

Performance Tuning of Servlet and JSP

Introduction:

In this I will discuss technique to improve performance of Servlet and JSP. These techniques help for developing high performance and scalable JSP (JavaServer Pages) pages and Servlets. That means building applications that are reasonably and consistently fast, and can scale up to the increasing number of users and/or requests. Following are technique for improving performance of Servlet and JSP

- a) HttpServlet init() method
- b) Servlet & JSP auto-reloading
- c) Controlling HttpSession
- d) gzip compression
- e) SingleThreadModel
- f) Thread pool
- g) Include mechanism
- h) useBean action
- i) Miscellaneous techniques

Technique 1: [HttpServlet init() method]

Every Servlet contains `init()` method which is called only once in lifetime. Web Container invoke `init()` method after creating Servlet instance and before handling any client request. We can use `init()` method to improve performance by caching the static data and/or completing the expensive operations that need to be performed only during initialization.

For example:- It is a best practice to use JDBC (Java Database Connectivity) connection pooling, for database operation. We get the `DataSource` from the JNDI (Java Naming and Directory Interface) tree. Performing the JNDI lookup for `DataSource` for every SQL call is expensive and severely affects an application's performance. We should use Servlet's `init()` method to get `DataSource` and cache it for later reuse:

```
public class ControllerServlet extends HttpServlet
{
    private javax.sql.DataSource ds = null;
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
```

```
Context ctx = null;
try
{
    ctx = new InitialContext();
    ds = (javax.sql.DataSource)ctx.lookup("jdbc/hr");
}
catch(NamingException ne)
{
    ne.printStackTrace();
}
catch(Exception e)
{
    e.printStackTrace();
}
}
public javax.sql.DataSource getDS()
{
    return ds;
}
...
...
}
```

Technique 2: [Servlet and JSP auto-reloading]

Servlet/JSP auto-reloading is useful during the development phase because it reduces development time, generally we do not have to restart the server after every change in the servlet/JSP. However, it is expensive in the production phase; servlet/JSP auto-reloading gives poor performance because of unnecessary loading and burdening on the [classloader](#). Also, it may put your application in strange conflicts when classes loaded by a certain [classloader](#) cannot cooperate with classes loaded by the current [classloader](#). So turn off auto-reloading for servlet/JSP in a production environment to receive better performance.

Technique 3: [Controlling HttpSession]

Many applications require a series of client requests so they can associate with one another. Web-based applications are responsible for maintaining such state, called a [session](#), because the HTTP protocol is stateless. To support applications that must maintain state, Java servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions. Sessions are represented by an [HttpSession](#) object. An [HttpSession](#) must be read by the servlet whenever it is used and rewritten when it is updated. We can improve performance by applying the following techniques:

- a) **Do not create HttpSession in JSP pages by default:** By default, JSP pages create HttpSession. If we do not use HttpSession in your JSP pages, disable session by following page directive tag, it will save some performance overhead.

```
<%@ page session="false"%>
```

- b) **Do not store large object graphs inside an HttpSession:** If we store the data in the HttpSession as one large object graph, the application server will have to process the entire HttpSession object each time. This forces Java serialization and adds computational overhead. The throughput decreases as the size of the objects stored in the HttpSession increases because of the serialization cost.
- c) **Release HttpSession when done:** Invalidate sessions when they are no longer needed using the HttpSession.invalidate() method.
- d) **Set session time-out value:** A servlet engine has a default session time-out value set. If we do not either remove the session or use it for the time equal to the session time-out, the servlet engine will remove the session from memory. The larger the session time-out value, the more it affects scalability and performance because of overhead on memory and garbage collection. Try to keep the session time-out value as low as possible.

Technique 4: [gzip compression]

Compression is the act of removing redundant information, representing what we want in as little possible space. Using [gzip](#) (GNU zip) to compress the document can dramatically reduce download times for HTML files. The smaller our information's size, the faster it can all be sent. Therefore, if we compress the content of Web application generates, it will get to a user faster and appear to display on the user's screen faster. Not every browser supports gzip compression, but we can easily check whether a browser supports it and then send gzip-compressed content to only those browsers that do.

Here is the code snippet that shows how to send compressed content whenever possible:

```
public void doGet(HttpServletRequest request, HttpServletResponse
response)
    throws IOException, ServletException
{
    OutputStream out = null
    // Check the Accepting-Encoding header from the HTTP request.
    // If the header includes gzip, choose GZIP.
    // If the header includes compress, choose ZIP.
    // Otherwise choose no compression.
```

```
String encoding = request.getHeader("Accept-Encoding");

if (encoding != null && encoding.indexOf("gzip") != -1)
{
    response.setHeader("Content-Encoding" , "gzip");
    out = new GZIPOutputStream(response.getOutputStream());
}
else if (encoding != null && encoding.indexOf("compress") != -1)
{
    response.setHeader("Content-Encoding" , "compress");
    out = new ZIPOutputStream(response.getOutputStream());
}
else
{
    out = response.getOutputStream();
}
...
...
}
```

Technique 5: [SingleThreadModel]

SingleThreadModel ensures that servlets handle only one request at a time. If a servlet implements this interface, the servlet engine will create separate servlet instances for each new request, which will cause a great amount of system overhead. If we need to solve thread safety issues, use other means instead of implementing this interface. [SingleThreadModel](#) interface is deprecated in Servlet 2.4.

Technique 6: [Thread pool]

A servlet engine creates a separate thread for every request, assigns that thread to the `service()` method, and removes that thread after `service()` executes. By default, the servlet engine may create a new thread for every request. This default behavior reduces performance because creating and removing threads is expensive. Performance can be improved by using the thread pool. Depending on the expected number of concurrent users, configure a thread pool by setting the values for minimum and maximum number of both threads in a pool and increments. At startup, the servlet engine creates a thread pool with number of threads in a pool equal to the minimum number of threads configured. Then the servlet engine assigns a thread from the pool to every request instead of creating a new thread every time, and returns that thread to the pool after completion. Using the thread pool, performance can drastically improve. If needed, more threads can be created based on the values for maximum number of threads and increments.

Technique 7: [Include mechanism]

There are two ways you can include files in a JSP page: include directive (`<%@ include file="test.jsp" %>`) and include action (`<jsp:include page="test.jsp" flush="true" />`). The include directive includes the specified file's content during the translation phase; i.e., when the page converts to a servlet. The include action includes the file's content during the request processing phase; i.e., when a user requests the page. Include directive is faster than include action. So unless the included file changes often, use include directive for better performance.

Technique 8: [useBean action scope]

One of the most powerful ways to use JSP pages is in cooperation with a JavaBeans component. JavaBeans can be directly embedded in a JSP page using the `<jsp:useBean>` action tag. The syntax is as follows:

```
<jsp:useBean id="name" scope="page|request|session|application" class=
  "package.className" type="typeName">
</jsp:useBean>
```

The scope attribute specifies the scope of the bean's visibility. The default value for scope attribute is page. You should select the correct scope based on your application's requirements; otherwise it will affect application performance.

For example, if we need an object only for a particular request, but your scope is set to session, that object will remain in memory even after we are done with the request. It will stay in memory until you explicitly remove it from memory, we invalidate the session, or the session times out as per the session time-out value configured with the servlet engine. If we do not select the right scope attribute value, it will affect the performance because of overhead on memory and garbage collection. So set the exact scope value for the objects and remove them immediately when finished with them.

Technique 9: [Miscellaneous]

- **Avoid string concatenation:** The use of the + operator to concatenate strings results in the creation of many temporary objects because strings are immutable objects. The more you use +, the more temporary objects are created, which will adversely affect performance. Use StringBuffer instead of + when you concatenate multiple strings.

- **Avoid the use of System.out.println:** *System.out.println* synchronizes processing for the duration of disk I/O, and that can slow throughput significantly. As much as possible, avoid the use of System.out.println. Even though sophisticated debugging tools are available, sometimes System.out.println remains useful for tracing purpose, or for error and debugging situations. We should configure System.out.println so it turns on during error and debugging situations only. Do that by using a final Boolean variable, which, when configured to false, optimizes out both the check and execution of the tracing at compile time.
- **ServletOutputStream versus PrintWriter:** Using PrintWriter involves small overhead because it is meant for character output stream and encodes data to bytes. So PrintWriter should be used to ensure all character-set conversions are done correctly. On the other hand, use ServletOutputStream when you know that your servlet returns only binary data, thus you can eliminate the character-set conversion overhead as the servlet container does not encode the binary data.

Conclusion:

In this article we discuss techniques for improving performance of servlet & JSP, which are: use of HttpServlet *init()* method for caching data, disabling of servlet & JSP auto-reloading, controlling HttpSession, use of *gzip* compression, not using of SingleThreadModel, Use of Thread pool, choose the right include mechanism, select the right scope in useBean action, and Miscellaneous techniques. Some of these techniques apply during the development phase, i.e., while we design an application and write the code. And some of these techniques used for configuration.