Article: Gearing up the JSP Performance

By Manivannan Palanichamy. Email: manivannan.p (at) gmail.com

9th Feb 2006

Abstract

The real challenge of any application development is performance. Java Server Pages Technology was unveiled on 2nd June 1999. After the J2EE specification release, JSP plays a very prominent role in application layer.

This articles discusses various performance factors and trade-offs when designing JSP modules for J2EE applications.

Contents

1. JSP Life Cycle and essential methods

i. JSP lifecycle

After developing and testing a JSP file, it is deployed in the container. When the first request is made to the JSP page, the container translates the JSP page into its equivalent servlet program, then compiles it and generates the class file. (However this process is container or application server vendor dependent).

For example, when a “HelloWorld.jsp” file is deployed in Apache Tomcat container, the Tomcat doesn’t take any action on the deployed jsp file initially.

When the first request hits the “HelloWorld.jsp”, the Tomcat translates it into “HelloWorld_jsp.java” servlet (the naming convention is container vendor dependent), then compiles and generates “HelloWorld_jsp.class”.

A simple example follows:

HelloWorld.jsp

```
<%@page language=”java”%>
<%
        out.write(“hello world”);
%>
```

When the first request reaches the container, HelloWorld_jsp.java servlet is generated.The generated servlet will have the code to achieve the same functionality for whatever the JSP page was built for. But, it differs very much in syntax.

The generated HelloWorld_jsp.java servlet comes next.

HelloWorld_jsp.java

```
/*  This is not exact code;
  * Also, this servlet code translation (once again) “vendor    dependent”!
  */

public final class HelloWorld_jsp extends HttpJspBase {

public void _jspInit() {
        // we didn’t override this method, so nothing is inside.
}

public void _jspService (HttpServletRequest request, HttpServletResponse response)
{
        out.write(“hello world!”);  // Hmm.. this is what all we wanted!
}

public void _jspDestroy() {
        // we didn’t override this method too!
}

}
```

A compiled JSP page has the following methods:

1. _jspInit() – This method is executed only once in the life of a JSP.
2. _jspService() – This method is executed for each request coming to the JSP.
3. _jspDestroy() – This method is executed when the JSP page is removed from the container environment

ii. JSP’s init, service and destroy methods

_jspInit() method

_jspInit() method is executed only once in the JSP’s life-time. And hence, it is good practice to initialize the permanent resources inside this method. For example, the JDBC connection initialization code may be placed inside the _jspInit() method.

If the connection code is placed in _jspService () method, a new JDBC connection should be established for each request coming to the JSP page (as the service method is executed every time) and, will increase the number of I/O's. This may degrade the performance.

So, the catch is to override the _jspInit () method and place the initialization code inside it.

Example:

<u>JDBCExample.jsp</u>

```
1. <%@page language="java" import="java.sql.*,javax.sql.*">
2. <%!
3. Connection con;
4.  public void _jspInit() {
5. Con = DriverManager.getConnection (connectionString); // a sample
connection
6.             // Do some Transactions
7. conn.close ();      // at last close the connection
8.   }
9. %>
```

After the JDBCExample.jsp is deployed, the container translates it into "JDBCExample_jsp.java". Then the lines from 3 to 7 are put in _jspInit() method of JDBCExample_jsp.java.

When the first request is made, the _jspInit() method is executed. Then, the JDBC connection is established. This connection establishment happens only once and hence it is optimal.

_jspService ()

The _jspService () method is run by the container for each new request. In its lifetime a JSP page spends the maximum time in _jspService () method. Each line of code present in this method counts against performance. Hence, this method needs to be designed with great care.

In general, this method writes the dynamic and static content to the client (browser). Hence the I/O will be freqeuent in this method.

The followings are some optimal techniques to improve the JSP's Http response and I/O performance.

1. Unnecessary encoding and encryption of data can be avoided since, it takes extra burden to pack and unpack the actual data.

2. The output buffer size can be set larger to support big volume of output (this point is covered in detail later).
3. When the output data (Html) is static, the client can be advised to cache the page content with expiry date and time (using Meta-tags).
4. The out.write() usage to give html output can be minimized and html code could be directly written in the page itself.
5. Lesser the synchronized code, greater the performance.

_jspDestroy ()

This method is called when the JSP page retires from container's environment. This method cane be overridden to claim the resources (initialized in _jspInit () method).

iii. Context path options

The JSP page is loaded only once in the container. It stays permanent then. When the JSP page is modified, the server has to be restarted or the particular application should be reloaded and then the changes will take effect. The container provides an option to reload the file automatically. This option could be disabled to avoid frequent reloading.

(Tomcat specifies this option for each application in <tomcat-home>conf/server.xml file)

```
<Context path="/MyExample" docBase="MyExample" reloadable="false" />
```

iv. .html Vs. .jsp files

When a web page needed to output only static html contents, it can be saved as .html file rather than a .jsp. Because, every .jsp file is taken control by the container, to be compiled, loaded and maintained. So, it slows down the speed.

An example follows.

Example.html

```
<html>
<body>
This is an example.
</body>
</html>
```

When request comes for the Example.html, the webserver server directly transfers the file without concerning the JSP Engine (container)

Example.jsp

```
<%@page language=”java”%>
<html>
<body>
This is an example.
</body>
</html>
```

Now the JSP Engine (container) comes into picture for each request to Example.jsp. But, functionality wise both give the same result. Hence, the former gives the performance (Example.html).

2. JSP Directives

JSP directives instruct the container to process some special functionality. They are dynamic and they count very much against the performance.

i. <%@page %> directive

The page directive has many attributes and we discuss some of them.

extends=”class” – This attribute makes the JSP page to extend the specified class. So, this attribute should be used cautiously. The compiled JSP file may become larger, which may consume large memory and give burden to container.

buffer=”nkb” – This attribute specifies the output buffer size. The buffer size could be made larger to support bulk data output. The default size of the buffer is 8 kb. (The buffer can be autoflushed using ‘autoFlush’ attribute).

ii.<%@include> Vs. <jsp:include>

These are the two essential tags to insert static html contents into JSP files.

```
<%@include file=”add.html”%>
```

The above line inserts the content of add.html file into the jsp file at compile time. It happens only once (at compile time). So, it does not affect the runtime performance.

In case of, <jsp:include> directive,

```
<jsp:include page="add.html" />
```

the file add.html is inserted at runtime . It happens every time the code is executed. It affects the runtime performance.

Hence, the <%@include> is preferable than <jsp:include>.

iii. <jsp:forward> Vs. sendRedirect()

The fundamental difference between the redirect and forward is that the redirect simply commands the browser (or client) to contact some other page and the later passes the request, response objects to the forwarded page on behalf of the browser.

Hence, the sendRedirect() is optimal which reduces the container's overhead of passing the request and response objects.

3. MVC and Custom tags

Recently, MVC and Custom tags have given a new structure for JSP application development. They come with their own frame-works to improve JSP based applications.

i. Trade-offs with MVC

When the application is very complex, MVC gives better solution with features like reusability, logic-presentation separation, modularity and etc. But, the resources (classes, interfaces and API's) used to achieve the functionality is high.

For example, even a simple functionality in MVC needs minimum a Servlet class, Business class and a JSP page. This bulk resource utilization makes the container's performance become poor.

MVC design could be used only for the applications that require it.

ii. Trade-offs with Custom tags

Custom tags give high degree of reusability. But, it is always questionable what components are really going to be reused.Custom tags are very much dynamic. They play hard at runtime. As like MVC framework, custom tags also utilize many classes and interfaces by the name ‘tag handlers’.

Usage of custom tags could be reduced for unnecessary functionality and hence the performance will boost up.

4. Conclusion

So far many techniques were discussed to gear up the JSP performance. The following is the summary of the discussion:

- The _jspInit() method can be overridden to handle the code that should run only once.
- The _jspService() is the prime method, needs careful design for performance. This method spends its maximum time in doing I/O with client (broswer).
- The _jspDestroy() method can be overridden to claim the resources used.
- Reloading option of JSP can be disabled to avoid frequent loading (except development time).
- Files that give only static html output can be named with .html instead .jsp, to directly give the output.
- <%@page%> directive gives many attribute, some of them (extends, buffer) influences the performance.
- <%@inlclude %> directive works at compile time. <jsp:include> plays at runtime, affecting performance.
- sendRedirect() method makes the browser to take diversion, but <jsp:forward> does itself.
- MVC structures the application, but consumes much resources. MVC could be used in necessary cases.
- Custom tags offers reusability, by utilizing the resources. Lesser the custom tags, greater the performance.