J2EE

J2EE is a platform independent, Java centric environment for developing, building and deploying web-based enterprise applications online.

J2EE/Java EE applications aren't self-contained. In order to be executed, they need to be deployed in a container. In other words, the container provides an execution environment on top of the JVM.

J2EE consists of only specifications for the primary services required by any enterprise application. The specifications are provided for Servlet API, JSP API, JPA API, etc.

Why J2EE?

1. Because it supports variety of clients: thin (web clients), thick (application clients) and smart (mobile clients).

2. J2EE is server independent, as it only provides specifications and the implementation is given by the server vendors. Such servers provide readymade implementation for all the primary services like Servlet, JSP, JPA, etc. Hence such an application created using J2EE can be deployed on any compliant server and produce the same output without change any code.

Packaging J2EE Applications

A Java EE application is delivered in a Java Archive (JAR) file, a Web Archive (WAR) file, or an Enterprise Archive (EAR) file. A WAR or EAR file is a standard JAR (.jar) file with a .war or . ear extension. Using JAR, WAR, and EAR files and modules makes it possible to assemble a number of different Java EE applications using some of the same components. No extra coding is needed; it is only a matter of assembling (or packaging) various Java EE modules into Java EE JAR, WAR, or EAR files.

Servlet

Before Servlet, CGI (Common Gateway Interface) was a common web-programming language.

Servlet is a web API which provides many interfaces and classes, “javax.servlet.Servlet” being one of them.

Servlet is basically a small program that runs on a web server. This servlet receives requests from client and responds to them using response object.

Implementation classes of Servlet Interface are GenericServlet -> HttpServlet.

GenericServlet is a protocol-independent servlet, whereas HttpServlet is Http protocol-specific.

HttpServlet

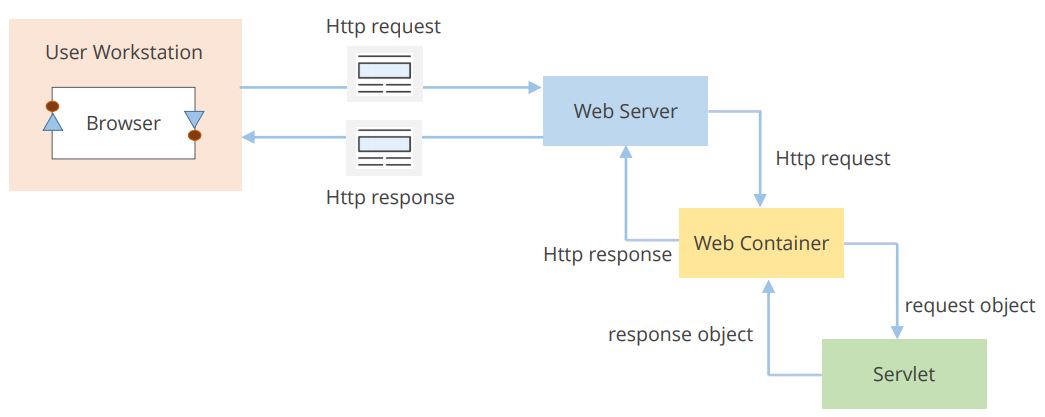
Why HttpServlet class is declared as abstract class BUT with 100 % concrete functionality?

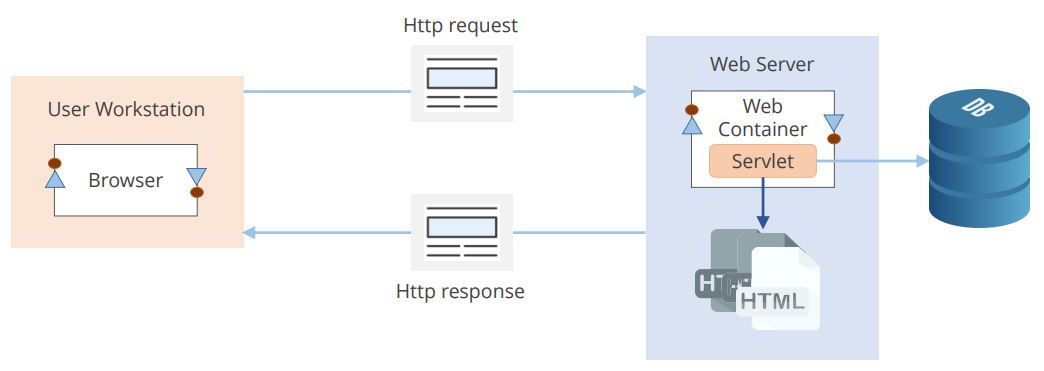
This is because when programmer extends HttpServlet class he provides the implementation of key servicing methods. So making this class as abstract restricts us from instantiating the HttpServlet class directly.

Now if the methods within HttpServlet class were to be made abstract as well, we would have an overhead of implementing every single abstract method, even though we might not need all in our Servlet controller.

For example we would have to implement all do\* methods even though we needed only doGet() method.

If you extend the class without overriding any methods, you will get a useless servlet; i.e. it will give an error response for all requests (HTTP 405 : Method not implemented) . So, if the class was not abstract, then any direct instance of HttpServlet would be useless.

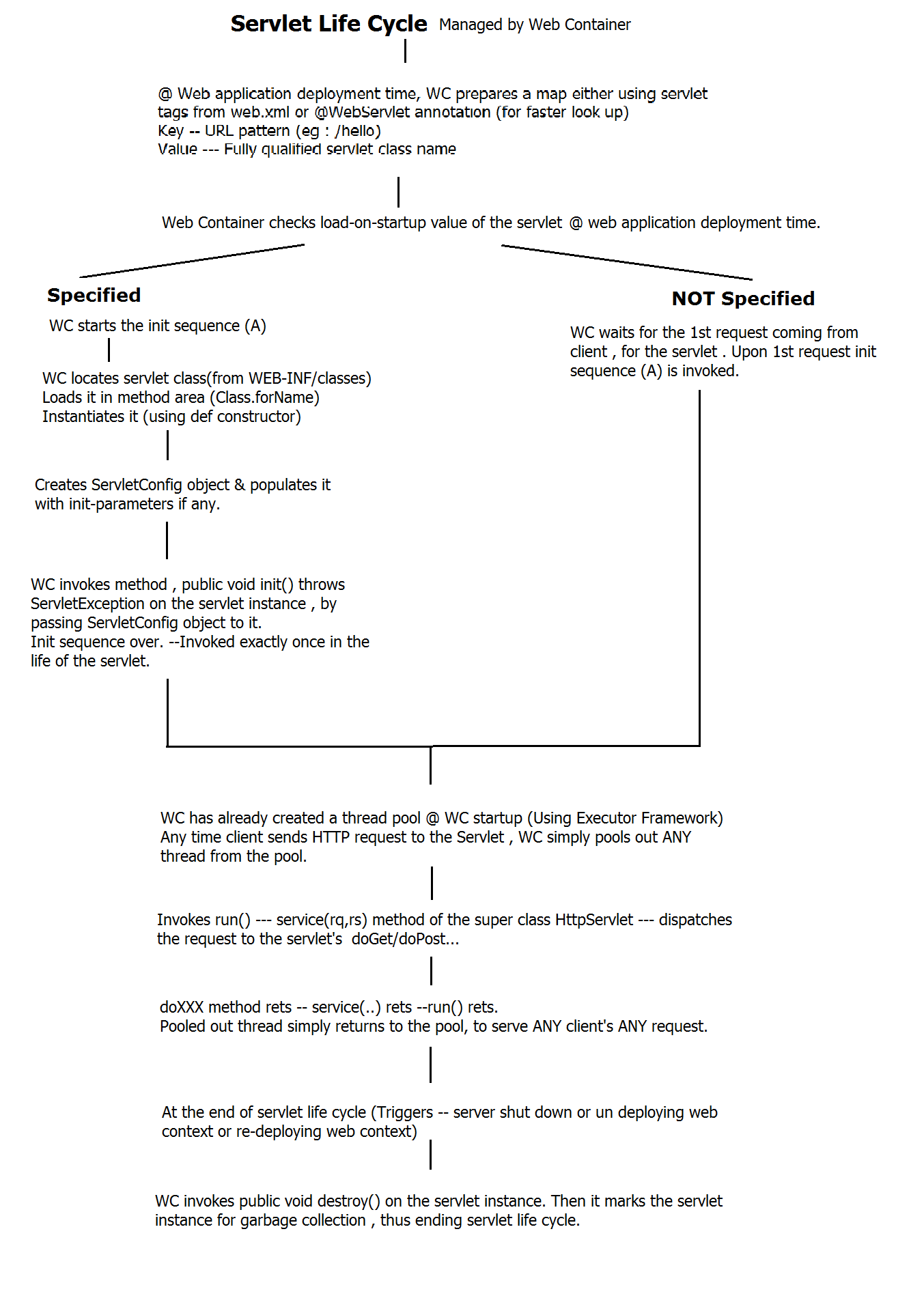




The Apache Tomcat web server has a server side JVM called as the Web Container. Servlet is just one of the many types of Dynamic Web Components that a web container can hold.

Tasks performed by a web container:

* The task of handling the incoming requests and mapping it to appropriate web component (servlet in this case) is done by the web container.
* The creating of http request and response objects is also handled by the web container.
* Each web components (ex. Servlet) have their own lifecycle methods, and are managed/invoked by web container.
* Servlet lifecycle methods: init(ServletConfig config), service(ServletRequest request, ServletResponse response), destroy().
* The web container uses lazy loading strategy, i.e the servlet is invoked only when the client sends a request. To make a servlet load eagerly, use ‘loadOnStartup = 1’ parameter either inside @WebContainer annotation or <load-on-startup>1</load-on-startup> in web.xml file.
* Web container also creates an empty map object at deployment time which contains key value pairs meant for URL mapping purpose. This configuration can be done with annotations or using xml configuration.
* Other miscellaneous tasks like session tracking is also managed by the webserver.



ServletConfig Interface:

An object of ServletConfig Interface is used by the Web Container to pass information to any specific servlet during its initialization. This object simply represents servlet specific configurations.

The instance of ServletConfig interface is created by the web container (one per servlet) after it has created a servlet instance using default constructor (default constructor must be provided), and then invokes the init(ServletConfig config) method of that servlet.

Use case: Providing DB specific parameters (ex. Class.forName, getConnection url, hostname, password) via ServletConfig object, to make our web app db independent (partially).

Now, these servlet specific init parameters or ServletConfig parameters can be added either in web.xml or can be provided via @WebServlet annotations.

However we prefer placing the ServletConfig parameters in web.xml because then we will not have to make changes in our Java code, making it a complete dynamic web application.

<servlet>

<servlet-name>init</servlet-name>

<servlet-class>ex.TestInitParam</servlet-class>

<init-param>

<param-name>name</param-name>

<param-value>value</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>init</servlet-name>

<url-pattern>/test\_init</url-pattern>

</servlet-mapping>

Or

@WebServlet(name = "MyServlet", urlPatterns = {"/MyServlet"}, initParams = {

@WebInitParam(name = "name", value = "Guest")})

The above mentioned process of reading the init parameters from web.xml file and providing it to the appropriate servlet is done by web container itself and is called as Dependency Injection.

More Explanation: Instead of the dependent object (here Servlet’s init method is dependent on ServletCongfig object) managing its dependencies all by itself, Web Container will manage the dependencies for every Servlet. i.e. programmer will not have to create a ServletConfig object consisting of all the init parameters within the servlet.

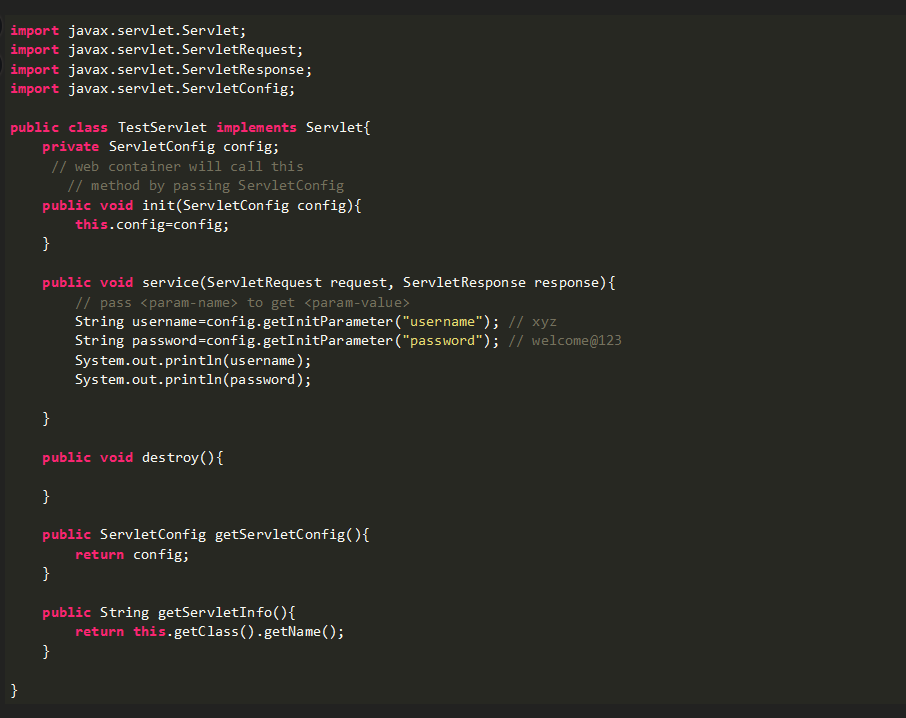
Even in case of no init parameters provided, an empty ServletConfig object will be generated by the Web Container.

Finally we as a programmer will have to access the ServletConfig init parameters. The method used for the same is:

String getInitParameter(String paramName)

Ex: ServletConfig cf = getServletConfig();

String str = cf.getInitParameter(“name”);



Attributes:

An attribute is a server side entry of key and value pair (String name, Object object). An attribute can be added under various scopes. Ex: page (JSP specific), request, session and application. These attributes are created and managed within a servlet using the following API methods:

1. setAttribute(String name, Object object)
2. getAttribute(String name)
3. removeAttribute(String name)
4. Enumeration<String> getAttributeNames()

The above mentioned API methods are common for all scopes. The different scopes are explained in detail below:

ServletRequest: Available for current request only, garbage collected after response is sent.

Ex. request.setAttribute(“key”, val);

HttpSession: Available for current session, shared between all servlets/web pages, but for each unique client.

HttpSession httpSession = request.getSession();

httpSession.setAttribute(String attributeName, Object attributeObject);

ServletContext: Available throughout the web application.

ServletContext:

Similar to ServletConfig an object of ServletContext is created by web container by at the time of deployment of our web application (one per web application). This object can be used to get configuration information from web.xml.

The ServletContext object is contained within the ServletConfig object, which the WC provides the servlet when the servlet is initialized.

<context-param>

<param-name>Password</param-name>

<param-value>SomePassword</param-value>

</context-param>

Accessing this ContextParameters:

ServletContext cx = getServletContext();

String str = cf.getInitParameter(“name”);

One web application can have only one ServletContext object. This ServletContext object is inherently thread un-safe, hence should be accessed in thread safe manner.

Usage: To provide server side login details (eg.Admin Login)

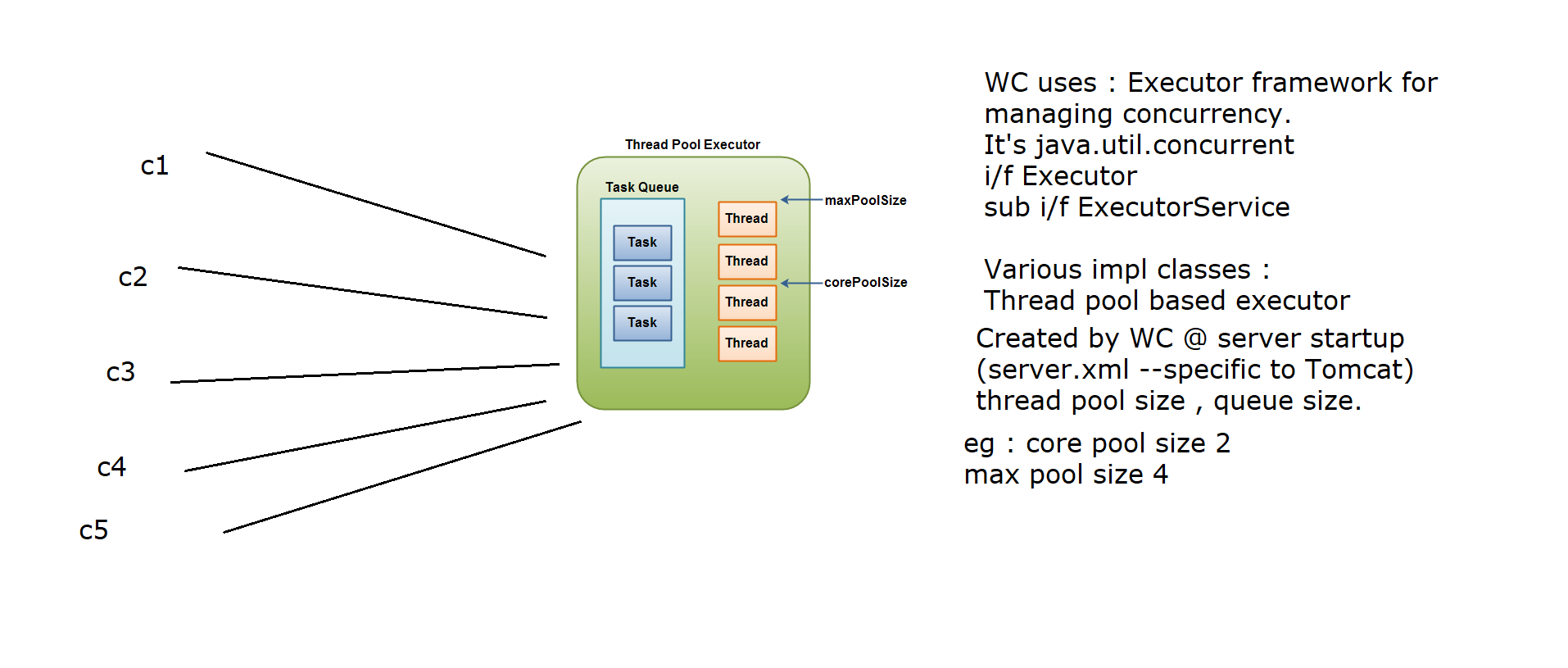
Executor Framework: “java.util.concurrent.Executor”

In order to handle multiple clients sending concurrent requests our web container has to implement multithreading pattern to manage concurrency. However if our web container starts creating a new thread for each incoming request, the bandwidth of our server will exhaust soon.

This is where Executor Framework comes to help.

A framework having a bunch of components that are used for managing worker threads efficiently is referred to as **Executor Framework.** In executor framework, you can create specified number of threads and reuse them to execute more tasks once it completes its current task. All these threads are stored in a thread pool.

The programmer does not have to create or manage threads themselves, that’s the biggest advantage of executor framework.



J2EE Compliant web application folder structure. This folder structure will be uniform across all the servers.

Source Path Deploy Path

src/main/java WEB-INF/classes

src/main/webapp /

WEB-INF (Web Information): Mandatory branch. Private folder, not accessible to client.

1. web.xml: Deployment descriptor, one for each web application.
2. classes: All .java files are compiled and kept here
3. lib: It consists of all external dependencies and .jar files.

webapp: Root folder, public folder.

Routing / Navigating in Servlets: We have two ways to navigate a client across multiple webpages.

1. Client Pull: Navigating the client through a new request-response cycle.

2. Server Pull: Navigating the client by stretching the same request on the server side before sending the response.

1. The Client Pull technique consists of two scenarios, one where client is directly involved (ex. click event) and the other where client is indirectly involved (ex. client browser generating new uri on successful login to redirect the user).

In this scenario client sends a new request to the server in order to navigate to next page.

void sendRedirect([String](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html?is-external=true) location)throws [IOException](http://docs.oracle.com/javase/7/docs/api/java/io/IOException.html?is-external=true)

Syntax: response.sendRedirect(“servleturi”);

Sends a temporary redirect response to the client using the specified redirect location URL and clears the buffer.

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.

2. The Server Pull technique stretches the existing request made by a client across multiple servlets. This process is called as resource chaining/request dispatching.

RequestDispatcher Interface: Defines an object that receives requests from the client and forwards/includes them to any resource (such as a servlet, HTML file, or JSP file) on the server.

Why use Server Pull? For separation of concerns. Instead of letting one Servlet handle all the logic, we divide it across multiple servlets and chain it using RequestDispatcher.

Implementation:

Step 1: Create a RequestDispatcher object.

RequestDispatcher rd = request.getRequestDispatcher(“S2”);

Step 2: Forward the current request made by client. doPost requests are forwarded to doPost of next servlet and so on. Before forwarding the request WC clears the PrintWriter buffer.

rd.forward(request, response);

Step 3: Once the execution of S2 servlet is done, the control returns back to S1 servlet, however S1 servlet cannot contribute anything to the response object. i.e. while using forward method, the last servlet in resource chaining sends the dynamic response to client.

Step 4: Using include method instead of forward at step 2. It includes the response contents of S2 servlet into S1 at runtime. Finally the information from both S1 and S2 servlet are appended to response buffer and then sent back to client.

In this scenario, once the execution of S2 servlet has completed and the control returns back to S1 servlet, S1 can continue to add information to response buffer.

Limitations - The included servlet/JSP cannot change the response status code or set headers; any attempt to make a change is ignored.

J2EE specific Session Tracking techniques:

What is a session? Session is a conversational state between the server and client.

1. Plain cookie based scenario.
2. HttpSession interface
3. HttpSession + Url Rewriting.

1. Plain Cookie based Session Tracking:

“javax.servlet.http” package.

Cookie is small amount of text data. Cookies are created by server and sent to web browser, saved by the browser, and later sent back to the server. The values stored in a cookie can be used to uniquely identify a client, so cookies are commonly used for session management.

Step 1: Create a Cookie object.

Cookie c1 = new Cookie(“userid”, user.userid);

Step 2: Add this cookie to response header.

response.addCookie(c1);

Step 3: As the browser receives cookies attached to response header from the server, it checks for ‘Cookies Enabled’ settings. If enabled, saves it at client side, if not, the cookie data is lost.

Step 4: Setup expiry for cookie.

setMaxAge(-1);

* Non-Persistent: -1 is default expiry value. Indicating to save cookie in cache memory of the browser, meaning this cookie will be deleted as soon as the browser window is closed.
* 0 is to delete the cookie
* Persistence: >0 Indicates expiry age in seconds. This will make browser store the cookie on user’s hard disk rather than in cache memory.

Step 5: When a request is sent from clients browser to the server, all the cookies available on client side are attached to request header before making the request.

Step 6: Fetching the cookies.

Cookie cookies[] = request.getCookies(); //Now iterate over this array for required cookie.

Step 7: Deleting cookies. Simply set the age of cookie to zero.

c1.setMaxAge(0);

Other useful methods of Cookie class:

1. getMaxAge()
2. getName()
3. getValue()
4. setName(String name)
5. setValue(String value)

Disadvantages of the above technique:

1. Programmer has to write extensive code to manage cookies.
2. Cookies can manage only text based data i.e. every data has to be converted to string.
3. More cookies increases network traffic.
4. If cookies are deleted or disabled, this session tracking fails.

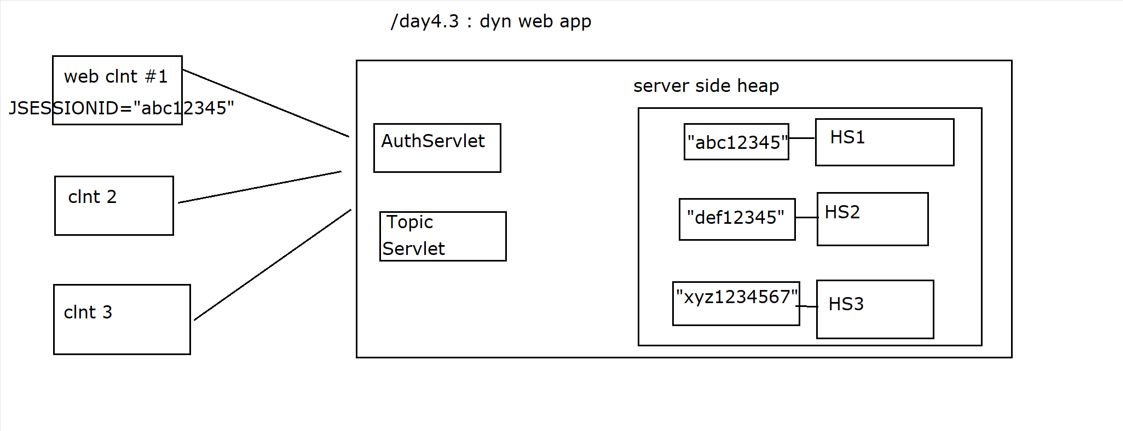
2. HttpSession Interface based Session Tracking:

“javax.servlet.http” : HttpSession Interface

In this technique, the entire state of client is not stored on client side, but is saved on server side data structure HttpSession object. However the key to this HttpSession object is sent to client in form of a cookie. This cookie management is done by server itself.

To identify which session object belongs to which client, Web container creates internal cookie called JSessionId (Java Session Id). This cookie is sent to client browser, which will be further used to uniquely identify the client.

Every HttpSession object internally creates a map of session scope attributes.



Step 1: Getting the session object from server.

HttpSession httpSession = request.getSession();

Step 2: Saving a session object on server.

httpSession.setAttribute(String attributeName, Object attributeObject);

Step 3: Retrieving session attribute from server.

Customer c = (Customer) httpSession.getAttribute(“attributeName”); //Object returning method.

Step 4: Changing session timeout value. Default time for tomcat is 30min.

httpSession.setMaxInactiveInterval(seconds);

Step 5: Removing a session attribute from server.

httpSession.removeAttribute(“sttributeName”);

Step 6: Invalidating/Destroying a session object present on server. Ready for garbage collection on server’s JVM.

httpSession.invalidate();

Other methods of HttpSession API:

1. getId() = Returns JSessionId
2. isNew()

Disadvantages:

1. Requires cookies to be enabled on client’s browser to save the cookie containing JSessionId.

3. HttpSession + URL Rewriting:

Url rewriting is an alternative preferred when client’s browser doesn’t support cookies.

For tracking the client’s session, the only information WC needs from the client’s browser is the JSessionID value. If client’s browser is not sending it using cookies Servlet/JSP program can embed the JSessionID info in each outgoing URL. (Response: location / href /form action)

The WC first checks for JSessionID within incoming cookies, if not found then it will check in URL.

How to extract JSessionID from URL?

API:

For URLs generated by clicking link/buttons (client pull I) use:

HttpServletResponse method: public String encodeURL(String origURL)

Returns: origURL; JSESSIONID=12345

For URLs generated by sendRedirect (client pull II) use:

HttpServletResponse method: public String encodeRedirectURL(String redirectURL)

Returns: redirectURL; JSESSIONID=12345

Note: The exact syntax of appending JSessionID to a URL differs from server to server and cannot be exactly determined.

Web Servlet Listener:

Also known as web application listener.

During the lifetime of a typical web application, a number of events take place. ex: requests are created or destroyed.

Sessions are created & destroyed.

Contexts(web apps) are created & destroyed.

request or session or context attributes are added, removed, or modified etc.

The Servlet API provides a number of listener interfaces that one can implement in order to react to these events.

1. ServletRequestListener
2. ServletRequestAttributeListener
3. HttpSessionListener
4. ServletContextListener
5. ServletContextAttributeListener
6. AsyncListener

Steps

1. Create a class, implementing from Listener i/f.

2. Register it with WC using

2.1 @WebListener annotation (class level)

OR

2.2 XML tags in web.xml

<listener>

<listener-class>F.Q cls name of listener</listener-class>

</listener>