Java Servlet and JSP

fwang2@ornl.gov

My study notes based on wonderful O'reilly Head First Servlet book.

Table of Contents:

---

1. Web Basics

1.1. HTTP GET Request

GET request append form data to the end of the URL. An Example:

```
GET /select/test.jsp?color=dark&shape=circle HTTP/1.1
Host: ...
Accept-Encoding: gzip, deflate
...
Connection: keep-alive
```

We need to know:

- HTTP method: `GET`
- Path to resource on server: `/select/test.jsp`
- Parameter appended: `color=dark&shape=circle`
- Protocol: `HTTP/1.1`

1.2. HTTP POST Request: Sending and Using Two Parameters

- HTML form

```
<select name="color" size="1">
    <option> Light </option>
    ...
</select>

<select name="shape" size="1">
    <option> circle </option>
    ...
</select>
```

- POST request includes form data in the body of the request.

```
POST /select/test.jsp HTTP/1.1
...
Connection: keep-alive
    color=dark&shape=circle    // this is known as message body or payload
```

- Servlet class

```
public void doPost( ... ) {
    String colorParam = request.getParameter("color");
    String bodyParam = request.getParameter("shape");
    ...
}
```

We need to know:

- This time, the parameters are put in the message body, not limited as in HTTP Get Request.

1.3. HTTP Response

```
HTTP/1.1 200 OK
Content-Type: text/html
...
<html> ....
</html>
```

The content-type response header is known as MIME type - it tells to browser what kind of data it is about to get so browser knows how to render it.

Common response/status code:

- 200: OK
- 301: moved and redirect
- 401: unauthorized
- 403: forbidden
- 404: not found
- 500: internal error

1.4. An Servlet Example

```
// test.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class TestServ extends HttpServlet {
    public void doGet(HttpServlet request, HttpServletResponse response)
        throws IOException {
        PrintWriter out = response.getWriter();
        java.util.Date today = new java.util.Date();
        out.println("<html><body>" + today + "</body></html>");
    }
}

// compile
% javac -classpath $CATALINA_HOME/common/lib/servlet-api.jar \
    -d classes src/TestServ.java

// deploy
```

As you can see, a servlet has `doGet` and `doPost` to handle GET request and POST request, it doesn't have a `main()` method, that is where the container comes in.

2. Servlet and Container

2.1. What is container?

When a request for a *servlet* comes in, the server (such as Apache) hands the request not to servlet itself, but to the **Container** in which the servlet is deployed. It is the container that gives the servlet the HTTP request and response, and it is the container that calls servlet's method such as `doGet()` and `doPost()`.

2.2. Why Container?

- Communication support (easier for servlet to talk to web server)
- Life cycle management (life and death of servlet)
- Multithreading support
- Declarative security
- JSP support (translating JSP code into Java code)

2.3. How does container work?

- As container "sees" the request for a servlet, it creates two objects: `HttpServletResponse` and `HttpServletRequest`
- The container finds the correct servlet based on the URL in the reqeust, create or allocates a thread for that request, and passes the request and response object to the servlet thread.
- The container calls the servlet's `service()` method. Depending on the type of request, it calls either the `doGet()` or `doPost()` method.
- The `doGet()` method generates the dynamic page and stuffs the page into the response object.
- After servlet thread completes, the container converts the repose object into HTTP response, sends it back to the client, then deletes the request and response object.

2.4. Servlet name and Deployment Descriptor (DD) mapping

One key function of container is to locate corresponding servlet. A servlet is known to have three names:

- *file path name*: such as `org.esgf.web.SearchController`.
- *deployment name*: internal name, can be any.
- *public URL name*: this is the name clients knows about. For example, `/search/facets`.

The mapping can be accomplished by two DD elements:

- `<servlet>`: maps internal name to fully-qualified class name

```
<servlet>
    <servlet-name>internal name X</servlet-name>
    <servlet-class>foo.bar.Servlet1</servlet-class>
</servlet>
```

- `<servlet-mapping>`: maps internal name to public URL name

```
<servlet-mapping>
    <servlet-name>Internal name X</servlet-name>
    <url-pattern>/public_whatever</url-pattern>
</servlet-mapping>
```

Note `/public_whatever` is what client sees and uses to get to the servlet.

2.5. Tomcat is a web container, not a J2EE application server.

- J2EE is a super spec: it includes not only servlet 2.4/2.5 spec, JSP 2.0/2.1 spec, but also Enterprise JavaBean 2.1 spec for EJB container.
- Tomcat only supports servlet and JSP, that is for *web component*, not EJB, which is for *business component*. Thus Tomcat is known as web container, not J2EE applicaton server.

- The real application server such as JBoss, BEA's web logic, IBM's WebSphere.

3. Servlet Lifecycle and API

3.1. Servlet's life

- There is only one main state of servlet: **initialized**. If a servlet is not initialized, then it is either being initialized (running its constructor, or `init()` method), being destroyed (running its `destroy()` method), or doesn't exisit.

3.2. Three Life cycle moments

- `init()`: The container calls init() on servlet instance after servlet instance is created, but before servlet can service any client request.
- `service()`: When first client request comes in, the Container starts a new thread or allocate a thread from the pool, and causes the servlet's `service()` method to be invoked.
- `doGet()` or `doPost()`: The `service()` method invokes either based on HTTP methods.

3.3. `ServletConfig` object

- One `ServletConfig` object per servlet
- Use it to pass deploy-time information to servlet (database, bean lookup name etc.)
- Use it to access the `ServletContext`.
- Parameters are configured in the Deployment Descriptor.

3.4. `ServletContext` object

- One `ServletContext` per web app.
- Use it access web app parameters, as configured by Deployment Descriptor

3.5. Idempotent (or Not)

- GET is not suppose to change anything on the server, so GET is, idempotent. In HTTP/servlet context, it means the same request can be made twice with no negative consequences.
- POST is NOT idempotent.
- POST is not default. For example, in the form tag, if you don't specify, the submit request is GET request.

3.6. Example: servlet code to download a JAR

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

    // let browser know this is a jar, not html
```

```
    response.getContentType("application/jar");

    ServeletContext ctx = getServletContext();
    InputStream is = ctx.getResourceAsStream("/myCode.jar");
    int read = 0;
    byte[] bytes = new byte[1024];

    OutputStream os = response.getOutputStream();
    while ((read = is.read(bytes) != -1) {
        os.write(bytes, 0, read);
    }
    os.flush(); os.close();
}
```

The `getResourceAsStream()` requires forward slash, which is the root of web app. That is

```
webapps/ --> JarDownload/ --> myCode.jar
                          --> WEB-INF/
```

3.7. Request Redirect

- You can handle request in a servlet, you can also *dispatch* request to some other components in your web app, typicall a JSP. You can ALSO redirect as:

  ```
  if (I_can_do_it) {
      // handle the request
  } else {
      response.sendRedirect("http://www.google.com");
  }
  ```

- Using Relative URL in `sendRedirect()`

  suppose your request comes in with URL as:

  ```
  http://www.hotdot.com/myApp/cool/bar
  ```

  If you redirect it with

  ```
  sendRedirect("foo/bar")
  ```

  Then container will build the full URL (needed for the Location header it puts in the HTTP response):

  ```
  http://www.hotdot.com/myApp/foo/bar
  ```

  However, you start with a forward slash, such as `sendRedirect('/foo/bar')`, then the full URL will be:

  ```
  http://www.hotdot.com/foo/bar
  ```

- **Important**: You can't do `sendRedirect()` after you write to response. If you do that, you will see `IllegalStateException`.

3.8. Redirect vs. Request Dispatch

The biggest difference is that "redirect" makes the client do the work, while "request dispatch" make something else on the server do the work.

- *Request dispatch*: user puts in a URL, request comes to server, servlet decides the request should go to a JSP, the servlet calls:

  ```
  RequestDispatcher view = request.getRequestDispatcher("result.jsp");
  view.forward(request, response);
  ```

  From this point on, JSP takes over the response.

  The brwoser gets the response in the usual way, the render it for user. The browser location doesn't change, user doesn't know JSP generates the response.

- *Request redirect*: it simply tells the browser to go to a different URL.

3.9. Summary

- The Container initializes a servlet by loading the class, invoking the servlet's no-arg constructor, and calling the servlet's `init()` method.
- The `init()` method, which developer can override, is called only once in a servlet's life and always before the servlet can service any client requests.
- The `init()` method give the servlet access to the `ServletConfig` and `ServletContext` objects, which the servlet needs to get information about the servlet configuration and web app, respectively.
- The Container ends a servlet's life by calling its `destroy()` method.
- Most of a servlet's life is spent running a service method for a client request.
- Every request to a servlet runs in a separate thread! There is only one instance of any particular servlet class.
- Your servlet will almost always extend `javax.servlet.http.HttpServlet`, from which it inherits an implementation of the service method that takes an `HttpServletRequest` and an `HttpServletResponse`.
- `HttpServlet` extends `javax.servlet.GenericServlet` - an abstract class that implements most of the basic servlet methods.
- `GenericServlet` implements the `Servlet` interface.
- Your servlet must override at least one service method, `doGet()`, `doPost()`, etc.

4. Web App: Attributes and Listeners

4.1. Servlet Init Parameters

```
# web.xml
<servlet>
    <servlet-name> ... </servlet-name>
    <servlet-class> ... </servlet-class>
    <init-param>
        <param-name>adminEmail</param-name>
        <param-value> abc@google.com</param-value>
    </init-param>
</servlet>
```

In the servlet code:

```
out.println(getServletConfig().getInitParameter("adminEmail"));
```

A few caveats:

- You can't use servlet init parameter unitl it is initialized - that means you can't use it in your servlet constructor.
- Servlet init parameters are read only one - when Container initialize the servlet.
- You can have multiple `init-param` for the servlet, and you can use `getInitParameterNames()` to get a list of names and further use `getInitParameter()` to get each value.

4.2. Context Init Parameters

If you want all other parts of your web app have access to that email address, you can put it as part of context parameter.

```
# web.xml
<servlet>
        ...
</servlet>

<context-param>
         <param-name>adminEmail</param-name>
         <param-value> abc@google.com</param-value>
</context-param>
```

Note that we are not nesting `context-param` inside `servlet` anymore, it is for the whole web app. In the servlet code:

```
out.println(getServletContext().getInitParameter("adminEmail"));
```

4.3. Context Listener and why?

Context parameter can't be anything except Strings. What if you want the whole web app have access to a shared database connection? You can't put that code in a servlet, since then you must have ways to guarantee the servlet must run before anybody else.

What in need is a *listener* for a context initialization event, so **it can run some code before the rest of the app can service a request**.

We need three classes and one definition for doing this:

- Write the listener class: its gets the context init parameter, create some domain object, Dog; and set Dog as **context attribute**.

```
public class MyServletContextListner implements ServletContextListner
{

    public void contextInitialized(ServletContextEvent event) {
        ServletContext sc = event.getServletContext();
        String dogBreed = sc.getInitParameter("breed");
        Dog d = new Dog(dogBreed);
        // now other parts of the app will be able to get the value
        // of the attribute
        sc.setAttribute("dog", d);
    }

    public void contextDestroyed(ServletContentEvent event) {
        // nothing
    }
}
```

- The attribute class: say we will define Dog.java - its job is to be the attribute value that `ServletContextListner` instantiates and sets in the `ServletContext`, for other servlet to retrieve.

```
public class Dog {
    private String breed;
    public Dog(String breed) {
        this.breed = breed;
    }
    public String getBreed() {
        return breed;
    }
}
```

- Deployment descriptor

```
<servlet>
    <servlet-name> ListnerTester </servlet-name>
    <servlet-class> example.ListenerTester </servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name> ListnerTester </servlet-name>
    <url-pattern> /ListenerTest.do </url-pattern>
</servlet-mapping>
```

```
<context-param>
    <param-name> breed </param-name>
    <param-value> Great Dane </param-value>
</context-param>

<listener>
    <listener-class>
        example.MyServletContextListener
    </listener-class>
</listener>
```

- The servlet itself, just for testing purpose.

```
public void doGet() {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("test context attribute set by listener<br>");
    Dog dog = (Dog) getServletContext().getAttribute("dog");
    out.println("Dog's breed is: " + dog.getBreed());
}
```

4.4. Other type of Listeners

Besides context event, you can listen for events related to context attributes, servlet requests and attributes, and HTTP session and session attributes. (P182)

- You want to know if an attribute in a web app context has been added, removed, or replaced. `ServletContextAttributeListener`.
- You want to know how many active sessions. `HttpSessionListener`.
- You want to know each time a request comes in, so you can log it. `ServletRequestListener`.
- You want to know when a request attribute has been added, remove, or replaced. `ServletRequestAttributeListener`.
- You have a attribute class (a class for an object that will stored as an attribute) and you want to objects of this type to be notified when they are bound or removed from a sesson. `HttpSessionBindingListener`.
- You want to know when a sessiion attribute has been added, removed or replaced. `HttpSessionAttributeListener`.
- You want to know if a context has been created or destroyed. `ServletContextListener`.
- You have a attribute class (a class for an object that will stored as an attribute) and you want to objects of this type to be notified when the session to which they are bound is migrating to and from antoher VM. `HttpSessionActivationListener`.

A plain old `HttpSessionAttributeLisener` is just a class that wants to know when *any* type of attribute has been added, removed, or replaced in a session. But the `HttpSessionBindingListener` exists so taht the attribute itself can

find out when it has been added or removed from a session. Example:

```
public class Dog implements HttpSessionBindingListener {
    ...
    public void ValueBound(HttpSessionBindingEvent event) {
    ...
    }

    public void ValueUnbound(HttpSessionBindingEvent event) {
    ...
    }
}
```

4.5. Attribute, Scope and API.

- What is an attribute? An attribute is an object *set* or *bound* into one of three other servlet API objects: `ServletContext`, `HttpServletRequest`, or `HttpSession`. The most important thing to know about attribute is who can see it and how long it lives.

  Note that Attributes are not parameters: the name/value pair for attribute, where value is object, instead of string in the case of parameter.

- Three scopes: Context, Request and Session

- Fortunately, all three attribute scopes, that are handled by `ServletContext`, `HttpServletRequest`, and `HttpSession` has the exact same API:

  ```
  Object getAttribute(String name);
  void setAttribute(String name, Object value);
  void removeAttribute(String name);
  Enumeration getAttributeNames();
  ```

A few caveats:

- Context and Session scope attributes are not thread-safe: somebody with access it can change it …

- Only local variable and request attributes are thread-safe.

- Request attribute make sense when you want other component such as JSP to take over all or part of the request. Typically, the controller communicate with the model, and get back dagta that the view needs in order to build the response. There is no reason to put the data in a context or session attribute, since it applies *only* to the request, so we put it in the request scope:

  ```
  request.setAttribute("styles", result);
  ```

4.6. More on `RequestDispatcher`

There are two ways you can get this object:

- Get it from `ServletRequest`:

```
RequestDispatcher view = request.getRequestDispatcher("result.jsp");
```

Since we use relative path here, the Container looks for “result.jsp” in the same logical location the request is “in”.

- Get it from `ServletContext`:

```
RequestDispatcher view =
getServletContext().getRequestDispatcher("/results.jsp");
```

Same as above, except you *must* use forward slash for absolute path.

5. Session Management

A session is keep client-specific state across multiple requests.

5.1. How does Session work?

- User X select beer “dark” and submit
- The Container sends the request to a new thread of the BeerApp servlet, this servlet thread finds the ssession associated with user X, and store his choice in the session as attribute.
- The servlet runs business logic, and return a response, in this case, another question: “how expensive?”
- X consider the new question, and select “expensive” and submit his answer.
- The container sends the request to a new thread of BeerApp servlet, and this thread finds the session associated with user X, and store his choice “expensive” in the session as attribute.
- Meanwhile, another user Y might go through same process, but this time, the BeerApp thread starts a new Session for user Y. The key points are:

  - different clients

  - same servlet

  - different request

  - different threads

  - different session

5.2. Session ID

HTTP is stateless, so as far as the Container’s concerned, **each request is from a new client**. How will container recognize client X from client Y?

The idea is simple: on the client’s first request, the Container generates a unique session ID and gives it back to the client with the response. **the client sends back the session ID with each subsequent request**. The container sees the ID,

finds the matching session, and assocaites the session with the request.

The session ID is exchanged through *cookies*:

- Server response:

```
HTTP/1.1 200 OK
Set-Cookie: JSESSIONID=0ABCDEF2324
...
```

- Client Next Request:

```
POST /select/taste HTTP/1.1
Cookie: JSESSIONID=0ABCDEF2324
...
```

- Enable cookies: All the cookie setting work behind the scenes by container, all you need to is:

```
HttpSession session = request.getSession();
```

This will *create a new session* if no existing session found.

- If you want to know if the sssion already existed or was just created, you can:

```
HttpSession session = request.getSession();
if (session.isNew()) {
    ...
}
```

- If you want only pre-existing session:

```
// passing "false" ask the method return pre-existing session or null
HttpSession session = request.getSession(false);

if (session == null) {
    session = request.getSession();
    ...
}
```

5.3. If client doesn't take cookies: URL Rewriting

- URL rewriting always work. It takes the session ID that in the cookie and sticks it right onto the end the every URL that comes to this app.

```
URL + ;jsessionid=1234567
```

- The session ID comes back as "extra" info to the end of request URL. It kicks in only if cookies fails and ONLY if

you tell the response to encode the URL.

- To encode URL, call `response.encodeURL(String)`:

```
out.println("<a href='" + response.encodeURL("/Beertest.do") +
    "'>TestBeer</a>");
```

- There is no way to get automatic URL rewriting with your static pages, so if you depend on sessions, you must use dynamically generated pages.

5.4. Timeout/Invalidate Sessions

Three ways a session can die:

- Configuring session timeout in DD, the unit here is minutes.

```
<session-config>
    <session-timeout> 15 </session-timeout>
</session-config>
```

- Setting timeout for a specific session

```
session.setMaxInactiveInterval(20*60); // seconds
```

- Invalidate: ends the session, this includes unbinding all session attributes currently stored in this session.

```
session.invalidate();
```

5.5. Cookie API

Cookie can be used for stuff other than session, the best thing about it is, user doesn't have to be involved. And you can tell a cookie to stay alive even after the browser shuts down.

- Create new cookie

```
Cookie cookie = new Cookie("username", name);
```

- Setting how long a cookie lives on client

```
cookie.setMaxAge(30*60); // in seconds
```

- Sending the cookie to client

```
response.addCookie(cookie);
```

- Getting cookie(s) from client

```
Cookie[] cookies = request.getCookies();
for (int i = 0; i< cookies.length; i++) {
    if (cookie.getName().equals("username")) {
        String userName = cookie.getValue();
        ...
     }
}
```

5.6. Final note on session

There is whole bunch of events related to session such as sesson created, destory, session attributes added, removed, replaced, session is passivated in one VM, activated on another etc. where you can attach listeners.

6. JSP Is Just a Servlet

6.1. JSP Basic

- Use page directive to import packages. A directive is a way for you to give special instructions to the Container at page translation time. Directives come in three flavors: **page**, **include**, and **taglib**.

```
<%@ page import="foo.*, java.util.*" %>

<%@ taglib tagdir="/WEB-INF/tags/cool" prefix="cool" %>

<%@include file="header.html" %>
```

- JSP expression:

```
<%@ page import="foo.*" %>
The page count is:
<%= Counter.getCount() %>
```

- JSP scriptlet:

```
<%@ page import="foo.*" %>
<%
    out.println(Counter.getCount());
 %>
```

- JSP declarition: all scriptlet and experssion code lands in a `service()` method. That means variable declared in a scriptlet are always LOCAL variables. For instance variables or method definition, you need:

```
<%! init count = 0 %>
```

6.2. JSP Implicit objects

The container makes the following implicit object available to JSP when they do the servlet translation:

```
API                     Implicit object
---------------------------------------
JspWriter               out
HttpServletRequest      request
HttpServletResponse     response
HttpSession             session
ServletContext          application
ServletConfig           config
JspException            exception      // only available to error page
PageContext             pageContext
Object                  page
```

"page" is a new fourth scope, "page-level", and page-scoped attributes are stored in `pageContext`. It encapulates other implicit objects.

6.3. API for generated servlet

- `jspinit()`: the method is called from `init()` method.
- `jspdestroy()`: the method is called from servlet's `destroy()` method.
- `_jspService()`: this method is called from the servlet's `service()` method, which means it runs in separate thread for each request. You can't override this method!

6.4. Lifecycle of JSP

- Your jsp is deployed, the container doesn't do anything else until the first time it is requested.
- Upon request, container tries to *translate* the .jsp into .java source; then container tries to *compile* .java into .class.
- If all successful, the container *loads* the newly generated servlet class.
- The container instantiates the servlet and causes the servlet's `jspInit()` method to run. It is now a full-fledge servlet, ready to accept client request.
- The container creates a new thread to handle this client's request, and the servlet's `_jspService()` method runs.
- Eventually, the servlet sends a response back to the client (or forward to request to another web app component).

6.5. From Servlet to JSP

```
// Servlet Controller

public doPost(...)
{
    String name = request.getParameter("userName");
    request.setAttribute("name", name);
```

```
    ...
    RequestDispatcher view = request.getRequestDispacher("/result.jsp");
    view.forward(request, response);
}

// JSP (view)

<html><body>
<%= request.getAttribute("name");
```

6.6. Access attribute in a JSP

- Application

```
// servlet
getServletContext().setAttribute("foo", barObj);

// JSP
application.setAttribute("foo", barObj);
```

- Request

```
// servlet
request.setAttribute("foo", barObj);

// JSP
request.setAttribute("foo", barObj);
```

- Session

```
// servlet
reqeust.getSession().setAttribute("foo", barObj);

// JSP
session.setAttribute("foo", barObj);
```

- Page

```
// servlet
don't apply

// JSP
pageContext.setAttribute("foo", barObj);
```

You can use pageContext to get and set attributes in any scope:

```
<%= pageContext.getAttribute("foo", PageContext.SESSION_SCOPE) %>
```

or you can use it to find attribute in any scope if you don't where it is:

```
<%= pageContext.findAttribute("foo") %>
```

7. EL: Expression Language

When JSP wants to access an attribute, assuming the value is an object, here is what you do without EL:

- You can:

```
<% foo.Person p = (foo.Person) request.getAttribute("person"); %>
Person is: <%= p.getName() %>
```

- or

```
<%= ( (foo.Person) request.getAttribute("person")).getName() %>
```

We can get rid of scripting by using bean-related standard actions.

7.1. Bean actions

To treat it like an bean and use standard action

```
<jsp:useBean  id="person" class="foo.Person" scope="request" />

Person created by servlet:
<jsp:getProperty name="person" property="name" />
```

Here:

- `jsp:useBean` and `jsp:getProperty` identify standard actions
- `id="person"` declares identifier for the bean object. This matches to the name used when servlet code said:

```
request.setAttribute("person", p);
```

- `class="foo.Person` declares the fully qualified class type for the bean object.
- `scope="request"` identifies the attribute scope for the bean object.
- `name="person"` identifies the actual bean object, this will match the `id` value from `jsp:useBean` tag.
- `property=name` identifies the property name (things with getter and setting in the bean class).

7.2. Talk directly to JSP without Servlet

You can send request parameters straight into a a bean, without scripting: The html code:

```
<form action="TestBean.jsp">
    name: <input type="text" name="userName">
    ID#: <input type="text" name="userID">
    <input type="submit">
</form>
```

Inside `TestBean.jsp`:

```
<jsp:useBean id="person" type="foo.Person" class="foo.Employee">
    <jsp:setProperty name="person" property="name" param="userName">
```

Note that **param** attribute: the param value comes from the name attribute of the form input field.

It can be simplified further, though I don't know if that is a good idea, by following some conventions:

```
// abstract class foo.Person
String getName();
void setName();

// foo.Employee extends foo.Person
int getEmpID()
void setEmpID(int)
```

Now the HTML needs to be (basically BOTH parameters matches the property name of the bean):

```
<form ...>
    name: <input type="text" name="name">
    ID#$: <input type="text" name="empID">
    ...
</form>
```

Then we get to do this:

```
<jsp:useBean ... >
    <jsp:setProperty name="person" property="*" />
```

Note that only String and primitives are converted automatically.

7.3. JSP EL and Implicit Objects

With EL, the access to JavaBean is even easier:

This

```
<%= ((foo.Person) reqeust.getAttribute("person")).getDog().getName() %>
```

can be replaced by:

```
${person.dog.name}
```

EL's first field can be either implicit object of the following

```
pageScope           // map of scope attributes
requestScope
sessionScope
applicationScope

param               // map of request parameter
paramValues
header              // map of request header
headerValues
cookie
initParam           // map of context init parameters

pageContext  // of all implicit objects, only pageContext is NOT a map.
             // it is an actual reference to pageContext object.
```

OR it can be an attribute in page scope, request scope, session scope, application scope.

7.4. dot `.` operator and `[]` operator

You can use dot operator to access bean properties and map values. However, what if it (e.g. Person) is an array or list? Here are the rules:

- If the expression has a variable folloed by a bracket `[]`, the left hand variable can be a map, a bean, a List or an array.
- If the thing inside the brackets is a String literal (i.e., in quotes), it can be a Map key, or a bean property, or an index into a List or array.
- String index will be *coerced* into an int for arrays and Lists
- If it is not a a string literal, it is **evaluated**.

For example, in servlet:

```
String[] favoriateMusic = ["Dan", "Metal", "Foo bar"];
request.setAttribute("musicList", favoriteMusic);
reqeust.setAttribute("secret", 2);
```

The in JSP

```
First song is: ${musicList[0]}          // Dan
Second song is: ${musicList["1"]        // Meta  (coerced)
Third song is ${musicList[secret]}      // evaluted as ${musicList[2]}
```

7.5. More EL examples on request parameter, and cookies

- Handle Request parameters:

  Suppose you have HTML form as:

```
<form ...>
    Name: <input type="text" name="name">
    ID#:  <input type="text" name="empID">

    First food: <input type="text" name="food">
    Second food: <input type="text" name="food">
</form>
```

  In JSP, here is what you can do:

```
Request param name is: ${param.name} <br>
Request param empID is: ${param.empID} <br>
Request param food is: ${param.food} <br>
```

  Now, even though there are multiple values for the `food` parameter, the way above line is written, you only get the first value. Here is how you can get to the rest:

```
First food param: ${paramValues.food[0]} <br>
Second food param: ${paramValues.food[1]} <br>
```

  Even name doesn't have mutliple values, I can still access by:

```
${paramValues.name[0]}
```

- To Print out the value of the "userName" cookie:

```
${cookie.userName.value}
```

- To Print out the value of context init parameter:

```
<context-param>
    <param-name>mainEmail</param-name>
    <param-value>fwang2@gmail.com</param-value>
</context-param>
```

  With EL: `email is ${initParam.mainEmail}.`

7.6. The `requestScope` is NOT request object

The implicit `requestScope` is just a Map of the request scope **attributes**, not the request object itself. Say you want to know the HTTP method, which is a *property* of the request object, not an attribute at the request scope, you need to go

through `pageContext`:

```
Method is: ${pageContext.request.method}.
```

The above code works because `pageContext` has a `request` property, and `request` has a `method` property.

The same can be said for other Map scope objects: for example, `applicationScope` **IS NOT** a reference to `ServletContext`, it is just where the application-scope attrbutes are bound.

7.7. EL functions

I am skipping this, see an example on Page 389.

7.8. EL operators

- Arithmetic

```
+ - * / % mod
```

- Logical

```
and     &&
or      ||
not     !
```

- Relational

```
Equals:             ==
Not Equals          !=
Less than           <
Greater than        >
Less or euqal       <=
Greater or equal    >=
```

- Reserved words

```
true                a boolean literal
false               other boolean literal
null
instanceof          not in use but reserved
empty               an operator to see if something is null
                    or empty,

                    e.g. ${empty A}
                    return true if A is null or empty.
```

7.9. Other JSP standard actions

- `<jsp:include>` and `<jsp:param>`: customize content. First, let's show the JSP that does the include:

```
<jsp:include page="header.jspf" >
    <jsp:param name="subTitle" value="Subject assigned to you" />
<jsp:include>
```

The header that is being included can use the new param as:

```
<em><strong>${param.subTitle}</strong></em>
```

- `<jsp:forward`: the buffer is cleared before forward

```
<% if (request.getParameter("userName") == null) { %>
    <jsp:forward page="Handleit.jsp" /> <% } %>

Hello ${param.userName}.
```

7.10. EL Summary

- EL expression are always with curly braces, and prefixed with dollar `$` sign.
- The first named variable in the expression is either an implicit object or an attribute in oe of the four scope (page, request, session, or application).
- The dot operator lets you access values by using a Map key or bean proerty name. Whatever comes to the right of the dot operator must follow normal java naming rules for identifiers, in other words, must start with a letter, underscore, or dollar sign.
- The `[]` operator is more powerful than dot, as you can put expression including named variable within the brackets.
- Don't confuse the implicit EL scope objects (maps of atttributes) with the objects to which the attributes are bound.
- EL functions allow you to call a public static method in a plain old java class. The function name doesn't have to match the actual method name. The mappingis done through TLD (Tag library descriptor) file. To use such a function in JSP, you must delcare namespace using `taglib` directive.

8. Using JSTL

JSP standard tag library, also known as custom tags.

8.1. Loop: `<c:forEach>`

Servlet code:

```
String[] movieList = ["Rush hour", "Return of the King"];
request.setAttribute("movieList", movieList);
...
```

JSP code:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
...
<table>
    <c:forEach var="movie" items="${movieList}" varStatus="movieLoopCount" >
        <tr> <td> Count: ${movieLoopCount.count} </td></tr>
        <tr> <td> ${movie} </td></tr>
    </c:forEach>
</table>
```

The key feature is that the tag assign each element in the collection to the variable you declare with the `var` attribute. The `LoopTagStatus` class has a `count` property that gives you the current value of the iteration.

8.2. Conditional: `<c:if>`

An example: if user is a member, you want to allow them to comment, by including a comment box:

```
<c: if test="${userType == 'member' }" >
    <jsp:include page="inputComments.jsp" />
</c:if>
```

Assuming a servlet somewhere set the `userType` attribute based on the user's login information.

8.3. Switch: `<c:choose>`

For simple case of if else, JSTL has a tedious setup:

```
<c:choose>
    <c:when test="${userPref == 'performance' }">
        ...
    </c:when>

    <c:when test="${userPref == 'safety' }">
        ...
    </c:when>

    ...

    <c:otherwise>
        ...
    </c:otherwise>
</c:choose>
```

8.4. Set value by `<c:set>`, Remove by `<c:remove>`

- Set an attribute variable

```
<c:set var="userLevel" scope="session" value="cowboy" />
```

If there is NOT a session-scoped attribute named "userLevel", this tag creates one (assuming the `value` attribute is not null). If the value is null, then the attribute will be **removed**.

You can also have a body for "large" value:

```
<c:set var="userLevel" scope="session" >
    Sheriff, Bartender, Cowgirl
</c:set>
```

- Work with Beans and Maps

```
<c:set target="${PetMap}" property="dogName" value="Clover" />
```

Target **MUST** evaluate to a object, either a bean or a Map, and it must not be NULL. If the target is a map, then it set the value of a key named "dogName".

- Remove value

```
<c:remove var="userStatus" scope="request" />
userStatus is now: ${userStatus}
```

Nothing prints after the remove, the scope is optional, the default is the page scope.

8.5. `<c:url>` for hyperlink need

In session management, we mention that if client doesn't take cookie, servlet has to encode it to do URL rewriting. So if we have to do it in JSP, here is how it is done:

```
<a href="<c:url value='/inputComments.jsp' />"> Click here </a>
```

`<c:url>` does URL rewritting, but not URL encoding (deal with white space character for example). To do both, we need:

```
<c:url value="/inputComments.jsp" var="inputURL">
    <c:param name="firstName" value="${first}" />
    <c:param name="lastName" value="${last}" />
</c:url>
```

Assuming that:

```
<c:set var="last" value="Hidden Cursor" />
<c:set var="first" value="Crouching Pixels" />
```

Then, you can see the new URL with

```
${inputURL}
```

The output should be:

```
/myApp/inputComments.jsp?firstName=Crouching+Pixels&lastName=Hidden+Cursor
```

9. MVC example (Plain JSP)

9.1. HTML form

```
<form method="POST" action="SelectBeer.do" >

<select name="color" size="1">
    <option> light
    <option> amber
    <option> brown
    <option> dark
</select>
...
<input type="SUBMIT">
</form>
```

9.2. result.jsp

```
<%@ page import="java.util.*" %>
<html><body>
...
<%
    List styles = (List) request.getAttribute("styles");
    Iterator it = styles.iterator();
    while (it.hasNext()) {
        out.print("<br>try: " + it.next());
    }
%>
</body></html>
```

9.3. servlet

```
public class BeerSelect extends HttpServlet {

    public void doPost(HttpServletRequest request,
        HttpServletResponse response) throws IOException, ServletException
        {

        String c = request.getParameter("color");
        BeerExpert be = new BeerExpert()
```

```
        List result = be.getBrand(c);

        // add attribute for JSP to use
        request.setAttribute("styles", result);

        // initiate request dispatcher for JSP
        RequestDispatcher view =
        request.getRequestDispatcher("result.jsp");

        // use request dispatcher to tell container to
        // call JSP, and pass in request and response.
        view.forward(reqeust, response);

    }
}
```

10. Deployment

10.1. Configure Error Page

- Declaring catch all error page

```
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/errorPage.jsp</location>
</error-page>
```

- Declare an error page based on HTTP status code

```
<error-page>
    <error-code>404</error-code>
    <location>/notFoundError.jsp</location>
</error-page>
```

11. Appendix

11.1. `Servlet` interface

The first three are life cycle methods:

```
service(ServletRequest, ServletResponse);
init(ServletConfig);
destroy();
getServletConfig();
getServletInfo();
```

11.2. `GenericServlet` implements `Servlet` interface

An abstract class that implements most of the basic servlet methods you need, but you rarely extend this class.

```
service(ServletRequest, ServletResponse);
init(ServletConfig);
destroy();
...
getInitParameter(String);
getInitParameterNames();
getServletContext();
```

11.3. `HttpServlet` extends `GenericServlet`

Also an abstract class, but reflect the HTTP-ness of the servlet. Notice that `service()` method doesn't just take any servlet request, it is a HTTP-specific request and response.

```
service(HttpServletRequest, HttpServletResponse);
doGet(HttpServletRequest, HttpServletResponse);
doPost(HttpServletRequest, HttpServletResponse);
doHead(HttpServletRequest, HttpServletResponse);
doOptions(HttpServletRequest, HttpServletResponse);
doPut(HttpServletRequest, HttpServletResponse);
doTrace(HttpServletRequest, HttpServletResponse);
doDelete(HttpServletRequest, HttpServletResponse);
getLastModified(HttpServletRequest);
```

11.4. `ServletRequest` interface

```
getAttribute(String);
getContentLength();
getInputStream();
getLocalPort();
getParameter();
getParameterNames();
...
```

11.5. `HttpServletRequest` extends `ServletRequest`

Also an interface, but add methods related to HTTP such as cookies.

```
getContextPath();
getCookies();
getHeader(String);  # request.getHeader("User-Agent") The client's platform and browser info
getQueryString();
getSession();
getMethod();
```

11.6. `ServletResponse` interface

```
getBufferSize();
setContentType();
getOutputStream();
getWriter();
...
```

11.7. `HttpServletResponse` extends `ServletResponse`

```
addCookies();
addHeader();
enccodeRedirectURL();
sendError();
setStatus();
...
```

11.8. `ServletContext` interface

```
getInitParameter(String);
getInitParameterNames();
getAttribute(String);
getAttributeNames()
setAttribute(String, Object);
removeAttribute(String)
...
getRequestDispathcer(String);
```