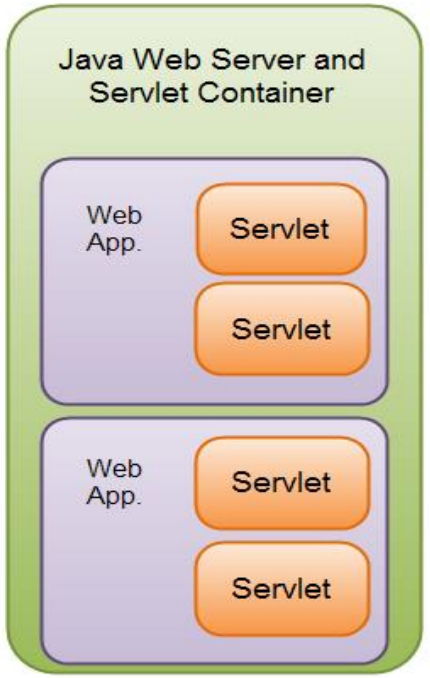
Java EE

- **Servlet overview & Servlet Life Cycle**
- **Servlet Interface**
  - **Implementations:**
    - **GenericServlet & HttpServlet**
- **MIME Types**
- **Creating WAR file**
- **Processing HttpRequest & HttpResponse Headers**
- **Handling form data**
- **ServletContext & ServletConfig objects**

Servlet Overview

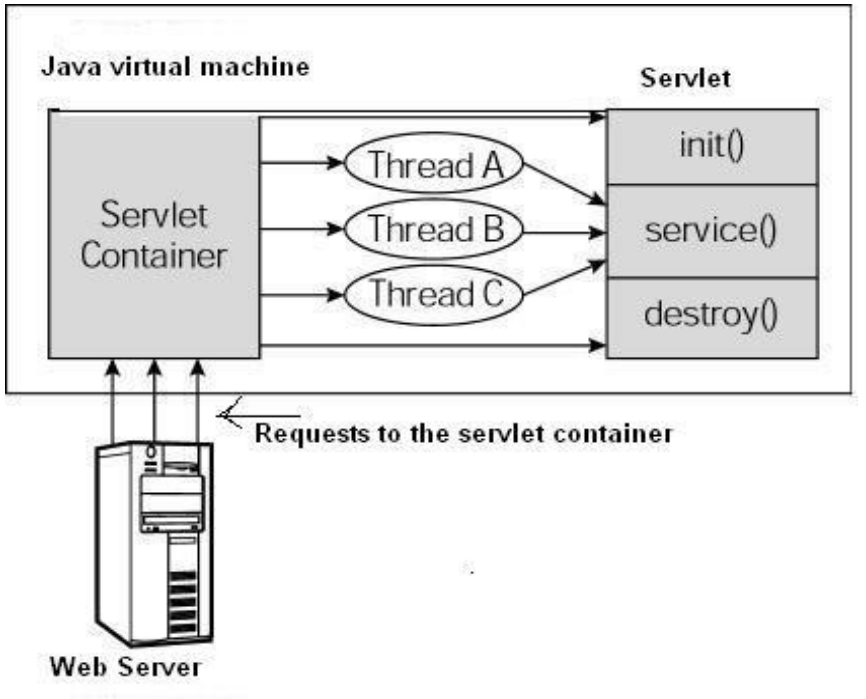
- Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.
- For such applications, Java Servlet technology defines **HTTP-specific servlet classes.**
- The ***javax.servlet*** and ***javax.servlet.http*** packages provide interfaces and classes for writing servlets.
- All servlets must implement the ***Servlet*** interface, which defines lifecycle methods.
- When implementing a generic service, you can use or extend the ***GenericServlet*** class provided with the Java Servlet API.
- The ***HttpServlet*** class provides methods, such as ***doGet()*** and ***doPost()*** , for handling HTTP-specific services.
- A Servlet is one of the main web components of a Java web application.
- A Servlet container may run multiple web applications and each web application may have multiple servlets running inside.



Servlet Lifecycle

The lifecycle of a servlet is controlled by the container in which the servlet has been deployed.

When a request is mapped to a servlet, the container performs the following steps.

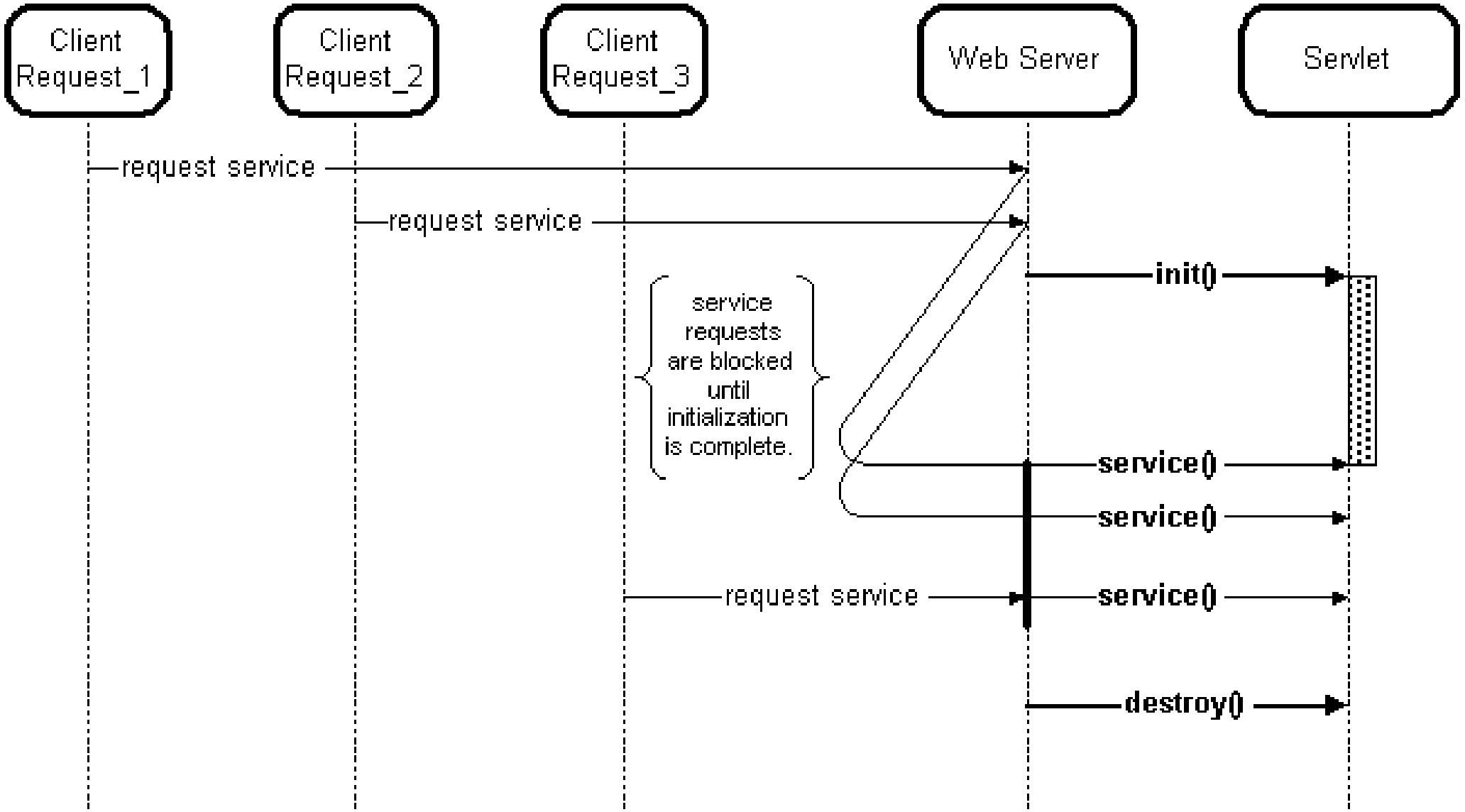


1.If an instance of the servlet does not exist, the web container

- Loads the servlet class.
- Creates an instance of the servlet class.
- Initializes the servlet instance by calling the *init()* method.
- Spawns a new thread , hands over the control to the thread
- This thread invokes the service method, passing request and response objects.
- In case your servlet extends HttpServlet, the service() method which receives the request will pass on to *doGet() or doPost()* methods depending on method type.

2. If it needs to remove the servlet, the container finalizes the servlet by calling the servlet's destroy method.

Basic interactions between clients, a web server, and a servlet registered with the web server



Servlet API

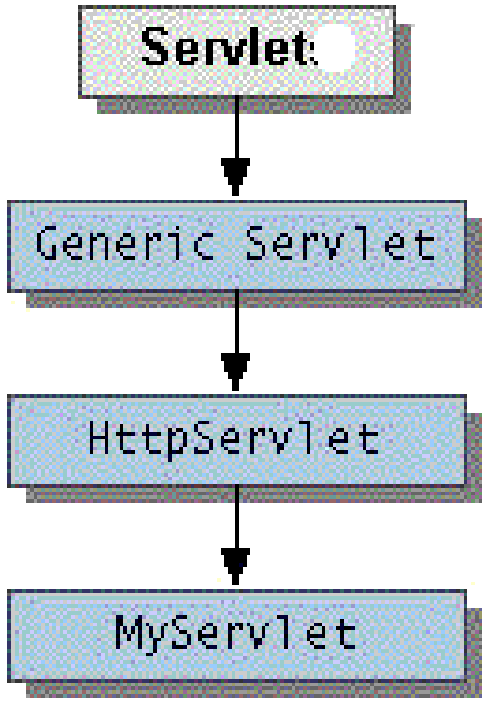
Servlet Lifecycle Methods

The central abstraction in the Servlet API is the Servlet interface.

All servlets implement this interface, either directly or indirectly, more commonly, by extending a class that implements it such as HttpServlet.

The Servlet interface contains **abstract methods** that define the servlet lifecycle.

Rarely servlets directly implement Servlet interface.



*The **Servlet interface** provides the following methods that manage the servlet and its communications with clients.*

- **destroy()** : Cleans up whatever resources are being held and makes sure that any persistent state is synchronized with the servlet's current in-memory state.
- **getServletConfig() :** Returns a **servlet config object**, which contains any initialization parameters and startup configuration for this servlet.
- **getServletInfo()** :Returns a string containing information about the servlet, such as its author, version, and copyright.
- **init(ServletConfig) :** Initializes the servlet. Run once before any requests can be serviced.
- **service(ServletRequest, ServletResponse) :** Carries out a single request from the client.

GenericServlet

public abstract class **GenericServlet** implements **Servlet, ServletConfig, java.io.Serializable**

GenericServlet is a generic, protocol-independent servlet.

GenericServlet implements the **Servlet** and **ServletConfig** interfaces.

GenericServlet may be directly extended by a servlet, although it's more common to extend a protocol-specific subclass such as **HttpServlet**.

GenericServlet makes writing servlets easier. It provides simple versions of the lifecycle methods ***init() and destroy()*** and also the methods in the **ServletConfig** interface. **GenericServlet** also implements the log method, declared in the **ServletContext** interface.

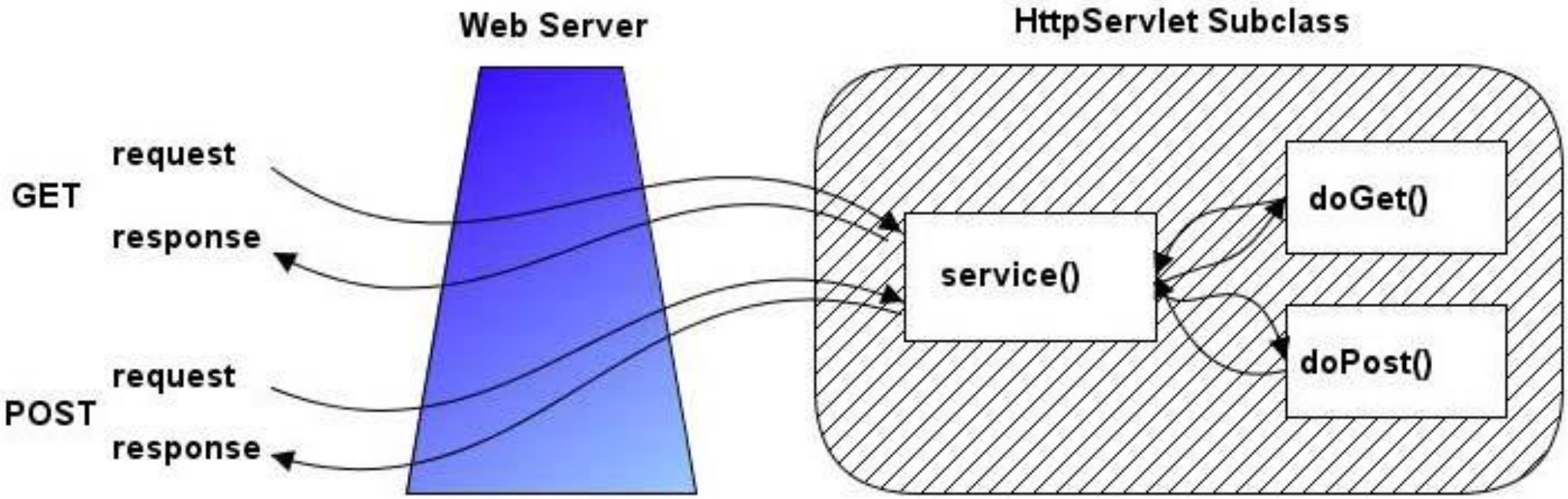
To write a generic servlet, you need only override the abstract service method.

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

javax.servlet.http.HttpServlet

public abstract class HttpServlet extends GenericServlet implements java.io.Serializable

- Defines a protocol-specific Servlet.
- **HttpServlet** extends the **GenericServlet** and hence inherits the properties GenericServlet.



doGet() and doPost() methods of HttpServlet class

The doGet() Method

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        // Servlet code
}
```

The doPost() Method

```
public void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        // Servlet code
}
```

The doGet() and doPost() are most frequently used methods within each service request. So we have nothing to do with service() method. Override either doGet() or doPost() depending on what type of request you receive from the client.

What HttpServletRequest and HttpServletResponse objects contain?

- The HTTP request *received* by the Web Server will be in the form of plain text. It will be difficult for the programmer to parse this text and get the details.
- So, the Servlet Container wraps all the information in the HttpServletRequest object.
- The servlet container creates an HttpServletRequest object and passes it as an argument to the servlet's service methods (doGet, doPost, etc).
- The Servlet can get all the details of the request by calling the methods of this object.
- Similarly, the HTTP response is also in the form of plain text.
- The HttpServletResponse object methods can be used by our servlets for creating responses.
- The Servlet Container will convert this data into the format specified by the HTTP protocol

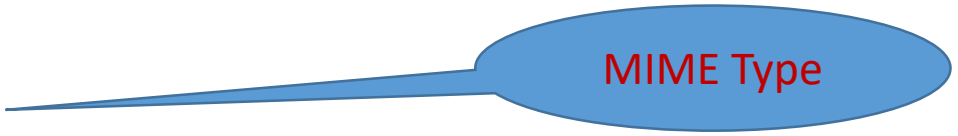
Note HttpServletRequest and HttpServletResponse are interfaces. The Servlet Container creates objects and passes them to service methods.

Creating First Servlet

```
@WebServlet(name="/MyFirstServlet",urlPatterns="/first")
public class MyFirstServlet extends HttpServlet {
private static final long serialVersionUID = 1L;
```

- Servlet 3.x provides the ***@WebServlet*** annotation to define a servlet.
- ***No need of entries in web.xml file.***

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
PrintWriter out=response.getWriter();
response.setContentType("text/html");
out.println("<html><h1><font color=\"blue\">"+"Welcome To My First
Servlet"+"</font></h1></html>");
}
```



URL

http://localhost:8080/MyFirstWebApp/first

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

Multipurpose Internet Mail Extension (MIME)

The browser can accept many types of files like html, text and different kinds of images. Before sending data to the browser, specify the type of that data so that browser can display the content properly.

Multipurpose Internet Mail Extension (MIME) is the standard that is used to specify the type of data to the browser.

MIME standard specifies a string for each data type –

- "text/html" for HTML data,
- "text/plain" for plain text data
- "image/gif" for gif images etc.

The method ***setContentType(String mime)*** can be used for sending the MIME to the browser.
For ex. ***response.setContentType("text/html");***

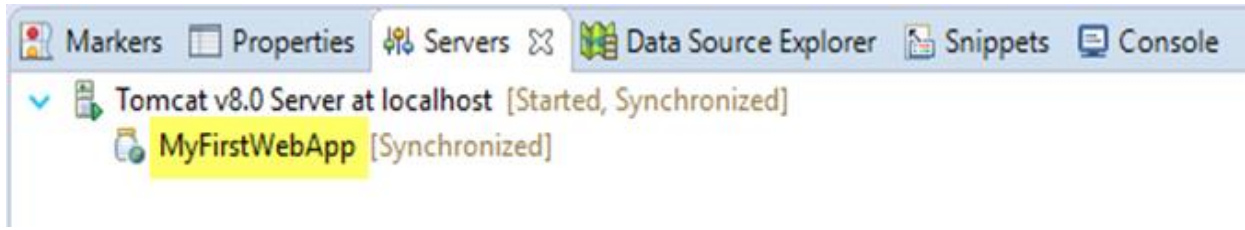
Deployment Descriptor file (web.xml)

Without Annotation (@WebServlet)

```
....
  <servlet>
    <servlet-name>comingsoon</servlet-name>
    <servlet-class>packagename.servletname</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>comingsoon</servlet-name>
    <url-pattern>/pattern</url-pattern>
  </servlet-mapping>
.....
```

HTTP Request-Response Model

1. When a HTTP client requests for a resource, the HTTP Server will check whether the ***context root*** specified in the URL is currently deployed in the server.



2. If found, then it will look into the web component of the application with *urlPattern* matching with the pattern provided in the URL.

```
@WebServlet(name="/MyFirstServlet",urlPatterns="/first")
public class MyFirstServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
```

3. The servlet container performs the following tasks:

- It creates an instance of the servlet and calls its init() method to initialize it.
- It **constructs a request object** that is passed to the servlet via service() method. The request includes, among other things:
  - Any HTTP headers from the client.
  - Parameters and values passed from the client (for example, names and values of query strings in the URL)
  - The complete URI of the servlet request.
- It **constructs a response object** for the servlet.
- It invokes the servlet service() method. Note that for HTTP servlets, the generic service method is usually overridden in the HttpServlet class methods, doGet() or doPost() methods, depending on the HTTP request type. (GET or POST)

Creating & Deploying Web Application Archive (WAR) file

1. **Right-Click on the project name(context-root) , Click on Export->WAR file**

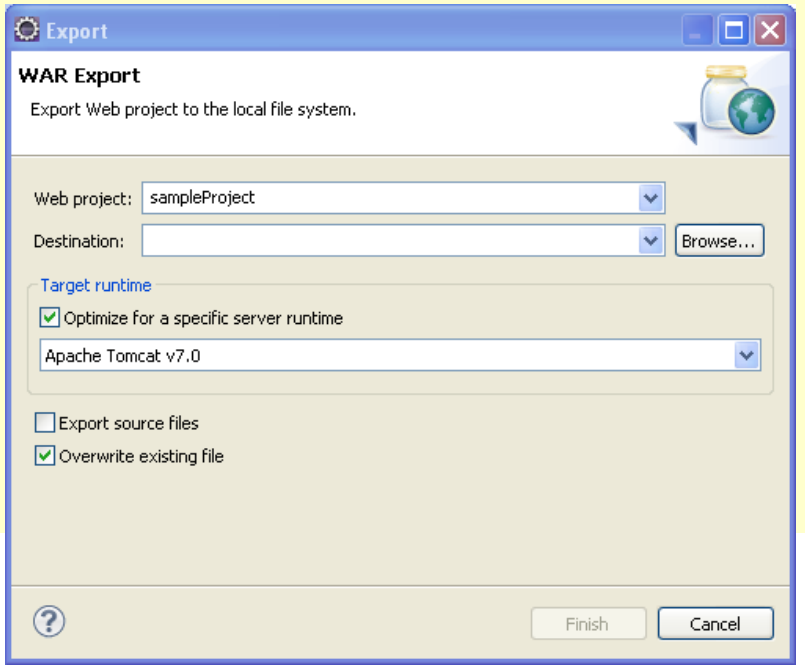
2. Type war file name, *ex. sample.war* and store in your file system.(Select the folder by clicking on **Browse** button and click on **Finish** button.

3. This war file can be deployed an any other Tomcat Server. We can make use of Tomcat Manager console to deploy the web application).

4. Click on **Manager App** button that you see on the Tomcat Home Page.

- Enter user-id and password. You require *manager-gui role* to access this page
- Edit the file, *tomcat-users* which is located in *conf* folder of tomcat installation directory:
  - *Ex. C:\Program Files\apache-tomcat-8.0.24\conf*
- Remove the comments surrounding the *roles* and enter the following:

```
<role rolename="tomcat"/>
<role rolename="role1"/>
<role rolename="manager-gui"/>
<user username="tomcat" password= " tomcat" roles="manager-gui"/>
<user username="both" password="tomcat" roles="tomcat,role1"/>
<user username="role1" password="tomcat" roles="role1"/>
```



HTTP Request & Response Header Methods

HTTP Request Header Methods

Reading Request Headers
(Important methods of HttpServletRequest interface)

- **General**
  - getHeader (parameter name)
  - getHeaderNames()

- **Specialized**
  - getCookies()
  - getAuthType() and getRemoteUser()
  - getContentLength()
  - getContentType()
  - getDateHeader()
  - getIntHeader()

- **Related info**
  - getMethod(), getRequestURI() ,
  - getQueryString(), getProtocol()

HTTP Response Header Methods

- HttpServletResponse is a predefined interface present in javax.servlet.http package.
- The response object is where the servlet can write information about the data it will send back.
- The majority of the methods in the request object start with GET, indicating that they get a value, many of the important methods in the response object start with SET, indicating that they change some property.

i. public String encodeRedirectURL(String url) :Encodes the specified URL for use in the sendRedirect method or, if encoding is not needed, returns the URL unchanged.
ii. public String encodeURL(String url) :Encodes the specified URL by including the session ID in it, or, if encoding is not needed, returns the URL unchanged.
iii. public void sendError(int *sc)* throws IOException :Sends an error response to the client using the specified status code and clearing the buffer.
iv. public void sendRedirect(string location) throws IOException : Sends a temporary redirect response to the client using the specified redirect location URL.
v. public void setHeader(String name, String value) : Sets a response header with the given name and value.
vi. public void setIntHeader (String name,int value) :Sets a response header with the given name and integer value.
vii. public void setStatus(int sc) :Sets the status code for this response.

Handling Form Data

Sending Form Data

In many cases, the purpose of an HTML Form is to send data to a server.
The server processes the data and then sends a response to the user.

On the client side: defining how to send the data

The <form> element defines how the data will be sent. All of its attributes are designed to let you configure the request to be sent when a user hits a submit button.

The two most important attributes are action and method.

1. The action attribute:

This attribute defines where the data gets sent. Its value must be a valid URL. If this attribute isn't provided, the data will be sent to the URL of the page containing the form.

Examples

In this example, the data is sent to http://foo.com:

```
<form action="http://foo.com">
```

Here, the data is sent to the same server that hosts the form's page, but to a different URL on the server:

```
<form action="/somewhere_else">
```

Sending Form Data

2. The method attribute

This attribute defines how data is sent.
The HTTP protocol provides several ways to perform a request; HTML forms data can be sent through at least two of them: **the GET method and the POST method.**

GET method

The GET method sends the encoded user information appended to the page request.

The page and the encoded information are separated by the ? character as follows:

http://www.test.com/hello?key1=value1&key2=value2

The GET method is the default method to pass information from browser to web server and it produces a long string that appears in your browser's location box.

Servlet handles this type of requests using **doGet()** method.

POST Method

POST method

A generally more reliable method of passing information to a backend program is the POST method.

This packages the information in exactly the same way as GET methods, but instead of sending it as a text string after a ? in the URL, the data is appended to the body of the HTTP request.

This message comes to the backend program in the form of the standard input which you can parse and use for your processing.

Servlet handles this type of requests using **doPost()** method.

HTTP has 8 methods – GET, POST, PUT, HEAD, DELETE ,OPTIONS , TRACE and CONNECT.

HttpServlet has methods corresponding to all these methods – doGet, doPost, doHead, doDelete etc.

The **doGet()** and **doPost()** methods are the most commonly used methods as GET and POST are the commonly used HTTP methods

GET and POST Methods

	GET (HTTP)	POST(HTTP)
History	Parameters remain in browser history because they are part of the URL	Parameters are not saved in browser history.
Bookmarked	Can be bookmarked.	Can not be bookmarked.
BACK button/re-submit behaviour	GET requests are re-executed but may not be re-submitted to server if the HTML is stored in the browser cache.	The browser usually alerts the user that data will need to be re-submitted.
Encoding type (enctype attribute)	application/x-www-form-urlencoded	multipart/form-data or application/x-www-form-urlencoded. Use multipart encoding for binary data.
Parameters	can send parameters but the parameter data is limited to what we can stuff into the request line (URL). Safest to use less than 2K of parameters, some servers handle up to 64K	Can send parameters, including uploading files, to the server.

GET and POST Methods

	GET (HTTP)	POST(HTTP)
Restrictions on form data type	Yes, only ASCII characters allowed.	No restrictions. Binary data is also allowed.
Security	GET is less secure compared to POST because data sent is part of the URL. So it's saved in browser history and server logs in plaintext.	POST is a little safer than GET because the parameters are not stored in browser history or in web server logs.
Restrictions on form data length	Yes, since form data is in the URL and URL length is restricted. A safe URL length limit is often 2048 characters but varies by browser and web server.	No restrictions
Usability	GET method should not be used when sending passwords or other sensitive information.	POST method used when sending passwords or other sensitive information.
Visibility	GET method is visible to everyone (it will be displayed in the browser's address bar) and has limits on the amount of information to send.	POST method variables are not displayed in the URL.

Creating and Processing HTML Forms

1. Use the FORM element to create an HTML form. Use the
ACTION attribute to designate the address of the servlet or JSP page
that will process the results; you can use an absolute or relative URL.
For example:
<FORM ACTION="...">...</FORM>
If ACTION is omitted, the data is submitted to the URL of the current
page.

2. Use input elements to collect user data.
Place the elements between the start and end tags of the FORM element and give each input element a NAME.

Text fields are the most common input element;
they are created with the following.
<INPUT TYPE="TEXT" NAME="...“>
3. Place a submit button near the bottom of the form.
For example:
<INPUT TYPE="SUBMIT">
When button is pressed, the URL designated by the form’s ACTION is invoked.

On the server side: retrieving the data

Reading form data in a Servlet

Servlets handles form data parsing automatically using the following methods depending on the situation:

request.getParameter() : Call *request.getParameter()* method to get the value of a form parameter.

Note: *The getParameter() method return a value of type, String.*

request.getParameterValues() : Call this method if the parameter appears more than once and returns multiple values, for example checkbox.

Note : This method returns all the values of the named parameter as an ***array of String.***

request.getParameterNames() : Call this method to read complete list of all parameters in the current request.

Note :The method getParameterNames() of ServletRequest returns the names of all parameters that are passed to the Servlet as an ***Enumeration of String objects****.*

request.getParameterMap()

– Returns Map of request parameters

Note : The arguments to *getParameter()* and *getParameterValues()* are case sensitive.

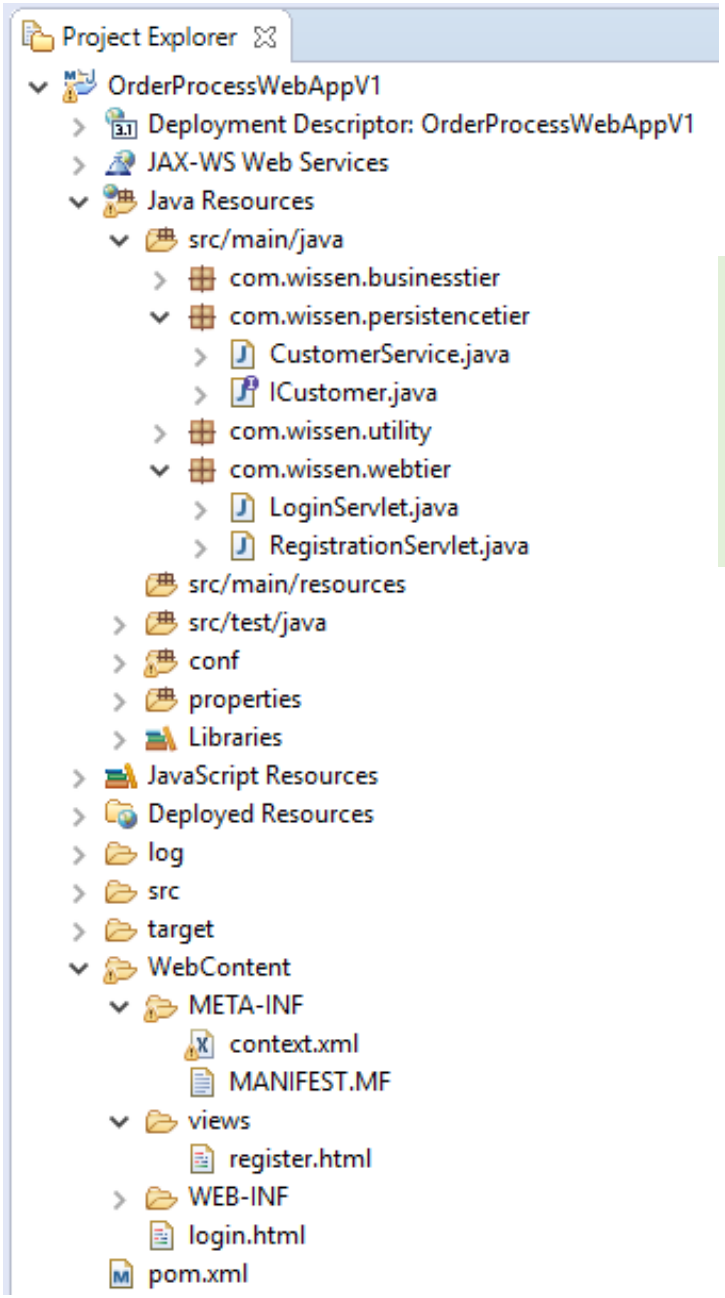
Servlet receiving form data

Example:

```
Enumeration paramNames = request.getParameterNames();
while(paramNames.hasMoreElements()) {
  String paramName = (String)paramNames.nextElement();

  String[] paramValues = request.getParameterValues(paramName);
  if (paramValues.length == 1) {
        out.println(paramValues[0]);
  } else {
        for(int i=0; i < paramValues.length; i++) {
          out.println(paramValues[i]);
         }
        }
}
```

Login and Registration Implementation



Take help of built-in login screens that use bootstrap and css.

Link:
http://bootsnipp.com/snippets/featured/clean-modal-login-form

Setting up database connection pool

JNDI stands for Java Naming and Directory Interface.

JNDI allows distributed applications to look up services in an abstract, resource-independent way.

The most common use case is to set up a database connection pool on a Java EE application server. Any application that's deployed on that server can gain access to the connections they need using the JNDI name java:comp/env/*<databasename>* without having to know the details about the connection.

This has several advantages:

1. If you have a deployment sequence where apps move from dev->int->test->prod environments, you can use the same JNDI name in each environment and hide the actual database being used. Applications don't have to change as they migrate between environments.
2. You can minimize the number of folks who need to know the credentials for accessing a production database. Only the Java EE app server needs to know if you use JNDI.

Tomcat provides a JNDI **InitialContext** implementation instance for each web application running under it, in a manner that is compatible with those provided by a Java Enterprise Edition application server.

Setting up database connection pool

Create file, ***context.xml*** with the following contents and place in in **WebContent/META-INF** folder

MySQL

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
   <!-- Specify a JDBC datasource -->
   <Resource name="jdbc/orderprocessDB" auth="Container"
      type="javax.sql.DataSource" username="root" password="root"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost:3306/orderprocess"
      maxActive="10" maxIdle="4" />
</Context>
```

Oracle

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
   <!-- Specify a JDBC datasource -->
   <Resource name="jdbc/orderprocessDB" auth="Container"
      type="javax.sql.DataSource" username= "scott" password= "tiger"
      driverClassName= "oracle.jdbc.OracleDriver"
      url= "jdbc:oracle:thin:@//localhost:1521/orcl"
      maxActive="10" maxIdle="4" />
</Context>
```

Setting up database connection pool

The **InitialContext** is configured as a web application is initially deployed, and is made available to web application components (for read-only access).

All configured entries and resources are placed in the **java:comp/env** portion of the JNDI namespace, so a typical access to a resource - in this case, to a JDBC data source - would look something like this:

```
// Obtain our environment naming context
context = new InitialContext();
envContext = (Context) context.lookup("java:comp/env");
// Look up our data source
dataSource = (DataSource) envContext.lookup("jdbc/orderprocessDB");
```

Note: The above statements are to be placed in the DAO class methods.

Note: DataSource doesn't implement *AutoCloseable* interface. Use try..catch block but not try .. with resources

ServletConfig object
ServletContext object

ServletConfig Interface

Every Servlet has its own ***ServletConfig*** object which holds initialization parameters.

Initialization parameters can be made available in web.xml or in annotations.

The methods used to read initialization parameters are:

- **String getInitParameter(String)**
- **Enumeration getInitParameterNames()**

Another useful instance method of ServletConfig object is:

Servlet Context getServletContext()

returns a reference to a ServletContext object, used by the caller to interact with its servlet container

To configure the initialization parameters for Servlet in web.xml:

```
<servlet>
        <servlet-name>Servlet Name</servlet-name>
        <servlet-class>package-name.servlet-classname</servlet-class>
        < init-param>
                <param-name>param1</param-name>
                <param-value>value1</param-value>
        </init-param>
        .....
</servlet>
```

Initialization parameters for a Servlet through annotations

The initial parameters for a Servlet can set using annotations, **@WebInitParam**

The @WebInitParam annotation supports two attributes:

- name - The name of the parameter
- value - The value of the parameter

Using @WebInitParam annotation to specify servlet init parameters

```
@WebServlet(name = "Servletname", urlPatterns = {"/urlpattern"},
 initParams = {@WebInitParam(name="param1", value="value1"),
@WebInitParam(name="param2", value="value2")}
                    )
```

Example

```
@WebServlet(name="InitParamsServlet",urlPatterns="/InitParamsServlet",
initParams={@WebInitParam(name="dbname",value="MySQL"),
@WebInitParam(name="username", value="root"),
@WebInitParam(name="password", value="root123"),
@WebInitParam(name="tablename",value="emp")
})
public class InitParamsServlet extends GenericServlet {
…….
}
```

Accessing initialization parameters in servlet

- The Servlet Container creates the ServletConfig object during Servlet instantiation and stores the initialization parameters and their values in it.
- We can access the initialization parameters in init() method or in service() methods.
- We know that init() method is invoked only once during its lifecycle during the servlet instantiation.

  public void init() throws ServletException

- *To acquire the ServletConfig object within init() or service() methods,*
  - **getServletConfig() method** of Servlet interface returns the object of ServletConfig

ServletContext Interface

public interface **ServletContext**

- 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.
- There is one context per "web application" per Java Virtual Machine.
- The ServletContext object is contained within the ServletConfig object, which the Web server provides the servlet when the servlet is initialized.
- ServletContext object is created by the web container and the context parameters are obtained from deployment descriptor file, **web.xml** file.
- There is no annotation support for context parameters.

How Context parameter is initialized inside web.xml

```
<web-app ...>

<context-param>

     <param-name>driverName</param-name>
     <param-value>sun.jdbc.JdbcOdbcDriver</param-value>

</context-param>

<servlet>
.....
</servlet>

</web-app>
```

The <context-param> is for whole appliaction, so it is put inside the <web-app> tag but outside any <servlet> declaration

Parameter name

Parameter value

We can configure more than one context parameters.

Acquiring *ServletContext* object in Servlet:

ServletContext **servletContext** = this.getServletContext();

OR

ServletContext **servletContext** = getServletConfig().getServletContext();

ServletContext Interface

Web application initialization:

•First, the web container reads the deployment descriptor file and creates a name/value pair for each ***<context-param>*** tag.

•After creating the name/value pair it creates a new instance of **ServletContext**

•It is the responsibility of the Container to give the reference of the ServletContext to the context init parameters.

•The servlets and jsps which are part of the same web application can have the access to the ServletContext.

•The Context init parameters are available to the entire web application.

ServletContext Interface Methods

Some Important method of ServletContext

Methods	Description
Object `getAttribute(String name)`	returns the container attribute with the given name, or NULL if there is no attribute by that name.
String `getInitParameter(String name)`	returns parameter value for the specified parameter name, or NULL if the parameter does not exist
Enumeration `getInitParameterNames()`	returns the names of the context's initialization parameters as an Enumeration of String objects
void `setAttribute(String name,Object obj)`	set an object with the given attribute name in the application scope
void `removeAttribute(String name)`	removes the attribute with the specified name from the application context

Thank You!