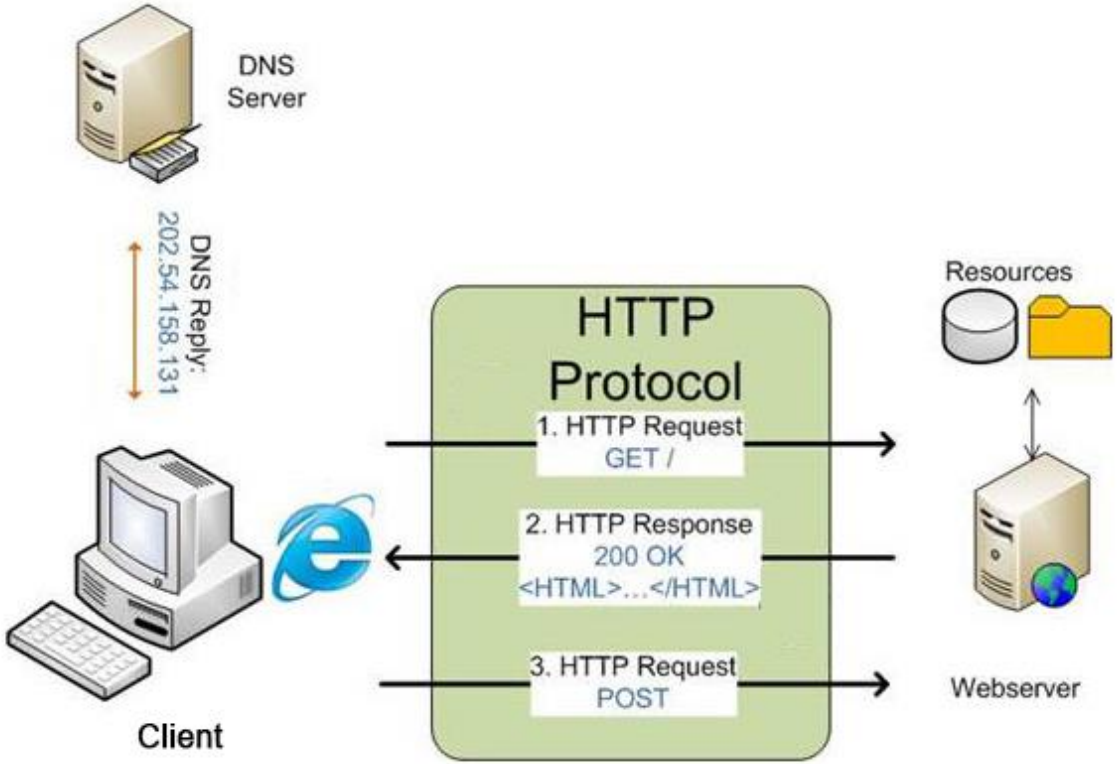
HTML:

```
<p class=“title”>
Content
</p>
```

- WEB WORKING PRINCIPLE.
- HTTP PROTOCOL.
- HTTP HEADERS.
- REQUEST AND RESPONSE STRUCTURES.
- RESPONSE CODES.
- REQUEST PROCESSING METHODS.

Web working principle.

Client–server computing model

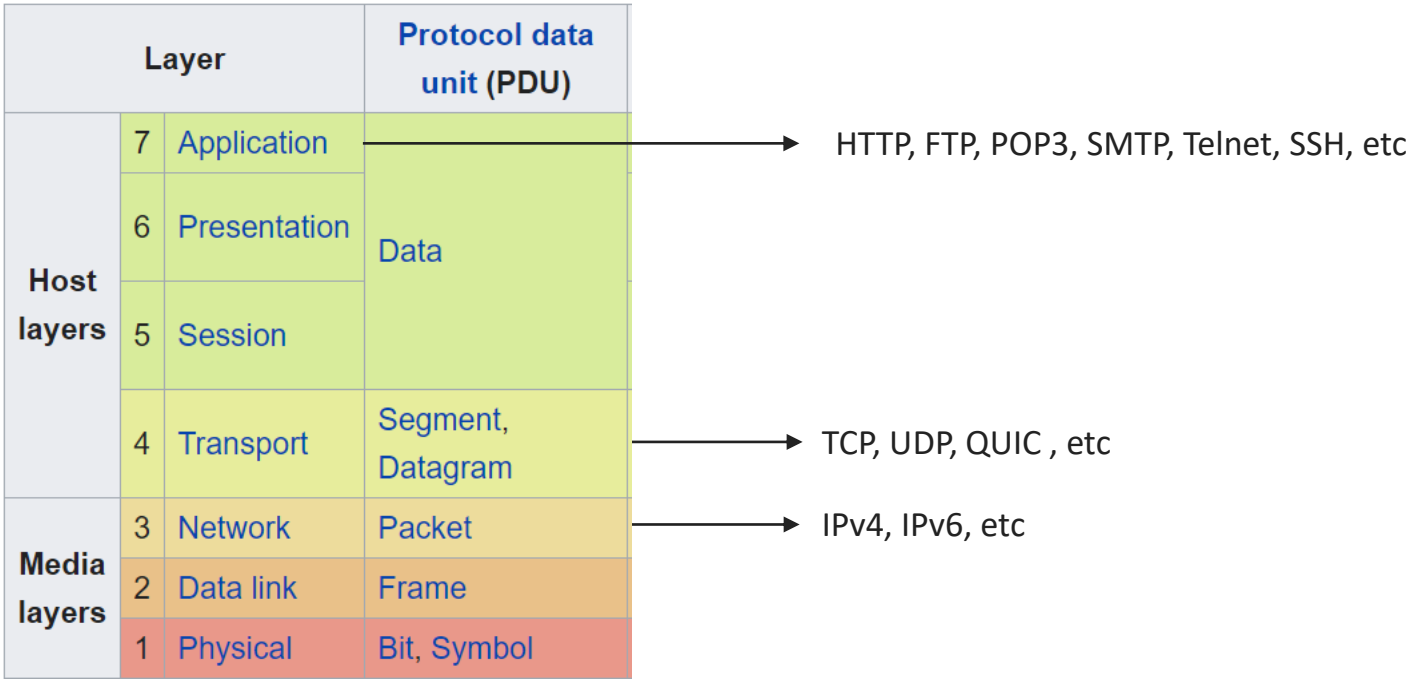


HTTP protocol.

HTTP Overview

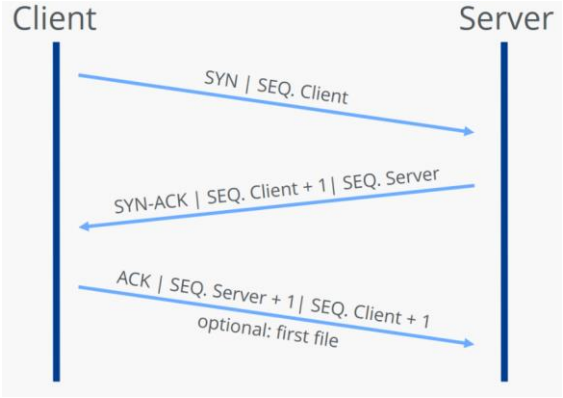
- **HTTP** - HyperText Transfer Protocol
- Development was initiated by **Tim Berners-Lee** (1989)
- Is an **application layer** protocol
- HTTP running over **TCP/IP**
- Resources are identified and located on the network by **URLs**
- Is the foundation of data communication for the **World Wide Web**
- Is a **request-response** protocol:
  - The client (user agent) submits an **HTTP request** message to the server
  - The server returns a **response** message to the client
- The server provides resources (such as **HTML** files and other content)
- **HTML** – Document language or Markup language used to create hypertext
- HTTP is media independent
- Is a **stateless** protocol

OSI model

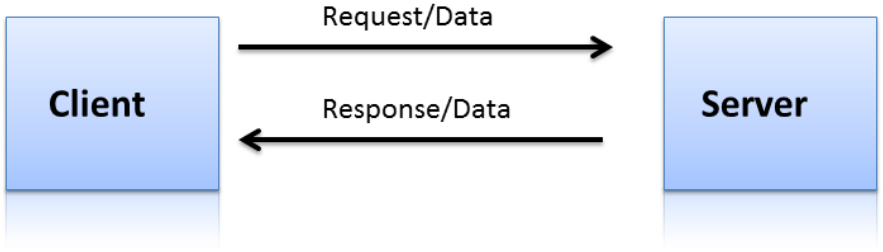

Layer
Protocol data unit (PDU)
Host layers
7 Application
6 Presentation
5 Session
Data
4 Transport
Segment, Datagram
Media layers
3 Network
Packet
2 Data link
Frame
1 Physical
Bit, Symbol
HTTP, FTP, POP3, SMTP, Telnet, SSH, etc
TCP, UDP, QUIC , etc
IPv4, IPv6, etc

TCP vs UDP

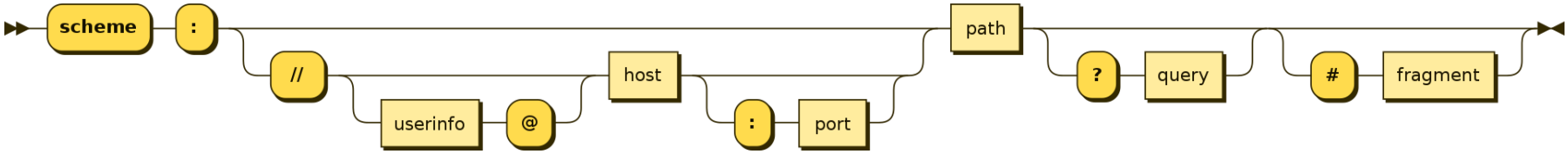
TCP handshake



Communication between the server and the client



URI - Uniform Resource Identifier

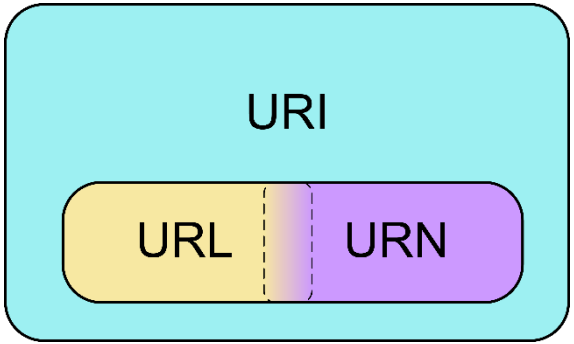



```
        userinfo       host      port
        ┌──┴───┐ ┌──────┴──────┐ ┌┴┐
https://john.doe@www.example.com:123/forum/questions/?tag=networking&order=newest#top
└─┬─┘   └─────────────┬─────────────┘└───────┬───────┘ └────────────┬────────────┘ └┬┘
scheme          authority                path                   query           fragment
```

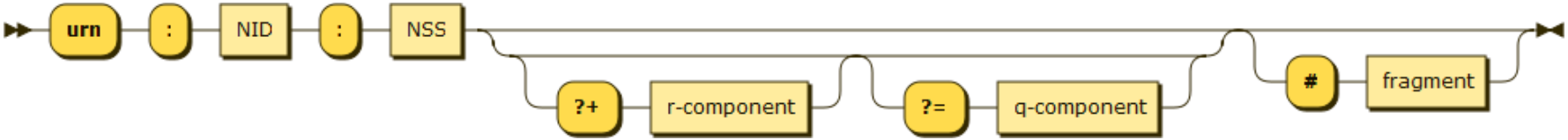
- mailto:John.Doe@example.com
- tel:+1-816-555-1212
- telnet://192.0.2.16:80/

URL, URN

- URL - Uniform Resource Locator
  - `http://www.example.com/pub/files/school.html`
  - `ftp://ftp.example.com/pub/files/school.tx`



- URN - Uniform Resource Name



  - `urn:isbn:0451450523`
  - `urn:isan:0000-0000-2CEA-0000-1-0000-0000-Y`
  - `urn:ietf:rfc:2648`

HTTP versions

Year	HTTP Version
1991	0.9
1996	1.0
1997	1.1
2015	2.0
2018	3.0

HTTP headers.

HTTP Headers

- HTTP header fields define the operating parameters of an HTTP transaction
- Header fields are colon-separated key-value pairs in clear-text string format, terminated by a CR and LF character sequence

```
Content-Type: text/html; charset=UTF-8
```

- Headers groups:
  - **General Headers**: These header fields have general applicability for both request and response messages
  - **Request Headers**: These header fields have applicability only for request messages
  - **Response Headers**: These header fields have applicability only for response messages
  - **Entity Headers**: These header fields define meta information about the entity-body or, if no body is present, about the resource identified by the request

MIME codes

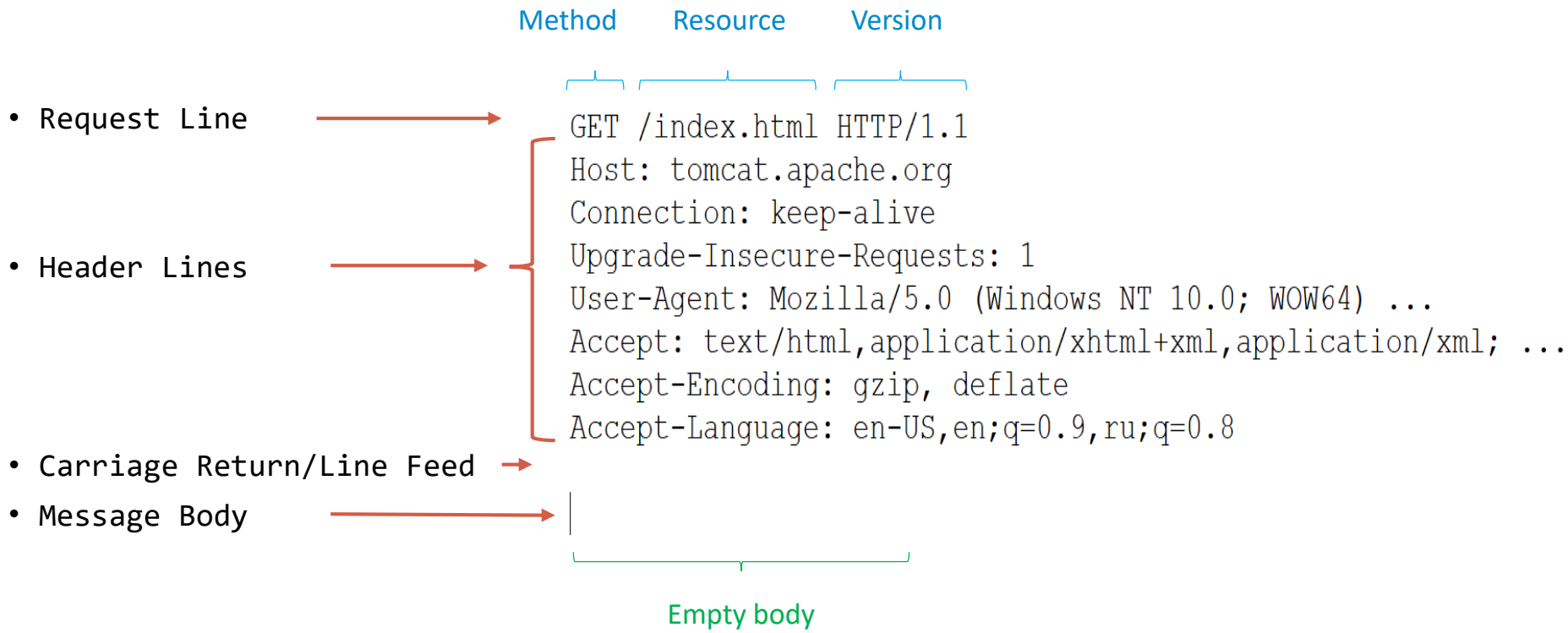
MIME - Multipurpose Internet Mail Extensions

Registered types:

- application (`application/javascript`, `application/xml`, `application/zip`)
- audio (`audio/mpeg`, `audio/ogg`)
- example
- font
- image (`image/gif`, `image/jpeg`, `image/png`)
- message
- model
- multipart (`multipart/form-data`)
- text (`text/css`, `text/csv`, `text/html`, `text/php`, `text/plain`, `text/xml`)
- video

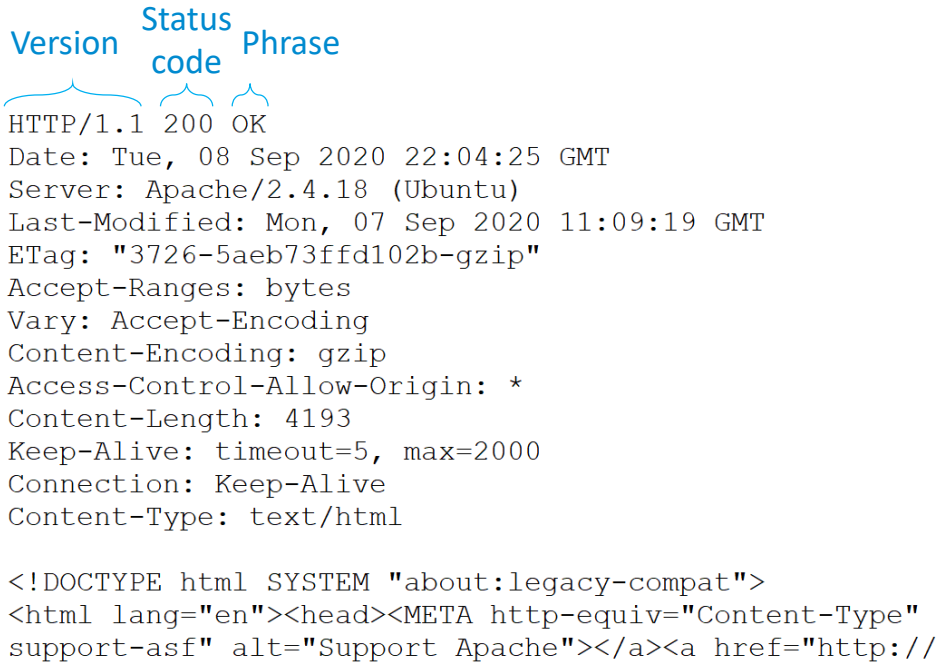
Request and response structures.

HTTP Request Format



HTTP Response Format

- Status Line
- Header Lines
- Carriage Return/Line Feed
- Entity Body



response codes.

Response codes

- 1xx: Informational
- 2xx: Successful
- 3xx: Redirection
- 4xx: Client Error
- 5xx: Server Error

```
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Wed, 09 Sep 2020 11:12:05 GMT
```

Request Processing Methods.

Request methods

- **GET** – requests a representation of the specified resource
- **HEAD** – asks for a response identical to that of a GET request, but without the response body
- **POST** – requests that the server accept the entity enclosed in the request as a new subordinate of the web resource identified by the URI
- **PUT** – requests that the enclosed entity be stored under the supplied URI
- **DELETE** – deletes the specified resource
- **TRACE** – echoes the received request so that a client can see what (if any) changes or additions have been made by intermediate servers
- **OPTIONS** – returns the HTTP methods that the server supports for the specified URL
- **CONNECT** – converts the request connection to a transparent TCP/IP tunnel
- **PATCH** – applies partial modifications to a resource

Sending GET requests

- GET requests:
  - request URL from the browser
  - reload resource in the browser
  - follow the URL from the HTML document

```
<a href="url">link text</a>
```

  - send data using the GET method from the form in the HTML document

```
<form action="/action_page">
        <input type="text" name="name" >
        <input type="submit" value="Submit">
</form>
```

  - use the appropriate user agent

```
curl -G google.com
```

Sending POST requests

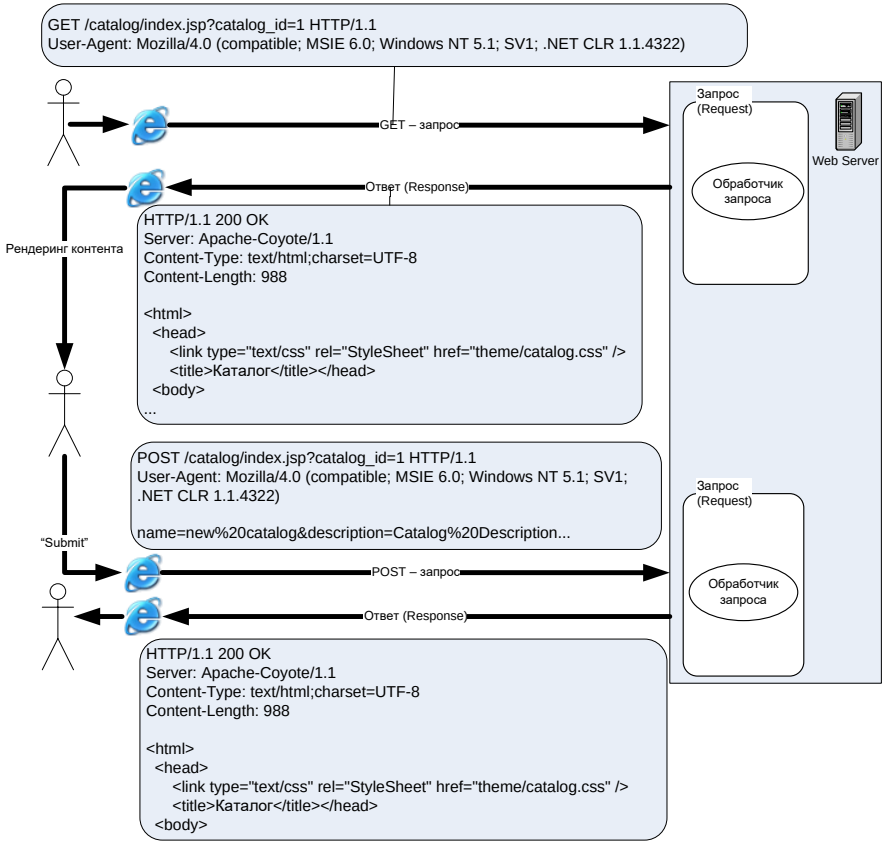
- POST requests:
  - send data using the GET method from the form in the HTML document

```
<form action="/action_page" method="post">
        <input type="text" name="name" >
        <input type="submit" value="Submit">
</form>
```

  - use the appropriate user agent

```
curl -d param=value https://www.w3schools.com/action_page.php
```

HTTP protocol in action



List of references

1) https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
2) https://www.w3.org/Protocols/
3) https://www.w3schools.com/html/html_forms.asp
4) https://www.tutorialspoint.com/http/index.htm

Introduction to Web Application Development (part II)

1. INTRODUCTION IN SERVLETS, HTTP SERVLETS.
2. SERVLET LIFE CYCLE.
3. SERVLET MAPPING.
4. TOMCAT: INSTALLATION, CONFIGURATION.
5. WEB-APPLICATION STRUCTURE.
6. DEPLOYING THE APPLICATION ON THE SERVER, WEB.XML

Introduction in Servlets

Java has strong support for web development. While Java on the desktop, with the notable exception of Eclipse RCP based application was never a huge success, Java is frequently used at the server side.

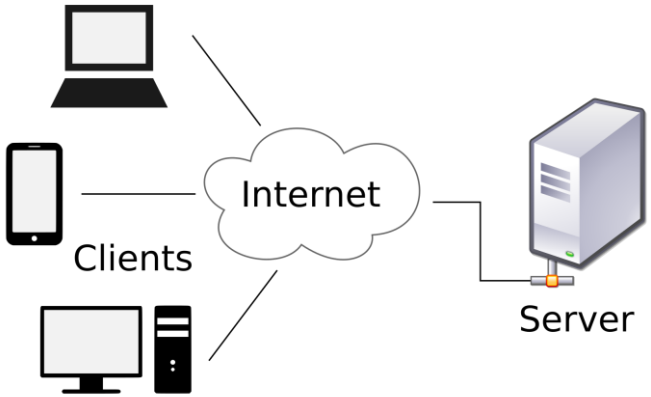
Introduction in Servlets

- **Jakarta EE**, formerly Java Enterprise Edition (Java EE) and Java 2 Platform, Enterprise Edition (J2EE) is a set of specifications, extending Java SE with specifications for enterprise features such as distributed computing and web services.
- Jakarta EE applications are run on reference runtimes, that can be microservices or application servers, which handle transactions, security, scalability, concurrency and management of the components it is deploying.
- Jakarta EE is defined by its specification. The specification defines APIs and their interactions.

Wiki: https://en.wikipedia.org/wiki/Jakarta_EE

Introduction in Servlets

- A **Jakarta Servlet** (formerly Java Servlet) is a Java software component that extends the capabilities of a server.
- **Servlets** can respond to many types of requests and most commonly implement web containers for hosting web applications on web servers.
- A **web container** (servlet container) is the component of a web server that interacts with Jakarta Servlets. It is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access-rights.
- A **web application** (or web app) is an application software that runs on a web server.
- A **web server** is server software, or hardware dedicated to running this software, that can satisfy client requests on the World Wide Web. It processes incoming network requests over HTTP and several other related protocols.



Introduction in Servlets

- A **servlet** is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients, usually across HTTP.
- To deploy and run a servlet, a web container must be used.
- A **servlet** is an object that receives a request and generates a response based on that request.
- The Servlet API, contained in the Java package hierarchy javax.servlet, defines the expected interactions of the web container and a servlet.
- The package javax.servlet.http defines HTTP-specific subclasses of the generic servlet elements, including session management objects that track multiple requests and responses between the web server and a client.
- Servlets may be packaged in a WAR file as a web application.

Wiki: https://en.wikipedia.org/wiki/Jakarta_Servlet

Introduction in Servlets

public interface javax.servlet.Servlet

Defines methods that all servlets must implement.

Method and Description
destroy() Called by the servlet container to indicate to a servlet that the servlet is being taken out of service.
getServletConfig() Returns a **ServletConfig** object, which contains initialization and startup parameters for this servlet.
getServletInfo() Returns information about the servlet, such as author, version, and copyright.
init(**ServletConfig** config) Called by the servlet container to indicate to a servlet that the servlet is being placed into service.
service(**ServletRequest** req, **ServletResponse** res) Called by the servlet container to allow the servlet to respond to a request.

Servlet life cycle

Servlet life cycle

public interface javax.servlet.Servlet

- Defines methods to initialize a servlet, to service requests, and to remove a servlet from the server. These are known as life-cycle methods and are called in the following sequence:
    1) The servlet is constructed, then initialized with the init method.
    2) Any calls from clients to the service method are handled.
    3) The servlet is taken out of service, then destroyed with the destroy method, then garbage collected and finalized.
- In addition to the life-cycle methods, this interface provides:
    - the getServletConfig method, which the servlet can use to get any startup information;
    - the getServletInfo method, which allows the servlet to return basic information about itself, such as author, version, and copyright.

Javadoc: https://javaee.github.io/javaee-spec/javadocs/javax/servlet/package-summary.html

Wiki: https://en.wikipedia.org/wiki/Jakarta_Servlet#Life_cycle_of_a_servlet

Servlet life cycle

public interface javax.servlet.ServletConfig

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

Method and Description
`getInitParameter(String name)` Gets the value of the initialization parameter with the given name.
`getInitParameterNames()` Returns the names of the servlet's initialization parameters as an `Enumeration` of `String` objects, or an empty `Enumeration` if the servlet has no initialization parameters.
`getServletContext()` Returns a reference to the **`ServletContext`** in which the caller is executing.
`getServletName()` Returns the name of this servlet instance.

Javadoc: https://javaee.github.io/javaee-spec/javadocs/javax/servlet/ServletConfig.html

Servlet life cycle

You can write a generic servlet that extends javax.servlet.GenericServlet or an HTTP servlet that extends javax.servlet.http.HttpServlet.

public abstract class javax.servlet.GenericServlet

- Defines a generic, protocol-independent servlet.
- Implements the Servlet and ServletConfig interfaces.
- To write a generic servlet, you need only override the abstract service method.

Javadoc: https://javaee.github.io/javaee-spec/javadocs/javax/servlet/GenericServlet.html

Servlet life cycle

public abstract class javax.servlet.http.HttpServlet

- Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site.
- A subclass of HttpServlet must override at least one method, usually one of these:
    - doGet, if the servlet supports HTTP GET requests
    - doPost, for HTTP POST requests
    - doPut, for HTTP PUT requests
    - doDelete, for HTTP DELETE requests
    - init and destroy, to manage resources that are held for the life of the servlet
    - getServletInfo, which the servlet uses to provide information about itself
- Servlets typically run on multithreaded servers, so be aware that a servlet must handle concurrent requests and be careful to synchronize access to shared resources.

Javadoc: https://javaee.github.io/javaee-spec/javadocs/javax/servlet/http/HttpServlet.html

Servlet life cycle

```
public abstract class javax.servlet.http.HttpServlet

protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
```

- Called by the server (via the service method) to allow a servlet to handle a GET request.
- When overriding this method, read the request data, write the response headers, get the response's writer or output stream object, and finally, write the response data. It's best to include content type and encoding.
- The GET method should be safe, that is, without any side effects for which users are held responsible. For example, most form queries have no side effects. If a client request is intended to change stored data, the request should use some other HTTP method.
- The GET method should also be idempotent, meaning that it can be safely repeated. Sometimes making a method safe also makes it idempotent. For example, repeating queries is both safe and idempotent, but buying a product online or modifying data is neither safe nor idempotent.

Servlet life cycle

```
public abstract class javax.servlet.http.HttpServlet

protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
```

- Called by the server (via the service method) to allow a servlet to handle a POST request. The HTTP POST method allows the client to send data of unlimited length to the Web server a single time and is useful when posting information such as credit card numbers.
- This method does not need to be either safe or idempotent.
- Operations requested through POST can have side effects for which the user can be held accountable, for example, updating stored data or buying items online.

Servlet life cycle

These methods (doGet and doPost) take two parameters: HttpServletRequest req, HttpServletResponse resp.

- public interface javax.servlet.http.HttpServletRequest
  - Extends the javax.servlet.ServletRequest interface to provide request information for HTTP servlets.
  - The servlet container creates an HttpServletRequest object and passes it as an argument to the servlet's service methods (doGet, doPost, etc).

- public interface javax.servlet.http.HttpServletResponse
  - Extends the javax.servlet.ServletResponse interface to provide HTTP-specific functionality in sending a response (for example, it has methods to access HTTP headers and cookies).
  - The servlet container creates an HttpServletResponse object and passes it as an argument to the servlet's service methods (doGet, doPost, etc).

Servlet mapping

Servlet mapping

- Upon receipt of a client request, the Web container determines the Web application to which to forward it.
- You define servlets as a part of a Web application in several entries in the Jakarta EE standard Web Application deployment descriptor, web.xml. The web.xml file is located in the WEB-INF directory of your Web application.
- The first entry, under the root servlet element in web.xml, defines a name for the servlet and specifies the compiled class that executes the servlet (or, instead of specifying a servlet class, you can specify a JSP.) The servlet element also contains definitions for initialization attributes and security roles for the servlet.
- The second entry in web.xml, under the servlet-mapping element, defines the URL pattern that calls this servlet.
- **Servlet mapping** controls how you access a servlet.

Servlet mapping

Specification of Mappings

- A string beginning with a ‘/’ character and ending with a ‘/*’ suffix is used for path mapping.
- A string beginning with a ‘*.’ prefix is used as an extension mapping.
- The empty string ("") is a special URL pattern that exactly maps to the application's context root, i.e., requests of the form http://host:port/<contextroot>/. In this case the path info is ’/’ and the servlet path and context path is empty string (““).
- 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.
- All other strings are used for exact matches only.

Section 12 of the Servlet specification:
https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf

Servlet mapping

Example Mapping Set

Consider the following set of mappings:

Path Pattern	Servlet
/foo/bar/*	servlet1
/baz/*	servlet2
/catalog	servlet3
*.bop	servlet4

Servlet mapping

Example Mapping Set

The following behavior would result:

Incoming Path	Servlet Handling Request
/foo/bar/index.html	servlet1
/foo/bar/index.bop	servlet1
/baz	servlet2
/baz/index.html	servlet2
/catalog	servlet3
/catalog/index.html	"default" servlet
/catalog/racecar.bop	servlet4
/index.bop	servlet4

Tomcat: Installation, configuration

Tomcat: Installation, configuration

Apache Tomcat is a webcontainer which allows to run servlet and JavaServer Pages (JSP) based web applications.

- Installation
- Managing Apache Tomcat
  - Start Tomcat
  - Test Tomcat
  - Admin console
  - Deployment
- Developing Java web applications

Apache Tomcat: http://tomcat.apache.org/

Tomcat: Installation, configuration

These are some of the key tomcat directories:

- /bin - Startup, shutdown, and other scripts. The *.sh files (for Unix systems) are functional duplicates of the *.bat files (for Windows systems). Since the Win32 command-line lacks certain functionality, there are some additional files in here.
- /conf - Configuration files and related DTDs. The most important file in here is server.xml. It is the main configuration file for the container.
- /logs - Log files are here by default.
- /webapps - This is where your webapps go.

Tomcat: Installation, configuration

CATALINA_HOME and CATALINA_BASE

- CATALINA_HOME: Represents the root of your Tomcat installation,
  for example */home/tomcat/apache-tomcat-9.0.10 or C:\Program Files\apache-tomcat-9.0.10.*
- CATALINA_BASE: Represents the root of a runtime configuration of a specific Tomcat instance.
  If you want to have multiple Tomcat instances on one machine, use the CATALINA_BASE property.

If you set the properties to different locations, the CATALINA_HOME location contains static sources, such as .jar files, or binary files. The CATALINA_BASE location contains configuration files, log files, deployed applications, and other runtime requirements.

Web-application structure

Web-application structure

Web application's "document root" directory

- *.html, *.jsp, etc. - The HTML and JSP pages, along with other files that must be visible to the client browser (such as JavaScript, stylesheet files, and images) for your application. In larger applications you may choose to divide these files into a subdirectory hierarchy, but for smaller apps, it is generally much simpler to maintain only a single directory for these files.

- /WEB-INF/web.xml - The Web Application Deployment Descriptor for your application. This is an XML file describing the servlets and other components that make up your application, along with any initialization parameters and container-managed security constraints that you want the server to enforce for you. This file is discussed in more detail in the following subsection.

Web-application structure

Web application's "document root" directory

- /WEB-INF/classes/ - This directory contains any Java class files (and associated resources) required for your application, including both servlet and non-servlet classes, that are not combined into JAR files. If your classes are organized into Java packages, you must reflect this in the directory hierarchy under /WEB-INF/classes/. For example, a Java class named com.mycompany.mypackage.MyServlet would need to be stored in a file named /WEB-INF/classes/com/mycompany/mypackage/MyServlet.class.

- /WEB-INF/lib/ - This directory contains JAR files that contain Java class files (and associated resources) required for your application, such as third party class libraries or JDBC drivers.

Section 10 of the Servlet specification:
https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf

Deploying the application on the server, web.xml

Deploying the application on the server, web.xml

Deployment With Tomcat

In order to be executed, a web application must be deployed on a servlet container. Web application can be deployed in Tomcat by one of the following approaches:

- Copy unpacked directory hierarchy into a subdirectory in directory $CATALINA_BASE/webapps/. Tomcat will assign a context path to your application based on the subdirectory name you choose.
- Copy the web application archive file into directory $CATALINA_BASE/webapps/. When Tomcat is started, it will automatically expand the web application archive file into its unpacked form, and execute the application that way. This approach would typically be used to install an additional application, provided by a third party vendor or by your internal development staff, into an existing Tomcat installation.
- Use the Tomcat "Manager" web application to deploy and undeploy web applications. Tomcat includes a web application, deployed by default on context path /manager, that allows you to deploy and undeploy applications on a running Tomcat server without restarting it.

Deploying the application on the server, web.xml

The /WEB-INF/web.xml file contains the Web Application Deployment Descriptor for your application. This file is an XML document, and defines everything about your application that a server needs to know.

- ServletContext Init Parameters
- Session Configuration
- Servlet Declaration and Servlet Mappings
- Application Lifecyle Listener classes
- Filter Definitions and Filter Mappings
- MIME Type Mappings
- Welcome File list
- Error Pages
- Locale and Encoding Mappings
- Security configuration

Section 14 of the Servlet specification:
https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf

Deploying the application on the server, web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
   <display-name>test</display-name>
   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
   <servlet>
      <servlet-name>my</servlet-name>
      <servlet-class>com.my.MyServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>my</servlet-name>
      <url-pattern>/servlet</url-pattern>
   </servlet-mapping>
</web-app>
```

Deploying the application on the server, web.xml

```
package com.my;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                                        throws ServletException, IOException {
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.print("Hello, ");
        out.print(req.getParameter("name"));
    }
}
```

Deploying the application on the server, web.xml

http://localhost:8080/test/servlet?name=John

Hello, John

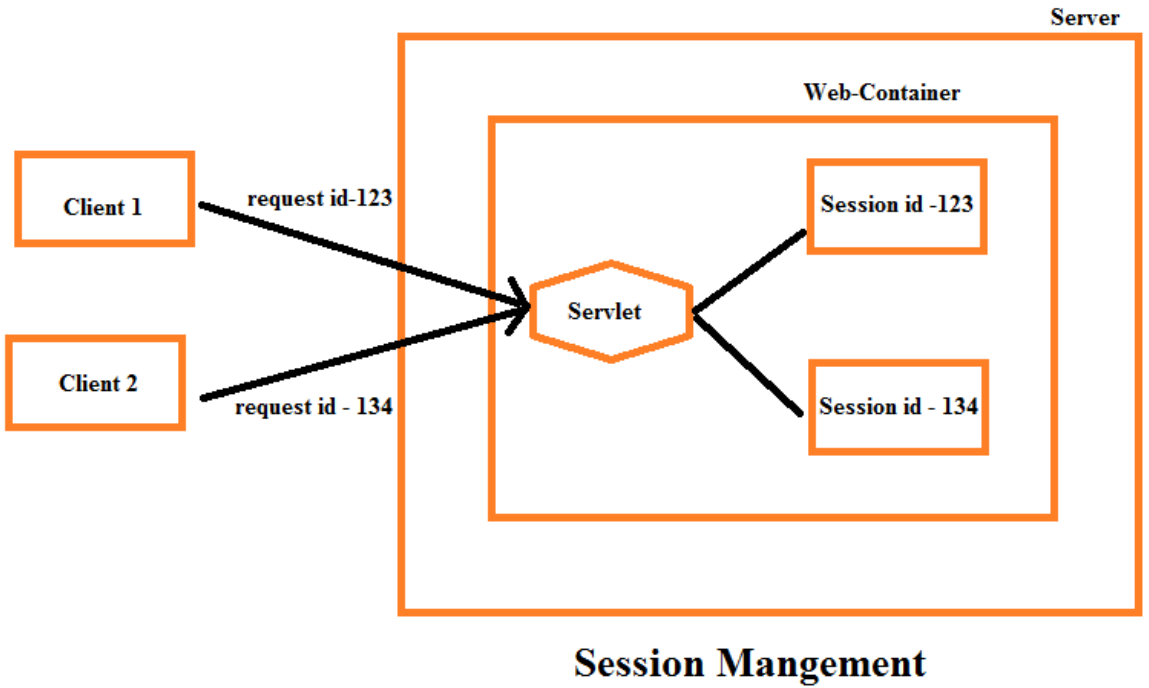
Sessions

CONCEPT OF SESSION.

COOKIES.

Concept of session.

Concept of session



Session Mangement

Concept of session

- Create new or get an existing session
- Set or get the data from the session
- Remove session if it is no longer needed

Concept of session

- Creating or Accessing a Session (**HttpServletRequest**)
  - `HttpSession getSession()`
  - `HttpSession getSession(boolean create);`
- Examining Session Properties (**HttpSession**)
  - `long getCreationTime()`
  - `String getId()`
  - `long getLastAccessedTime()`
  - `int getMaxInactiveInterval()`
  - `boolean isNew()`

Concept of session

- Binding Data to a Session (**HttpSession**)
  - `Object getAttribute(String name)`
  - `Enumeration getAttributeNames()`
  - `void removeAttribute(String name)`
  - `void setAttribute(String name, Object value)`
- Invalidating a Session (**HttpSession**)
  - `void invalidate()`
    - `InvalidStateException`

Concept of session

- Setting a Session Timeout (**`HttpSession`**) *in seconds*
  - `void setMaxInactiveInterval(int interval)`

- Setting a Session Timeout (**`Web.xml`***) in minutes*

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

Concept of session

- Interface HttpSessionBindingListener
  - `void valueBound(HttpSessionBindingEvent event)`
  - `void valueUnbound(HttpSessionBindingEvent event)`

- Negative parameter value - session timeout will never expire

Concept of session

- URL rewriting (**HttpServletResponse**)
  - `String encodeURL(String url)`
  - `String encodeRedirectURL(String url)`

Cookies.

Cookies

- HTTP is stateless protocol
- The information is stored on the client side as a small amount of information
- The information is transmitted in request and response headers
- Cookies are transmitted in clear text (security issue)
- Cookies are used to save user preferences, customize data, remember the last visit, etc.

Cookie Properties

- `name=value`
  - This sets both the cookie's name and its value
- `expires=date`
  - This optional value sets the date that the cookie will expire on. If the expires value is not given, the cookie will be destroyed the moment the browser is closed
- `max-age=seconds`
  - Set the cookie's expiration as an interval of seconds in the future, relative to the time the browser received the cookie
- `path=path`
  - The path gives you the chance to specify a directory where the cookie is active. Usually the path is set to /, which means the cookie is valid throughout the entire domain

Cookie Properties

- `domain=domain`
  - This optional value specifies a domain within which the cookie applies. Only websites in this domain will be able to retrieve the cookie. Usually this is left blank, meaning that only the domain that set the cookie can retrieve it
- `secure`
  - This optional flag indicates that the browser should use SSL when sending the cookie to the server. This flag is rarely used
- `HttpOnly`
  - Directs browsers not to expose cookies through channels other than HTTP (and HTTPS) requests

Set Or Remove Cookie

- Create Cookie (**`Cookie`**)
  - `Cookie(String name, String value)`

- Set Properties (**`Cookie`**)
  - `setValue(String newValue)`
  - `setMaxAge(int expiry)`
    - To remove Cookie set the `max-age` property to 0

- Send Cookie (**`HttpServletResponse`**)
  - `addCookie(Cookie cookie)`

Get Cookie

- Get all Cookies (**`HttpServletRequest`**)
  - `Cookie[] getCookies()`
- Find the required cookie (**`Cookie`**)
  - `String getName()`
- Get cookie value (**`Cookie`**)
  - `String getValue()`

List of references

1) https://en.wikipedia.org/wiki/Session_(computer_science)
2) https://tomcat.apache.org/tomcat-9.0-doc/servletapi/javax/servlet/http/HttpSession.html
3) https://www.javatpoint.com/http-session-in-session-tracking
4) https://en.wikipedia.org/wiki/HTTP_cookie

Filters and Sessions

1. CONCEPT OF FILTER.
2. FILTER LIFE CYCLE.
3. FILTERS AND SERVLET CONTEXT.
4. FILTER USAGE EXAMPLES.
5. REQUESTDISPATCHER: FORWARD, INCLUDE
6. HTTPSERVLETRESPONSE: SENDREDIRECT.

The **web.xml** file does not represent the entire configuration that is available for the web application. There are other servlets filters, and listeners that can be defined using programmatic configurations, annotations, and web fragments.

Marking the web application metadata complete will prevent annotations and web fragments from being able to configure components.

web.xml

```
...
  <servlet>
    <servlet-name>xxx</servlet-name>
    <servlet-class>Main</servlet-class>
    <init-param>
      <param-name>a</param-name>
      <param-value>aaa</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
```

```
<servlet>
  <servlet-name>yyy</servlet-name>
  <servlet-class>Main</servlet-class>
  <init-param>
    <param-name>b</param-name>
    <param-value>bbb</param-value>
  </init-param>
  <init-param>
    <param-name>c</param-name>
    <param-value>ccc</param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>
<servlet>
  <servlet-name>zzz</servlet-name>
  <servlet-class>Main</servlet-class>
  <load-on-startup>3</load-on-startup>
</servlet>
```

web.xml

```
<servlet-mapping>
   <servlet-name>xxx</servlet-name>
   <url-pattern>/x</url-pattern>
</servlet-mapping>
<servlet-mapping>
   <servlet-name>yyy</servlet-name>
   <url-pattern>/y</url-pattern>
</servlet-mapping>
<servlet-mapping>
   <servlet-name>zzz</servlet-name>
   <url-pattern>/z</url-pattern>
</servlet-mapping>
```

```
<context-param>
   <param-name>M</param-name>
   <param-value>MMM</param-value>
</context-param>
<context-param>
   <param-name>W</param-name>
   <param-value>WWW</param-value>
</context-param>
...
```

Main.java

```
@WebServlet("/Main")
public class Main extends HttpServlet {
   private static int stCounter = 0;
   private int counter = 0;
   private static final long serialVersionUID = 1L;
   public Main() {
      System.out.println("Main()");
      counter = ++stCounter;
   }
   @Override
   public void init() throws ServletException {
      System.out.println("init() #"
            + counter + " of " + stCounter + ". Context:"
            + Collections.list(getServletContext()
                  .getInitParameterNames())
            + ". Local: " + Collections
                  .list(getInitParameterNames()));
      super.init();
   }
   @Override
   protected void service(HttpServletRequest req,
         HttpServletResponse resp)
         throws ServletException, IOException {
      System.out.println("service() #" + counter
            + " of " + stCounter
            + ". Servlet name: " + getServletName());
      super.service(req, resp);
   }

}
```

http://localhost:8080/w01/

Main()
init() #1 of 1. Context:[W, M]. Local: [a]
Main()
init() #2 of 2. Context:[W, M]. Local: [b, c]
Main()
init() #3 of 3. Context:[W, M]. Local: []

http://localhost:8080/w01/x, ...y, ...z

service() #1 of 3. Servlet name: xxx

service() #2 of 3. Servlet name: yyy

service() #3 of 3. Servlet name: zzz

http://localhost:8080/w01/Main

metadata-complete is not specified or is set to "false"

Main()

init() #4 of 4. Context:[W, M]. Local: []

service() #4 of 4. Servlet name: Main

metadata-complete="true"

Concept of filter

Concept of Filter

- The Java Servlet API has classes and methods that provide a lightweight framework for filtering active and static content.
- A filter is an object that performs filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both.
- A filter is a reusable piece of code that can transform the content of HTTP requests, responses, and header information.
- Filters do not generally create a response or respond to a request as servlets do, rather they modify or adapt the requests for a resource, and modify or adapt responses from a resource.
- Filters perform filtering in the doFilter method.
- Every Filter has access to a FilterConfig object from which it can obtain its initialization parameters, and a reference to the ServletContext which it can use, for example, to load resources needed for filtering tasks.
- Filters are configured in the deployment descriptor of a web application.

Concept of Filter

Among the types of functionality available to the developer needing to use filters are the following:

- The accessing of a resource before a request to it is invoked.
- The processing of the request for a resource before it is invoked.
- The modification of request headers and data by wrapping the request in customized versions of the request object.
- The modification of response headers and response data by providing customized versions of the response object.
- The interception of an invocation of a resource after its call.
- Actions on a servlet, on groups of servlets, or static content by zero, one, or more filters in a specifiable order.

Concept of Filter

Examples of filtering components:

- Authentication Filters
- Logging and Auditing Filters
- Image conversion Filters
- Data compression Filters
- Encryption Filters
- Tokenizing Filters
- Filters that trigger resource access events
- XSL/T filters
- Mime-type chain Filter

Concept of Filter

Main Concepts

- The application developer creates a filter by implementing the javax.servlet.Filter interface and providing a public constructor taking no arguments. The class is packaged in the Web Archive along with the static content and servlets that make up the Web application.
- A filter is declared using the <filter> element in the deployment descriptor.
- A filter or collection of filters can be configured for invocation by defining <filter-mapping> elements in the deployment descriptor. This is done by mapping filters to a particular servlet by the servlet's logical name, or mapping to a group of servlets and static content resources by mapping a filter to a URL pattern.

Section 6 of the Servlet specification:
https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf

Filter life cycle

Filter Life Cycle

- After deployment of the Web application, and before a request causes the container to access a Web resource, the container must locate the list of filters that must be applied to the Web resource.
- The container must ensure that it has instantiated a filter of the appropriate class for each filter in the list, and called its init(FilterConfig config) method.
- Only one instance per <filter> declaration in the deployment descriptor is instantiated per JVM of the container.
- The container provides the filter config as declared in the filter's deployment descriptor, the reference to the ServletContext for the Web application, and the set of initialization parameters.
- When the container receives an incoming request, it takes the first filter instance in the list and calls its doFilter method, passing in the ServletRequest and ServletResponse, and a reference to the FilterChain object it will use.

Filter Life Cycle: A.java

```
@WebServlet("/A")
public class A extends HttpServlet {
   private static final long serialVersionUID = 1L;
   public A() {
      System.out.println("Servlet Main()");
   }
   public void init() throws ServletException {
      System.out.println("Servlet init()");
      super.init();
   }
   protected void service(HttpServletRequest req,
         HttpServletResponse resp)
               throws ServletException, IOException {
      System.out.println("Servlet service()");
      super.service(req, resp);
   }
   public void destroy() {
      System.out.println("Servlet destroy()");
      super.destroy();
   }

   protected void doGet(HttpServletRequest request,
         HttpServletResponse response)
               throws ServletException, IOException {
      System.out.println("Servlet doGet()");
      response.getWriter().append("Served at: ")
         .append(request.getContextPath());
   }

}
```

Filter Life Cycle: **One.java**

```
@WebFilter("/*")
public class One implements Filter {
    public One() {
        System.out.println("Filter One()");
    }
    public void init(FilterConfig fConfig) throws ServletException {
        System.out.println("Filter init()");
    }
    public void destroy() {
        System.out.println("Filter destroy()");
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        System.out.println("Filter --->>> before doFilter()");
        chain.doFilter(request, response);
        System.out.println("Filter <<<--- after doFilter()");
    }
}
```

Filter Life Cycle

http://localhost:8080/w02/

Filter One()

Filter init()

...

Filter --->>> before doFilter()

Filter <<<--- after doFilter()

http://localhost:8080/w02/A

Servlet Main()

Servlet init()

Filter --->>> before doFilter()

Servlet service()

Servlet doGet()

Filter <<<--- after doFilter()

Filters and servlet context

Filters and Servlet Context: One.java

```
@WebFilter(urlPatterns = { "/A" }, initParams = { @WebInitParam(name = "active", value = "xxx") })
public class One implements Filter {
    FilterConfig config;
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        if (config.getInitParameter("active").equalsIgnoreCase("true")) {
            System.out.println("Filter sets attribute.");
            config.getServletContext().setAttribute("Attribute", "Value");
        }
        chain.doFilter(request, response);
    }
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        config = filterConfig;
    }
}
```

Filters and Servlet Context: A.java

```
@WebServlet("/A")
public class A extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        System.out.println("Servlet doGet()");
        System.out.println(getServletContext().getAttribute("Attribute"));
        response.getWriter().append("Served at: ").append(request.getContextPath());
    }

}
```

Filters and Servlet Context: http://localhost:8080/w04/A

@WebFilter(...(name = "active", value = "xxx") })

Servlet doGet()
null

@WebFilter(...(name = "active", value = "xxx") })

Filter sets an attribute.
Servlet doGet()
Value

RequestDispatcher: forward, include

RequestDispatcher: forward, include

void javax.servlet.RequestDispatcher.forward(ServletRequest request, ServletResponse response)

throws ServletException, IOException

- Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.
- Allows one servlet to do preliminary processing of a request and another resource to generate the response.
- Should be called before the response has been committed to the client (before response body output has been flushed). If the response already has been committed, this method throws an IllegalStateException. Uncommitted output in the response buffer is automatically cleared before the forward.
- For a RequestDispatcher obtained via getRequestDispatcher(), the ServletRequest object has its path elements and parameters adjusted to match the path of the target resource.

RequestDispatcher: forward, include

void javax.servlet.RequestDispatcher.include(ServletRequest request, ServletResponse response)

throws ServletException, IOException

- Enables programmatic server-side includes. It includes the content of a resource (servlet, JSP page, HTML file) in the response.
- The ServletResponse object has its path elements and parameters remain unchanged from the caller's. The included servlet cannot change the response status code or set headers; any attempt to make a change is ignored.
- The request and response parameters must be either the same objects as were passed to the calling servlet's service method or be subclasses of the ServletRequestWrapper or ServletResponseWrapper classes that wrap them.

HttpServletResponse: sendRedirect

HttpServletResponse: sendRedirect

void javax.servlet.http.HttpServletResponse.sendRedirect(String location) throws IOException

- Sends a temporary redirect response to the client using the specified redirect location URL.
- This method can accept relative URLs; the servlet container must convert the relative URL to an absolute URL before sending the response to the client.
    - If the location is relative without a leading '/' the container interprets it as relative to the current request URI.
    - If the location is relative with a leading '/' the container interprets it as relative to the servlet container root.
- If the response has already been committed, this method throws an IllegalStateException. After using this method, the response should be considered to be committed and should not be written to.

HttpServletResponse: sendRedirect

```
@WebServlet("/a")
public class A extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        System.out.println("A#doGet()");
        response.getWriter().append("Served by A ");
    }

}
```

HttpServletResponse: sendRedirect

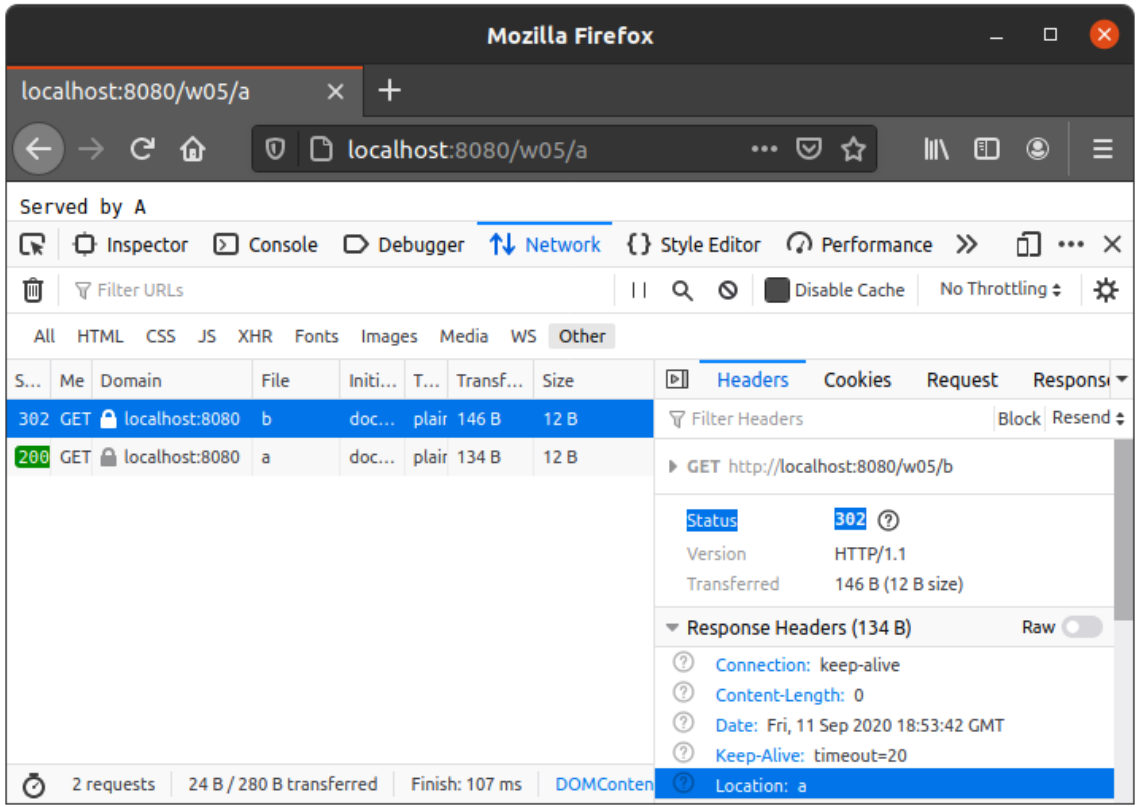
```
@WebServlet("/b")
public class B extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        System.out.println("B#doGet()");
        response.getWriter().append("Served by B ");
        response.sendRedirect("a");
        System.out.println("redirected");
    }

}
```

HttpServletResponse: sendRedirect

http://localhost:8080/w05/b

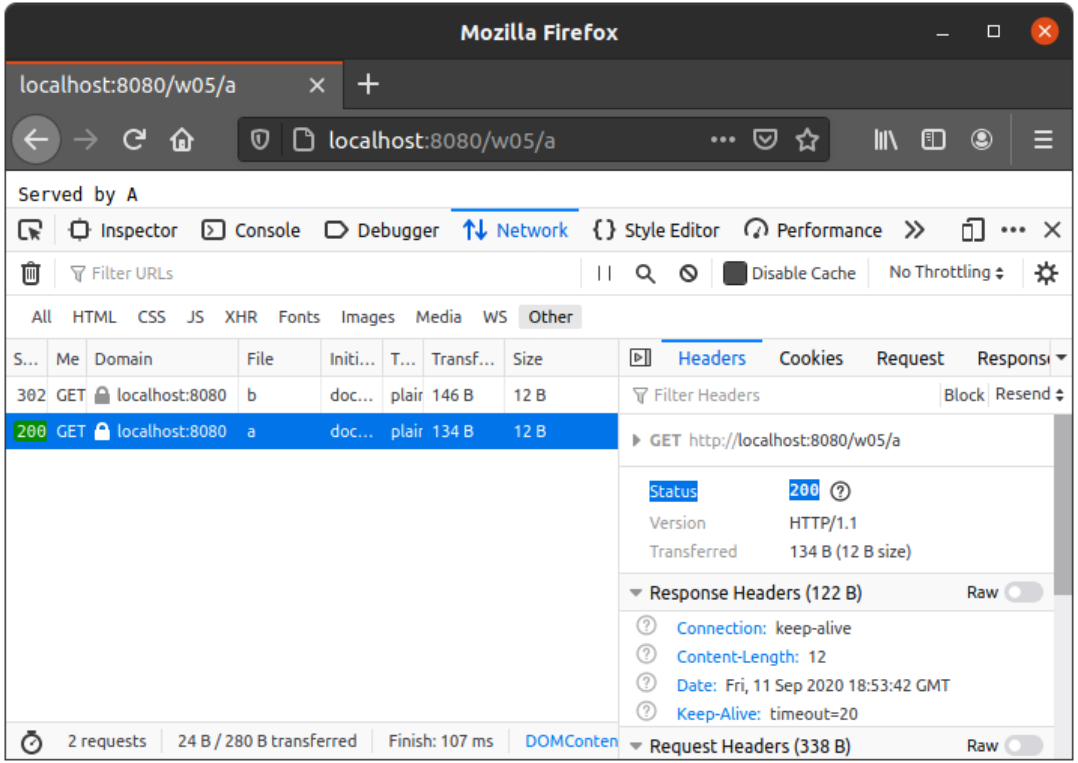
B#doGet()
redirected
A#doGet()

http://localhost:8080/w05/b



HttpServletResponse: sendRedirect

http://localhost:8080/w05/b



JSP and JSTL

(Part I: JSP)

1. JSP OVERVIEW
2. JSP LIFE CYCLE
3. JSP COMPONENTS
4. JSP: IMPLICIT OBJECTS
5. ERROR PAGE (CONFIGURATION IN WEB.XML)
6. PRG PATTERN (POST/REDIRECT/GET)

The JavaServer Pages is a technology for inserting dynamic content into an HTML or XML page using a Java servlet container.

You can think of servlets as Java code with HTML inside; you can think of JSP as HTML with Java code inside.

JSP overview

JSP Overview

- A JSP page is a textual document that describes how to create a response object from a request object for a given protocol.
- The processing of the JSP page may involve creating and/or using other objects.
- A JSP page defines a JSP page implementation class that implements the semantics of the JSP page. This class implements the javax.servlet.Servlet interface.
- At request time a request intended for the JSP page is delivered to the JSP page implementation object for processing.

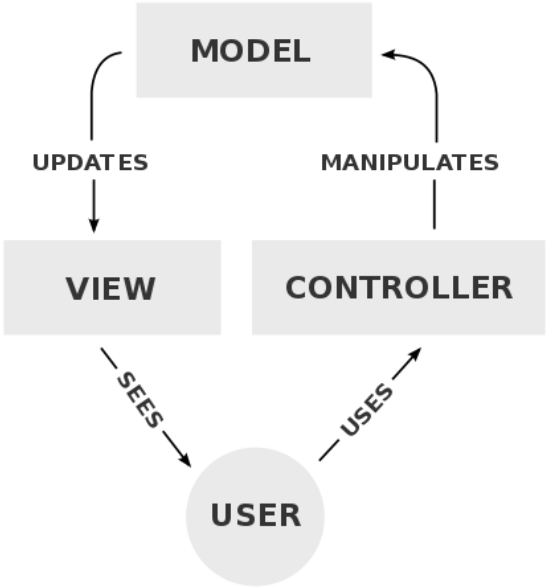
JSP 2.3 Specification: https://jcp.org/aboutJava/communityprocess/mrel/jsr245/index2.html

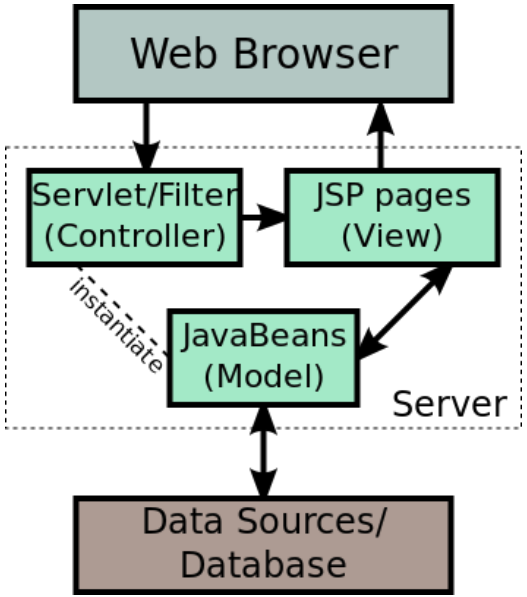
JSP Overview

- With servlets, it is easy to:
  - Read form data.
  - Read HTTP request headers.
  - Set HTTP status codes and response headers.
  - Use cookies and session tracking.
  - Share data among servlets.
  - Remember data between requests.
- JSP makes it easier to:
  - Write HTML and use standard HTML tools.
  - Read and maintain the HTML.
- JSP makes it possible to have different members of your team do the HTML layout than do the Java programming.

JSP Overview

- JSP encourages you to separate the code that creates the content (Java) from the code that presents it (HTML).
- JSP can be used independently or as the view component of a server-side model-view-controller design, normally with JavaBeans as the model and Java servlets as the controller.





JSP life cycle

JSP Life Cycle

A JSP compiler is a program that parses JSPs, and transforms them into executable Java Servlets. It is usually embedded into the application server and run automatically the first time a JSP is accessed.

- Jasper is Tomcat's JSP Engine.
- Jasper parses JSP files to compile them into Java code as servlets (that can be handled by Catalina).
- At runtime, Jasper detects changes to JSP files and recompiles them.

JSP Life Cycle

After the JSP page is created, on the first request, the server does the following:

1) translates the JSP into a servlet;
2) compiles the servlet;
3) instantiates the servlet;
4) initializes the servlet (init method);
5) Invokes the doGet method or equivalent.

JSP Life Cycle

On the second request for the same JSP, the server does the following:

1) ~~translates the JSP into a servlet;~~
2) ~~compiles the servlet;~~
3) ~~instantiates the servlet;~~
4) ~~initializes the servlet (init method);~~
5) Invokes the doGet method or equivalent.

JSP Life Cycle

If the server has been restarted, on the request for the same JSP, it does the following:

1) ~~translates the JSP into a servlet;~~
2) ~~compiles the servlet;~~
3) instantiates the servlet;
4) initializes the servlet (init method);
5) Invokes the doGet method or equivalent.

JSP Life Cycle

If the JSP page has been modified, on the request to it, the server does the following:

1) translates the JSP into a servlet;
2) compiles the servlet;
3) instantiates the servlet;
4) initializes the servlet (init method);
5) Invokes the doGet method or equivalent.

That is, the JSP will be compiled after changing its content (usually, on the first request to it).

JSP components
(declarations, expressions, actions, scriptlets, directives)

JSP components

- HTML Text

  <H1>XXX</H1>

  Passed through to client. Turned into servlet code that looks like out.print("<H1>XXX</H1>");
- HTML Comments

  <!-- Comment -->

  Same as other HTML: passed through to client
- JSP Comments

  <%-- Comment --%>

  Not sent to client
- Escaping <%

  To get <% in output, use <\%

JSP components

Scripting elements:

- JSP Expressions
  - Format: <%= expression %>
  - Wrapped in out.print and inserted into _jspService
- JSP Scriptlets
  - Format: <% code %>
  - Inserted into the servlet's _jspService method
- JSP Declarations
  - Format: <%! code %>
  - Inserted into the body of the servlet class

```
<h1>text</h1>
<%!String x = "123";%>
<%!String getX() {return x;} %>
<%!void m() {x = "987";} %>
<% m(); %>
<%= getX() %>
```

text

987

JSP components

- JSP actions are XML tags that invoke built-in web server functionality. They are executed at runtime.
- JSP directives control how the JSP compiler generates the servlet (include, page, taglib).
    - The include directive informs the JSP compiler to include a complete file into the current file.
      <%@ include file="somefile.jspf" %>
    - The page directive has several attributes (import, contentType, pageEncoding, taglib, etc.)
      <%@ page import="java.util.*" %>
      <%@ page contentType="text/html" %>
      <%@ page contentType="MIME-TYPE; charset=Encoding" %>
    - The taglib directive indicates that a JSP tag library is to be used.
      <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

JSP components: actions

jsp:useBean

- Is used to locate or instantiate a Java bean class. If bean object of the Bean class is already created, it doesn't create the bean depending on the scope. But if object of bean is not created, it instantiates the bean.

  <jsp:useBean id="beanName" class="pack.class" scope="SCOPE"/>

  id: is used to identify the bean in the specified scope.

  class: instantiates the specified bean class (i.e. creates an object of the bean class) but it must have no-arg constructor and must not be abstract.

  scope: represents the scope of the bean (page - default, request, session or application).

- The body of a <jsp:useBean> element often contains a <jsp:setProperty> element that sets property values in the bean. The body tags are only processed if <jsp:useBean> instantiates the bean. If the bean already exists and <jsp:useBean> locates it, the body tags have no effect.

JSP components: actions

jsp:useBean

```
<jsp:useBean id="loginForm" class="my.LoginForm">
    <jsp:setProperty name="loginForm" property="login" value="LOGIN"/>
    <jsp:setProperty name="loginForm" property="pass" value="PASS"/>
</jsp:useBean>
```

jsp:setProperty

```
<jsp:setProperty name="loginForm" property="login" value="LOGIN"/>
<jsp:setProperty name="loginForm" property="login" param="PARAM_NAME"/>
<jsp:setProperty name="loginForm" property="*" />
```

jsp:getProperty

```
<jsp:getProperty name="loginForm" property="login" />
```

JSP components: actions

jsp:include

Like the include directive, this tag includes a specified jsp into the returned HTML page but it works differently. The Java servlet temporarily hands the request and response off to the specified JavaServer Page. Control will then return to the current JSP, once the other JSP has finished.

```
...
  <body>
    <jsp:include page="mycommon.jsp" >
      <jsp:param name="extraparam" value="myvalue" />
    </jsp:include>
    name:<%=request.getParameter("extraparam")%>
  </body>
</html>
```

JSP components: actions

jsp:param

Can be used inside a jsp:include, jsp:forward or jsp:params block. Specifies a parameter that will be added to the request's current parameters.

jsp:forward

Used to hand off the request and response to another JSP or servlet. Control will never return to the current JSP.

```
<jsp:forward page="subpage.jsp" >
  <jsp:param name="forwardedFrom" value="this.jsp" />
</jsp:forward>
```

JSP components

- Limit the Java code that is directly in page.
- Use helper classes, beans, servlet/JSP combo (MVC), JSP expression language, custom tags.
- XML Syntax. There is alternative JSP syntax that is sometimes useful when generating XML-compliant documents, probably for Ajax apps. But is more trouble than it is worth for most HTML applications.

JSP components: page01.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!doctype html>
<html>
<head>
<title>JSP (Classic Syntax)</title>
</head>
<body bgcolor="#cdffcd">
    <h1>Sample (Classic Syntax)</h1>
    <h2>Num1: <%=Math.random() * 10%></h2>
    <% double num2 = Math.random() * 100; %>
    <h2>Num2: <%=num2%></h2>
    <%! private double num3 = Math.random() * 1000; %>
    <h2>Num3: <%=num3%></h2>
</body>
</html>
```

JSP components: page02.jspx

```
<?xml version="1.0" encoding="UTF-8" ?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page">
...
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>JSP (XML Syntax)</title>
</head>
<body bgcolor="#ffcdcd">
   <h1>Sample (XML Syntax)</h1>
   <h2> Num1: <jsp:expression>Math.random() * 10</jsp:expression></h2>
   <jsp:scriptlet>double num2 = Math.random() * 100;</jsp:scriptlet>
   <h2> Num2: <jsp:expression>num2</jsp:expression></h2>
   <jsp:declaration>private double num3 = Math.random() * 1000;</jsp:declaration>
   <h2> Num3: <jsp:expression>num3</jsp:expression></h2>
</body>
</html>
</jsp:root>
```

JSP components: http://localhost:8080/.../page01.jsp; ...page02.jspx

Sample (Classic Syntax)

Num1: 3.394818465047491

Num2: 5.756544486226711

Num3: 316.0124508002622

Sample (XML Syntax)

Num1: 2.027283322654342

Num2: 1.520581380842256 4

Num3: 736.9996983881256

JSP: implicit objects

JSP: Implicit objects

There are declared objects that can be used by the programmer.

- JspWriter **out**: The JspWriter used to write the data to the response stream.
- Object **page**: The servlet itself.
- PageContext **pageContext**: A PageContext instance that contains data associated with the whole page. A given HTML page may be passed among multiple JSPs.
- HttpServletRequest **request**: The HttpServletRequest object that provides HTTP request information.
- HttpServletResponse **response**: The HttpServletResponse object that can be used to send data back to the client.
- HttpSession **session**: The HttpSession object that can be used to track information about a user from one request to another.
- ServletConfig **config**: Provides servlet configuration data.
- ServletContext **application**: Data shared by all JSPs and servlets in the application.
- Throwable **exception**: Exceptions not caught by application code.

JSP: Implicit objects

```
<%
out.print("<br> Today is:"+java.util.Calendar.getInstance().getTime());
out.print("<br>Hello " + request.getParameter("user"));
session.setAttribute("name", "John");
out.print("<br>name: " + session.getAttribute("name"));
out.print("<br>init_param: " + config.getInitParameter("init_param"));
out.print("<br>context_param: "
      + application.getInitParameter("context_param"));
%>
```

```
<servlet>
    <servlet-name>page</servlet-name>
    <jsp-file>/page.jsp</jsp-file>

    <init-param>
        <param-name>init_param</param-name>
        <param-value>Init param value</param-value>
    </init-param>

</servlet>

<servlet-mapping>
    <servlet-name>page</servlet-name>
    <url-pattern>/x</url-pattern>
</servlet-mapping>

<context-param>
    <param-name>context_param</param-name>
    <param-value>Context param value</param-value>
</context-param>
```

```
<% response.sendRedirect("http://www.google.com");  %>
```

Error page
(configuration in web.xml)

Error page

You can define custom error handling using a web.xml file descriptor:

- Status code error handling – it allows you to map HTTP error codes to a static HTML error page or an error handling servlet.
- Exception type error handling – it allows you to map exception types to static HTML error pages or an error handling servlet

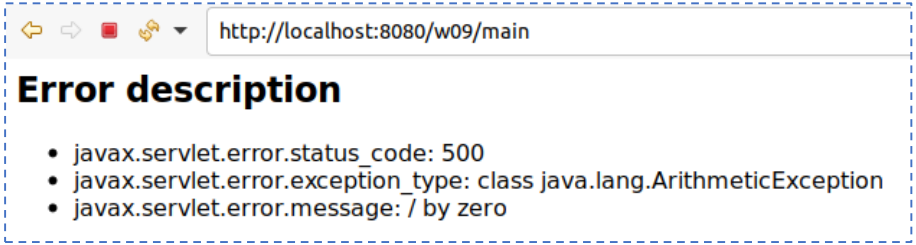
```
<error-page>
    <error-code>404</error-code>
    <location>/error-404.html</location>
 </error-page>
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/error</location>
</error-page>
```

Error page

```
@WebServlet("/error")
public class Error extends HttpServlet {
…
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
        resp.setContentType("text/html; charset=utf-8");
        try (PrintWriter writer = resp.getWriter()) {
            writer.write("<html><head><title>Error description</title></head><body>");
            writer.write("<h2>Error description</h2>");
            writer.write("<ul>");
            Arrays.asList(ERROR_STATUS_CODE, ERROR_EXCEPTION_TYPE, ERROR_MESSAGE)
                    .forEach(e -> writer.write("<li>" + e + ":" + req.getAttribute(e) + " </li>"));
            writer.write("</ul>");
            writer.write("</html></body>");
        }
    }
}
```

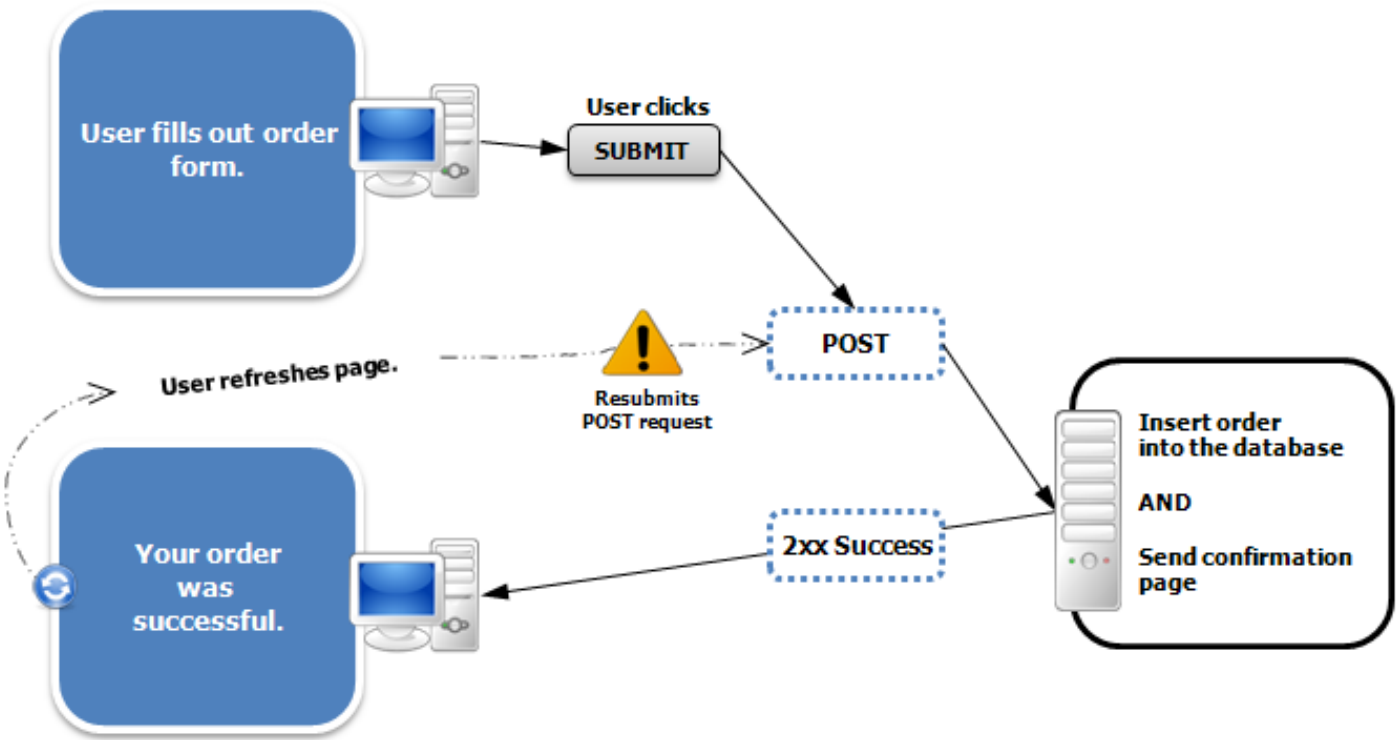
Error page

```
@WebServlet("/main")
public class Main extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doGet(HttpServletRequest request, HttpServletResponse response)
         throws ServletException, IOException {
      int a = 0;
      int b = 1 / a;
      // throw new ServletException("XXX");
   }

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

}
```

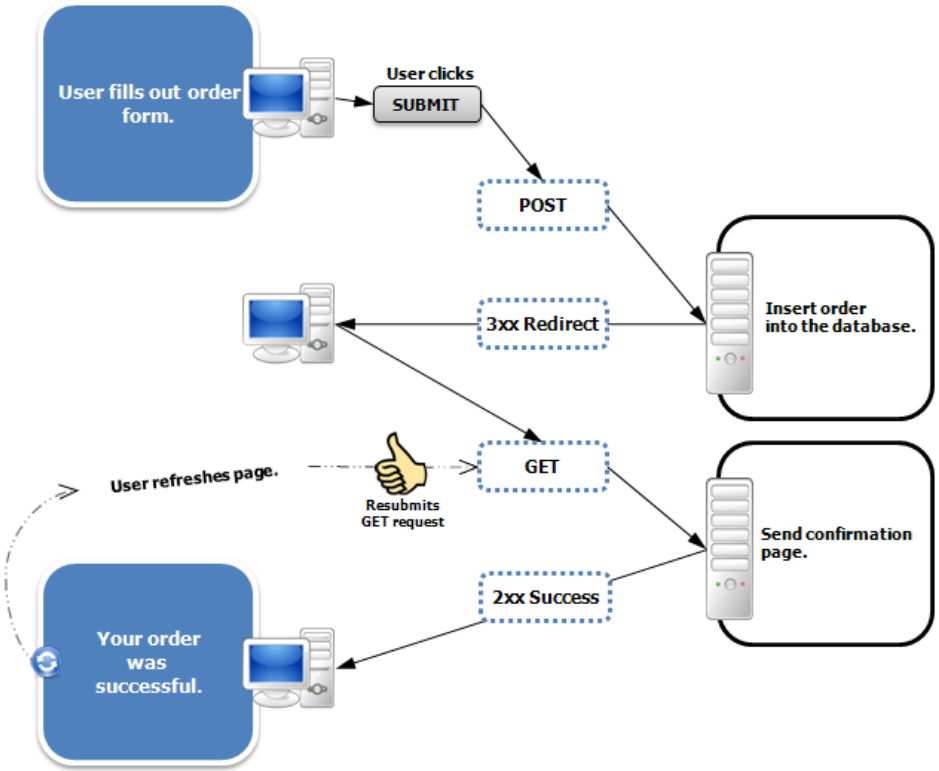


PRG pattern (Post/Redirect/Get)

PRG pattern (Post/Redirect/Get)



PRG pattern (Post/Redirect/Get)



JSTL

- EL (IMPLICIT OBJECTS, BASIC OPERATIONS).
- TAG LIBRARIES.
- CUSTOM TAGS (TAG FILES).

EL (implicit objects, basic operations).

EL

- EL - Expression Language
- Syntax
  - **${…}** **${obj.something.anotherSomething}**
- Dynamically typed language
- EL is best suited to work with objects, adhering notation JavaBean
- The EL identifier refers to the variable returned by calling
  - **PageContext.findAttribute (name)**
- A variable can be stored in any scope:
  - **page(PageContext)**
  - **request(HttpServletRequest)**
  - **session (HttpSession)**
  - **application (ServletContext)**
- If the variable is not found, **null** is returned
- Access the request parameters through predefined object **paramValues**

EL

- EL literals:
  - true, false
  - decimal integer, floating point, scientific-notation numeric literals
  - strings (single- or double-quoted)
  - null

EL

- EL variable names
  - like Java
  - Can contain letters, digits, _ , and $
  - Must not begin with a digit
  - Must not be reserved:

```
and   div   empty eq    false ge    gt    instanceof
le    lt    mod   ne    not   null  or    true
```

EL

- EL operators:
  - Relational: **<, >, <=, >=, ==, !=** (or: **lt, gt, le, ge, eq, ne**)
  - Logical: **&&, ||, !** (equivalents: **and, or, not**)
  - Arithmetic:
    - **+, -** (binary and unary), ***, /, %** (or **div, mod**)
  - **empty**: true if arg is null or empty string/array/Map/Collection
  - Conditional: **? :**
  - Array access: **[ ]** (or object notation)
  - Parentheses for grouping **()**

EL

- EL automatic type conversion
  - Conversion for + is like other binary arithmetic operators (+ does not represent string concatenation)
  - Otherwise similar to JavaScript

Tag libraries.

JSTL

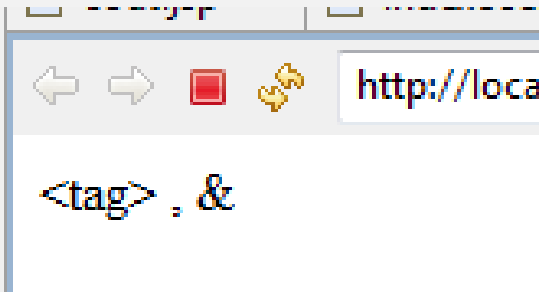
- JSTL - JSP Standard Tag Libraries
- Is a collection of JSP custom tags
- Allows to program JSP pages using tags, rather than the scriptlet code
- Can do nearly everything that regular JSP scriptlet code can do
- JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization tags, and SQL tags. It also provides a framework for integrating existing custom tags with JSTL tags
- JSTL tag library groups:
    - Core Tags
    - Formatting tags
    - SQL tags
    - XML tags
    - JSTL Functions

JSTL

- Core Tag Library—Contains tags that are essential to nearly any Web application. Examples of core tag libraries include looping, expression evaluation, and basic input and output.
- Formatting/Internationalization Tag Library—Contains tags that are used to parse data. Some of these tags will parse data, such as dates, differently based on the current locale.
- Database Tag Library—Contains tags that can be used to access SQL databases. These tags are normally used only to create prototype programs.
- XML Tag Library—Contains tags that can be used to access XML elements.
- JSTL Functions --JSTL includes a number of standard functions, most of which are common string manipulation functions.

JSTL

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
          pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>c:out Tag Example</title>
</head>
<body>
          <c:out value="${'<tag> , &'}" default="deftest" />
</body>
</html>
```



JSTL

- Core Tag Library—prefix c
- <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<c:out >	<c:set >	<c:remove >
<c:catch>	<c:if>	<c:choose>
<c:when>	<c:otherwise >	<c:import>
<c:forEach >	<c:forTokens>	<c:param>
<c:redirect >	<c:url>	

JSTL

- Formatting Tag Library—prefix fmt
- <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<fmt:formatNumber>	<fmt:parseNumber>	<fmt:formatDate>
<fmt:parseDate>	<fmt:bundle>	<fmt:setLocale>
<fmt:setBundle>	<fmt:timeZone>	<fmt:setTimeZone>
<fmt:message>	<fmt:requestEncoding>	

JSTL

- SQL Tag Library—prefix sql
- <%@ taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql" %>

<sql:setDataSource>	<sql:query>
<sql:update>	<sql:param>
<sql:dateParam>	<sql:transaction >

JSTL

- XML Tag Library—prefix x
- <%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>

<x:out>	<x:parse>	<x:set >
<x:if >	<x:forEach>	<x:choose>
<x:when >	<x:otherwise >	<x:transform >
<x:param >		

JSTL

- JSTL Functions Tag Library—prefix fn
- <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

fn:contains()	fn:containsIgnoreCase()	fn:endsWith()
fn:escapeXml()	fn:indexOf()	fn:join()
fn:length()	fn:replace()	fn:split()
fn:startsWith()	fn:substring()	fn:substringAfter()
fn:substringBefore()	fn:toLowerCase()	fn:toUpperCase()
fn:trim()		

Custom tags (tag files).

Custom tags

- Custom tags are developed in Java and defined and used with XML syntax
- Tags are used in a JSP to reduce or constrain the amount of Java scriptlets in the page
- Tags are useful for defining custom actions
- Collections of tags are grouped into JAR files called Tag Libraries
- Custom Tag Libraries contain:
    - One or more tag handler class files
    - May contain additional supporting classes
    - A tag library descriptor (**`taglib.tld`**) (XML formatted)
- To use a tag in a JSP, perform the following:
    - Invoke the tag library by using the **`<jsp:taglib/>`** directive.
    - Call the tag in the content of the JSP.
    - Include the location of the **`taglib.tld`** file in the **`web.xml`** file.

Custom tags

- Java class must implement the `javax.servlet.jsp.tagext.Tag` interface
- Usually extends one of the `TagSupport` or `BodyTagSupport` classes:
    - `int doStartTag()`
    - `void doInitBody()`
    - `int doAfterBody()`
    - `BodyContent getBodyContent()`
    - `void setBodyContent(BodyContent b)`
    - `int doEndTag()`

    - `SKIP_BODY, EVAL_BODY_INCLUDE, EVAL_BODY_BUFFERED, EVAL_BODY_AGAIN`

Custom tags

- A tag library descriptor (.tld) is an XML document that describes one or more tags and their attributes. It contains the following elements:
  - tlib-version — The tag library's version
  - short-name — A default name for the library
  - uri — Identifies the tag library location
  - info — Documentation regarding the library

- <tag> elements:
  - name — Defines the name of the base tag
  - tag-class — Full name of the tag handler class
  - info — Short description of the tag
  - tagdependent — **empty** or **jsp** or **tagdependent** value.

Custom tags

```
<?xml version="1.0" encoding="utf-8" ?>
<taglib xmlns="http://java.sun.com/JSP/TagLibraryDescriptor"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/ web-jsptaglibrary_2_0.xsd"
          version="2.0"><!--дескриптор библиотеки тегов -->
          <tlib-version> 1.0 </tlib-version>
          <short-name> mytag </short-name>
          <uri> /WEB-INF/tld/mytaglib.tld </uri>
          <tag>
                    <name> getinfo </name>
                    <tag-class> _java._ee._02.simpletag.GetInfoTag </tag-class>
                    <body-content> empty </body-content>
          </tag>
</taglib>
```

Custom tags

- Specify access to file (.tld) in the web.xml file, for which you should specify after <welcome-file-list>

```
<jsp-config>
        <taglib>
                <taglib-uri>
                        /WEB-INF/tld/mytaglib.tld
                </taglib-uri>

                <taglib-location>
                        /WEB-INF/tld/mytaglib.tld
                </taglib-location>
        </taglib>
</jsp-config>
```

- Register the URI of the library in the description file (.tld) of the library and place this file in the / WEB-INF folder of the project.

Custom tags

```
<?xml version="1.0" encoding="utf-8" ?>
<taglib xmlns="http://java.sun.com/JSP/TagLibraryDescriptor"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee/ web-jsptaglibrary_2_0.xsd"
          version="2.0">
          <tlib-version> 1.0 </tlib-version>
          <short-name> mytag </short-name>
          <uri> /WEB-INF/tld/mytaglib.tld </uri>
          <tag>
                    <name> bodyattr </name>
                    <tag-class> web_jstl.BodyTag </tag-class>
                    <body-content> JSP </body-content>
                    <attribute>
                              <name> num </name>
                              <required> false </required>
                              <rtexprvalue> true </rtexprvalue>
                    </attribute>
          </tag>
</taglib>
```

Custom tags

- Using a Custom Tag:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/tld/mytaglib.tld" prefix="mytag"%>
<html>
<body>
        <mytag:simple />
        <mytag:bodyattr> Tag text </mytag:bodyattr>
        <mytag:bodyattr attr=“param"> Tag text </mytag:bodyattr>
</body>
</html>
```

Custom tags

- Tag with attributes
- For each attribute of a tag, the class that implements it must contain the **`setAttribute`*`Name`*`()`** method
- Describe a tag with attributes in a * .tld file, attributes are used, which must be declared inside the tag element using the attribute element
- Inside the attribute element, between the <attribute> and </attribute> tags, there may be the following elements:
  - name — attribute name (required element)
  - required — this attribute must be present when using the tag, or not true or false (required element)
  - rtexprvalue — the attribute value is a JSP expression ${expr} or <% = expr%> (**`true`**) or specified as a string (**`false`**). The default is **`false`** (optional)

Custom tags

Body tag:

- the handler class must extends **`BodyTagSupport`** class
- **`doStartTag()`** method must return **`EVAL_BODY_INCLUDE`** or **`EVAL_BODY_BUFFERED`**
- if it returns **`SKIP_BODY`**, then the **`doInitBody()`** method is not called
- the content of the body-content element:
  - **`empty`** empty body
  - **`jsp`** the body consists of everything that can be in the JSP file
  - **`tagdependent`** the body is interpreted by the handler class

List of references

1) https://tomcat.apache.org/taglibs.html
2) https://www.tutorialspoint.com/jsp/jsp_expression_language.htm
3) https://www.tutorialspoint.com/jsp/jsp_standard_tag_library.htm
4) https://www.tutorialspoint.com/jsp/jsp_custom_tags.htm