Monitoring performance in a WebSphere Portal environment

A guide to help you understand some basics in monitoring and measuring performance problems in a WebSphere Portal environment

Jesse T. Bickmore
Dharmesh Mistry
Kevin Dzwonchyk
Gourishankar Menon

October 14, 2008

This article introduces the various ways that you can monitor performance in an IBM® Websphere® Portal environment. These monitoring methods are useful in situations where a portal-based application is experiencing poor performance, such as slow loading pages, sporadic problems, or stability issues. The article looks at areas such as caching, JVM monitoring, database analysis, logging, and debugging to aid in monitoring and measuring performance.

To tune your WebSphere Portal environment for optimal performance, you need to know what needs tuning. The monitoring methods discussed in this article cover several key areas of WebSphere Portal. They help you to view the true behavior of your WebSphere Portal environment and ultimately identify bottlenecks and potential problems. This article is meant to be an overview of monitoring methods rather than an in-depth look at any specific methods. This article provides a good starting point for most people tasked with resolving these types of problems. Our goal is to show you enough approaches so that you can resolve performance issues in a timely, cost-efficient manner.

NOTE: The Related topics section includes several links that can help in the performance tuning process.

This article discusses the following topics for monitoring performance:

- Caching
- JVM monitoring
- Database analysis
Monitoring performance in a WebSphere Portal environment

- Clustering
- Logging and debugging
- Custom page monitoring
- IBM Tivoli® Composite Application Manager (ITCAM)

Caching

In this section, we focus on discussing portal internal caches and Dynacache.

Internal portal caches

WebSphere Portal is a complex application that uses caching extensively through WebSphere Application Server. Tuning these caches for your specific environment is vital to a well-performing portal. It is not easy to monitor these caches with a standard installation, and many times you rely on turning on the correct logging level to see how a specific cache is being used. This approach is an inefficient method that can be time-consuming and painful. A portlet known as the Internal Portal Cache Listing portlet, however, allows you to monitor internal portal caches and to view important cache statistics including:

- A count of current cached entries for each cache
- A count of the highest number of cached entries for each cache
- The configured sized of each cache.

Figure 1 shows the csv format that the portlet provides.

**Figure 1. Sample image of portal internal cache listing portlet**

<table>
<thead>
<tr>
<th>Cache Name</th>
<th>Current number of entries</th>
<th>Observed hit rate</th>
<th>Configured maximum size</th>
<th>Hits</th>
<th>Expire</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.cache.search.cache.dynacache.CacheSomeholder</td>
<td>9</td>
<td>1000</td>
<td>87.59%</td>
<td>24</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>com.ibm.cache.search.cache.dynacache.CacheSomeholder</td>
<td>1</td>
<td>1000</td>
<td>90.00%</td>
<td>21</td>
<td>21</td>
<td>-1</td>
</tr>
<tr>
<td>com.ibm.cache.ApplicationRuleChildrenCache</td>
<td>0</td>
<td>1000</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>com.ibm.cache.ApplicationRuleChildrenCache</td>
<td>0</td>
<td>1000</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>com.ibm.cache.ApplicationRuleChildrenCache</td>
<td>0</td>
<td>1000</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>com.ibm.cache.ApplicationRuleChildrenCache</td>
<td>0</td>
<td>1000</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>com.ibm.cache.ApplicationRuleChildrenCache</td>
<td>7</td>
<td>1000</td>
<td>95.75%</td>
<td>69</td>
<td>14</td>
<td>-1</td>
</tr>
<tr>
<td>com.ibm.cache.ApplicationRuleChildrenCache</td>
<td>7</td>
<td>1000</td>
<td>95.75%</td>
<td>69</td>
<td>14</td>
<td>-1</td>
</tr>
<tr>
<td>com.ibm.cache.ApplicationRuleChildrenCache</td>
<td>0</td>
<td>1000</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>com.ibm.cache.ApplicationRuleChildrenCache</td>
<td>0</td>
<td>1000</td>
<td>NA</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

Setting the correct cache sizes and lifetime values is critical in achieving optimal performance:

- If a particular cache is too small, then the WebSphere Portal server can go to the database too often, which can result in performance degradation.
- If a cache is too big, then it can waste memory in the JVM and could result in low memory conditions or increased paging, both of which result in performance degradation.
- A cache that expires too often can result in unnecessary performance degradation; therefore cache lifetime values are also important to tune.

Also, keep in mind that portal caching behavior, and therefore performance, can change when upgrading between WebSphere Portal versions. It is important to check the caches any time that you investigate performance, and the Internal Portal Cache Listing portlet that can do just that.
When it is time to tune the cache sizes and performance, it is important that you use the data provided by this portlet in conjunction with the WebSphere Portal Performance Tuning Guide (see Related topics). This approach helps you set the optimal values for your specific environment and workload.

**Dynacache**

The WebSphere Application Server cache monitor is also a useful tool to help monitor your application. For portlets that are configured for caching it can show you:

- How many hits and misses you are getting
- How full your cache is
- How many items were evicted from the cache using the LRU (least recently used) algorithm
- If servlet caching is turned on

Figure 2 demonstrates the information it provides.

**Figure 2. Dynacache statistics screen sample**

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
<td>20300</td>
</tr>
<tr>
<td>Linked Entries</td>
<td>1853</td>
</tr>
<tr>
<td>Cache Hits</td>
<td>53751</td>
</tr>
<tr>
<td>Cache Misses</td>
<td>33330</td>
</tr>
<tr>
<td>LRU Evictions</td>
<td>440</td>
</tr>
<tr>
<td>Explicit Removals</td>
<td>37421</td>
</tr>
<tr>
<td>Default Priority</td>
<td>1</td>
</tr>
<tr>
<td>Servlet Caching Enabled</td>
<td>Yes</td>
</tr>
<tr>
<td>Disk Offload Enabled</td>
<td>No</td>
</tr>
</tbody>
</table>

As you should know, the output of a portlet can be cached when servlet caching is enabled on the WebSphere Application Server. In many cases, caching is disabled, or the portlets themselves are not configured to take advantage of caching. How to configure a portlet to be cached is out of the scope of this article, but there are some useful references that can help (see "Caching Portlet Output" and "Develop high performance Web sites with both static and dynamic content using WebSphere Portal V5.1" in Related topics). Keep in mind that caching is not desired for certain portlets, especially when using personalization. For portlets whose content is shared among users, caching can significantly improve overall performance.

**JVM monitoring**

As with any Java™ 2 Platform, Enterprise Edition application server, the Java virtual machine (JVM) is the key component responsible for a majority of the processing. In a WebSphere Portal environment, every page that is rendered is processed by the JVM. There are various components to a JVM, and each has a varying effect on the overall performance of a WebSphere Portal site. Monitoring the top-level components, such as the heap, servlet threads, and DB connection pool, provides an insight into what is happening when a request is being processed by the JVM. It also provides an indication to the location of potential bottlenecks.
Using the WebSphere Performance Monitoring Infrastructure (see Related topics), you can gather performance data from various WebSphere Application Server components and key parts of the JVM. By combining this infrastructure with the IBM Tivoli Performance Viewer (see Related topics), which is a Java client to display and monitor performance data, you can view performance monitoring interface (PMI) data without writing any custom code. The Tivoli Performance Viewer also includes an advisor that recommends tuning changes based on the data.

Of course, you can write custom monitoring code using the Java Management Extensions (JMX) API (see Related topics) provided by WebSphere. The first step is to turn on the Performance Monitoring Service and set the specification level to low. Refer to the WebSphere Library (see Related topics) for details on how to turn on the PMI service. By using JMX, you can easily write Java code to automatically poll application server metrics and record the data for analysis. The data collected can be stored in a comma-separated value (CSV) file, which can then be imported into most graphing tools, such as Microsoft® Excel and OpenOffice. By graphing the data, you can easily identify trends and patterns.

Sample code for creating the administrative client and getting the PMI values is also shown in the WebSphere Library. You can also refer to the developerWorks® article, "Writing a Performance Monitoring Tool". Including sample code here is out of the scope of this article.

Certain metrics worth monitoring include Java Database Connectivity (JDBC), JVM memory usage, servlet transport threads, and database connections. When combined with IBM HTTP Server thread statistics, this level of monitoring can be a powerful method of getting insight into the hosting environment.

Figure 3 shows a five-node WebSphere Portal environment that has two application servers per node.
Database analysis

The database is another key area that you should examine when looking into performance-related issues. WebSphere Portal uses the database extensively to store information. Specifically, monitor the following databases and optimize them in the following order for the best performance:

- Portal database
- LDAP database
- Application-specific databases

If any of these databases is running IBM DB2®, then you have a few options available to help monitor and tune your database performance. The two basic strategies of the DB2 database system monitor are these:

- **Snapshot monitors.** Let you capture the state of a database at a specific point in time.
- **Event monitors.** Allow you to capture and log monitor events as they occur.

The results of both monitors are stored in monitor elements. The following is a list of available monitor elements:

- **Counters.** Show the total number of times an event has occurred.
- **Gauges.** Indicate the current value for an item.
- **Watermarks.** Indicate the maximum and minimum value that an item has reached.
- **Information elements.** Show reference details.

Figure 3. JVM monitoring sample
• **Timestamps.** Indicate the date and time that an activity took place.
• **Time elements.** Show the amount of time spent performing an activity.

Additional details on the DB2 database system monitor can be found in the following articles: "Performance Monitoring, Part 1: It's a Snap(shot)" and "Performance Monitoring, Part 2."

If you are running DB2 on z/OS®, you should download the DB2 Performance Monitor for z/OS systems (see Related topics). This monitor is a great asset for identifying long-running SQL statements, locking conflicts and storage consumption.

You should monitor the number of database connections during normal activity. It is important to know the connection workload so that the connection pool settings within WebSphere Application Server can be properly tuned. Figure 4 shows a screen capture from a custom tool to monitor database connections. Keep in mind that a custom tool does not have to be written; this customization was done to make the process easier for developers who don't have access to production systems directly.

**Figure 4. Monitoring DB connections for various databases**

![Figure 4. Monitoring DB connections for various databases](image)

### Clustering

Tracking issues in a large-scale environment can be daunting. It is often difficult to reproduce an issue on just one clone. The Workload Manager (WLM) in WebSphere does load balancing and can direct traffic to any available clone in a cluster or cell defined for the application. You can hit a clone directly through the WebSphere port defined for the WebSphere Application Server, but in most environments that is blocked by a firewall. To go around this block, you can use IBM HTTP Server (IHS) (Apache) rules that direct traffic to a clone based on URL query strings.

The first thing to do is to identify the names for your clones. You can use the name specified in the WebSphere Application Server administrative console. You also need to know the port from which the clone is served. See figure 5.
After you have this information, you can edit the IHS (Apache) conf files with a new stanza. The http.conf file can be located usually by performing `ps -ef|grep http` and identifying the conf file in use. If your environment is new, refer to your installation instructions. For this example, the keyword is cloneID, and the CloneNames are `WebSphere_Portal` and `WebSphere_Portal_Clone_2` with ports 9085/9086. The target WebSphere Application Server hostname is `host name.ibm.com`. See listing 1.

Listing 1. New stanza

```
<LocationMatch "cloneID$" >
  RewriteEngine on
  RewriteCond %{QUERY_STRING} ^WebSphere_Portal_Clone_2
  RewriteRule /(.*)/cloneID$ http://host name.ibm.com:9086/$1 [P]
  RewriteCond %{QUERY_STRING} ^WebSphere_Portal
  RewriteRule /(.*)/cloneID$ http://host name.ibm.com:9085/$1 [P]
</LocationMatch>
```

This stanza, after it is added to all your IHS servers, allows you to obtain session affinity with a specific clone. To use this stanza, for example, you access the portal application and append `?cloneID=WebSphere_Portal_Clone_2` to the URL, and your request is sent to `WebSphere_Portal_Clone_2`.

This method is useful in large-scale environments where you have many clones and you need to debug or reproduce a problem, and you don't want to search through every clone on each server to find where your request went. It is also a great tool for finding issues that are clone-specific or when you want to measure performance times of components based on log timestamps.

### Logging and debugging

#### Portal and WebSphere logging

You can enable debug logging for different Websphere Portal components either in the properties file `<WP Root>/shared/app/config/log.properties` or by using the WebSphere Application Server
administrative console under Troubleshooting - Logs and Trace - <Application Server Name> - Diagnostic Trace Service. To enable debug logging within the console, select Enable Trace and then provide traceString name/value pairs to enable debug logging for particular components. Make sure to specify an output file name, and then click Apply. The trace files can get large quickly, so ensure that the location that you specify is sufficient enough to store the output.

Figure 6 shows the console settings.

**Figure 6. Portal logging using the console**

As an example, two types of traceStrings can be useful:

- **WMM**. Enabling WebSphere member manager (WMM) allows you to check the calls to your external membership repository (usually an LDAP server). You can use this type to identify large results or groups within groups.

  traceString=com.ibm.websphere.wmm.*=all=enabled;com.ibm.ws.wmm.*=all=enabled;WSMM=all=enabled;com.ibm.ws.security.*=all=enabled

- **URL mapping**. Enabling URL mapping allows you to monitor and count the amount of URL mappings that are called per page/label request. Doing this mapping in a controlled environment gives you a good starting point in tuning your mapping cache limits.

  traceString=com.ibm.wps.mappingurl.*=all=enabled;com.ibm.wps.command.mappingurl.*=all=enabled;com.ibm.wps.engine.*=all=enabled

For more information on other traceStrings, refer to the WebSphere Portal run-time logging topic (see Related topics) in the WebSphere Portal Information Center. You can also enable
WebSphere Portal tracing from within the WebSphere Portal administration user interface, under Portal Analysis - Enable Tracing. These traces are applied on the fly, and they do not persist after restarts, but they can be useful in debugging. See figure 7.

**Figure 7. Portal tracing at runtime**

If your application uses Log4J (see Related topics), there is a servlet that you can use to dynamically enable and disable Log4J logging levels without recycling the server. This servlet can make troubleshooting and development easier. Follow these steps:

1. To install the servlet, first obtain the source code (ConfigurationServlet.java) from the Apache Sandbox (see Related topics).
2. Compile the code and place the class file in `<WP root>/shared/app/org/apache/log4j/servlet/ConfigurationServlet.class`.
4. Add the code shown in listing 2 to the end of your list of servlets:

   **Listing 2. Code for list of servlets**

   ```xml
   <servlet>
     <servlet-name>log4j</servlet-name>
     <display-name>Log4j configuration Servlet</display-name>
     <servlet-class>org.apache.log4j.servlet.ConfigurationServlet</servlet-class>
   </servlet>
   ```

5. Add the code shown in listing 3 to the end of your list of servlet-mappings.

   **Listing 3. Code for list of servlet mappings**

   ```xml
   <servlet-mapping id="log4j-Config">
     <servlet-name>log4j</servlet-name>
     <url-pattern>/log4j/*</url-pattern>
   </servlet-mapping>
   ```

After recycling your server, you can quickly load your Log4J configuration and set the logging level dynamically using the URI of your base context:

```
http://<hostname>/wps/log4j
```

Figure 8 shows the interface:
Custom page monitoring

To identify the cause of slowly loading pages, especially when the problem is sporadic, it can be useful to add debug timing information into the generated HTML source. These timings can be for:

- Individual portlets
- Total portlets
- Servlet filters
- Masthead
- Left navigation
- Custom application processes

It is not always necessary to implement custom page monitoring to achieve performance data. Often the performance monitoring infrastructure within WebSphere Portal provides the necessary data. For instance, it is possible to monitor the number of sessions or time spent within requests using PMI at the Web application level alone.

For custom monitoring, however, our strategy is to output the timings as HTML comments so that only by viewing the source can you see the comments. If you are not monitoring in a production environment, then the timings can be displayed as part of the HTML on the page. You can see that this strategy is an efficient way to monitor even production environments without introducing significant overhead for performance.

The easiest way to add this timing information is to modify your theme JSPs. If you do not have a custom WebSphere Portal theme, you can find the existing theme JSPs at:
/WebSphere/AppServer/installedApps/<clone>/wps.ear/wps.war/themes

To add debug timing information, for example, you can update the Default.jsp of a theme with the following line early in the file:

```jsp
<% long start = java.lang.System.currentTimeMillis(); %>
```

This update initializes the start time for rendering the page. To output the time rendered since this point, the following line can be used later in the JSP:

```jsp
<!-- TOTAL TIME: <%= java.lang.System.currentTimeMillis() - start %>ms -->
```
The idea is to place code similar to this debug information around key areas of the rendered page. Then, when rendering a page, you can determine the time that is spent rendering each key area by viewing the HTML source. Depending on the breakdown of what you want to measure, it might be necessary to reset the start variable’s value at certain points. You might want to measure the time spent rendering the left navigation or masthead links, for example. If you have a custom application running, then your custom code might be invoked during these rendering phases.

To achieve timings on individual portlets in this manner, you'll first want to find the part of your theme's Default.jsp which renders the content space and reset the timer immediately before it. In this example, we also output the total time spent rendering all the portlets on the page as shown in listing 4:

**Listing 4. Total time spent rendering**

```
<% start = java.lang.System.currentTimeMillis(); %>
<%-CONTENTSPACE--> <wps:screenRender />
<%-/CONTENTSPACE-->
<%- PORTLET TOTAL TIME:
<%= java.lang.System.currentTimeMillis() - start %>ms -->
```

Then (and here's the trick), you need to edit the WebSphere Portal server's UnlayeredContainer-H.jsp file, located under:
/WebSphere/AppServer/installedApps/<clone>/wps.ear/wps.war/skins

Sometime immediately before or after the call to model.render(child); where it is iterating over the model, you want to add the lines:

before

```
<% start = java.lang.System.currentTimeMillis(); %>
```

after

```
<%= java.lang.System.currentTimeMillis() - start %>ms -->
```

Listing 5 shows the larger code excerpt.
Listing 5. Call to model.render(child)

for (Iterator iterator = model.getChildren(currentElement); iterator.hasNext();) {
    CompositionNode child = (CompositionNode) iterator.next();
    CompositionMetrics childMetrics = child.getMetrics();
    start = java.lang.System.currentTimeMillis();
}%>

<td valign="top" <% String width = (String)childMetrics.getValue(childMetrics.WIDTH);
    if (width != null) {
        out.print("width=");
        out.print(width);
        out.print("\"");
    } %>>
%>

<% model.render (child); %>
</td>
<!-- PORTLET TIME: <%= java.lang.System.currentTimeMillis() - start %> -->
<% } %>

The effect of placing this code here is that for each portlet rendered on the page, you have an HTML comment placed below it that displays how many milliseconds it took to render that portlet. Note that, due to the scope of the variables in the JSPs, you are not resetting the same start variable from Default.jsp.

**NOTE:** This particular strategy for showing individual portlet rendering times has been tested only when parallel portlet rendering is turned off.

What is the advantage of adding debug timing information to the HTML comments in this manner? First of all, after this information has been instrumented in your code, it is much easier to access individual component timings than digging through log files and debug output, especially when the problem is sporadic. Often, it is difficult to correlate back-end log files with a particular request. With this method, whenever a slowly loading page occurs, you can (manually or programatically) check the HTML source to determine the cause of the delay and ultimately break down the total page load time into individual components.

Furthermore, using this method, it is easy to write a custom application that probes your WebSphere Portal pages using polling and periodically loading the pages. The performance of the individual component timings can be extracted, graphed, and viewed over time. This method can identify trends in performance and performance spikes. For example, in one case study, it was found that particular WebSphere Portal caches were expiring prematurely on a regular basis and causing performance spikes in custom application code. After the individual components were graphed, it was easy to spot this problem.

**Apache HTTP client libraries for writing a custom tool**

For developers wanting to pursue writing a custom monitor and graphing tool, you can use the open source Jakarta Apache HTTP client libraries (see Related topics).
This client provides an easy way for developers to program typical browser interactions using the HTTP protocol. It supports cookies, SSL, and all HTTP commands. To show you how easy it is, listing 6 shows a code snippet that performs an HTTP GET on a URL:

Listing 6. HTTP GET

```java
HttpClient client = new HttpClient();
client.getState().setCookiePolicy(CookiePolicy.COM Patibility);
GetMethod getMethod = new GetMethod(url);
getMethod.setFollowRedirects(true);
int statusCode = client.executeMethod(getMethod);
String htmlData = getMethod.getResponseBodyAsString();
```

There is no limit to what you can measure with this method. If you have a custom application running, you can measure any step in processing a request. The timings for these steps can be added to the request object as attributes, then printed out to the HTML comments later on in the JSP with code such as this:

<!-- SERVLET FILTER TIME: <%= ((Long)request.getAttribute("servletEndTime"))\n.longValue() - ((Long)request.getAttribute("servletStartTime"))\n.longValue() %&gt;ms -->

**NOTE:** With the preceding example, it is necessary to add custom code into the application, which adds the appropriate start and end times for the measured components, which in this case are the overall time spent in servlet filters.

As an example of the kind of information this code can generate, figure 9 shows a custom application that polls a WebSphere Portal page and graphs the timings of individual components extracted from the HTML source.

**Figure 9. Custom application timing graph**

![Custom application timing graph](image)

**IBM Tivoli Composite Application Manager**

Many application today are quite complex and span various technologies and software such as Web servers, application servers, databases, and backend systems. Typical monitoring tools work well for each individual area, but there are not many tools available that allow you to monitor these composite applications as a whole.

Tivoli Composite Application Manager consists of two main components: a managing server and data collectors. The data collectors reside on your application servers and collect the desired...
information, while the managing server pulls together all the information from all the data collectors and allows you to analyze it.

Some features of Tivoli Composite Application Manager include the following:

- Monitoring on demand (MOD) allows the set up of a schedule to capture a percentage sampling of requests during a timeframe in which you suspect that there are issues. Tooling can then be used on the managing server to locate and analyze the slower transactions. Monitoring on Demand allows you to set different levels of monitoring from production mode (basic data), to problem determination mode, all the way to tracing mode, which lets you get to a low-level details.
- In-flight request search lets you analyze real-time requests on the system.
- It allows the analysis of your application's performance based on historical analysis and trends.
- It allows you to set the traps to detect and troubleshoot problem areas.
- It provides memory diagnostics to aid in detecting and fixing memory leaks
- Provides out of the box reporting tools to assist in problem analysis.

Tivoli Composite Application Manager is a useful tool in complex applications that span various applications and subsystems. It can provide useful information, including analyzing real time requests, looking at trends and helping find issues within your application.

**Conclusion**

To efficiently address Websphere Portal-related performance issues, it is important to have good monitoring and measurements of your environment. Having an accurate view of your WebSphere Portal environment helps you understand the problems and their causes.
Related topics

- Consult the WebSphere Portal zone.
- Refer to the WebSphere Portal Support site.
- Read about the Caching Portlet Output details in the WebSphere Portal Information Center.
- Learn more about the WebSphere Portal run-time logging trace strings.
- Refer to the "WebSphere Portal 5.1 Performance Tuning Guide"
- Read the article, "Develop high performance Web sites with both static and dynamic content using WebSphere Portal V5.1."
- Read the article, "Performance considerations for custom portal."
- Read the article, "DB2 Performance Monitoring, Part 1: It's a Snap(shot)."
- Read the article, "DB2 Performance Monitoring, Part 2."
- Refer to the WebSphere Library.
- Read the article, "A Technique for Monitoring Websphere Application Server Workload Management" to learn how to use standard Web server directives to log workload management activity in the WebSphere Application Server cluster and a mechanism for summarizing the behavior of WebSphere Application Server workload management in real time..
- Learn more about Java Management Extensions (JMX).
- Read about Tivoli Performance Viewer.
- "WebSphere Performance Monitoring Infrastructure"
- Read the article, "Writing a Performance Monitoring Tool."
- Learn more about IBM Tivoli Composite Application Manager for SOA.
- Learn more about IBM Tivoli Composite Application Manager for WebSphere.
- Consult the Apache Sandbox.
- Read about the Jakarta Commons HTTP Client.
- Refer to the JFreeChart.
- Learn about Log4J.