Lab 3. Performance Tuning

Visualize that you are on a vacation, enjoying the trip, and at 2 o'clock in the morning, your phone rings. You quickly pick up your phone to find out that your boss is calling to say that ABC Company's web system is down and you have to come online and fix the issue. If you don't want to face these sort of situations, please read this lab with more attention.

In this lab, we will cover the major topics of performance tuning including the following:

- Memory related issues
- JVM parameter optimization
- OS level optimization to improve the performance

Performance tuning for Tomcat 8

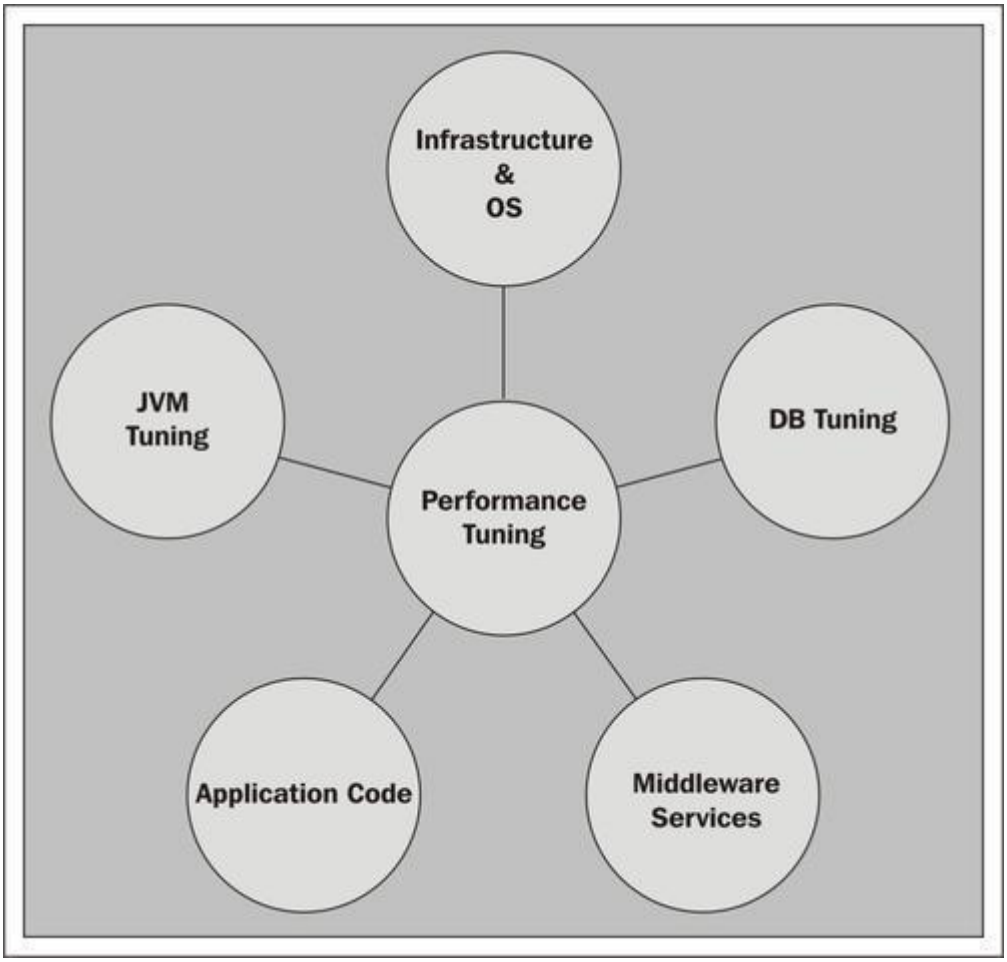
Performance tuning plays a vital role to run a web application without downtime. Also, it helps in improving the performance of Tomcat while running the applications. Tuning of the Tomcat server may vary from application to application. Since every application has its own requirements, it is a very tricky task to tune Tomcat 8 for every application. In this topic, we will discuss tuning of various components of Tomcat, and how they are useful in the server performance. Before we start with the configuration changes, let us quickly discuss why we need to tune Tomcat.

Why we need performance tuning?

People always ask, why do we need to do performance tuning for Tomcat 8 when, by default, Tomcat 8 packages are customized for the production run.

The answer to this question is very subjective; every web application has its own requirement. Some of the applications require high memory while others require less memory but a high GC pause. It varies from application to application and the administrator has to tune Tomcat based on the application's requirement.

Performance tuning is still incomplete with JVM tuning. There are multiple other things which also play a key role in the performance of an application, such as database configuration, OS level setting, and the hardware used for the application. The following figure shows the different types of performance tuning done on Tomcat 8:



There are different aspects, which have an impact on the performance of Tomcat 8, mentioned as follows:

- **Application Code:** If the application code is not developed properly, it may cause performance issues. For example, if the database connection from the application code is not closed properly, then it will create a stale connection in the database, hence, causing the application to run slowly.
- **Database Tuning:** The database can cause major issues in the performance of the application hosted in Tomcat. For example, if the database response is slow, then it sends a delayed response to the application query, hence, causing the application to run slowly in Tomcat 8.
- **JVM Tuning:** Every application has its own memory requirement. If an application (say abc) has a very huge memory requirement and you have allocated less memory, then you might face OOM (out of memory) issues, hence, causing performance issues.
- **Middleware Services:** Middleware services may lead to application connectivity issues with the external interface. For example, nowadays, mobile applications use web services to connect to the server and fetch the application data. In this way, we are only exposing web services to the Internet and not the entire application server.
- **Infrastructure and OS:** Infrastructure issues may also lead to performance degradation. For example, if there is an Internet connectivity issue with the network or there is a packet drop in the network, it causes degraded performance.

Note

In 70 percent of cases, the reason for bad performance of the application is incorrect code. There are various reasons such as improper closing of loops, connections are not closed, and so on.

Performance tuning for Tomcat 8

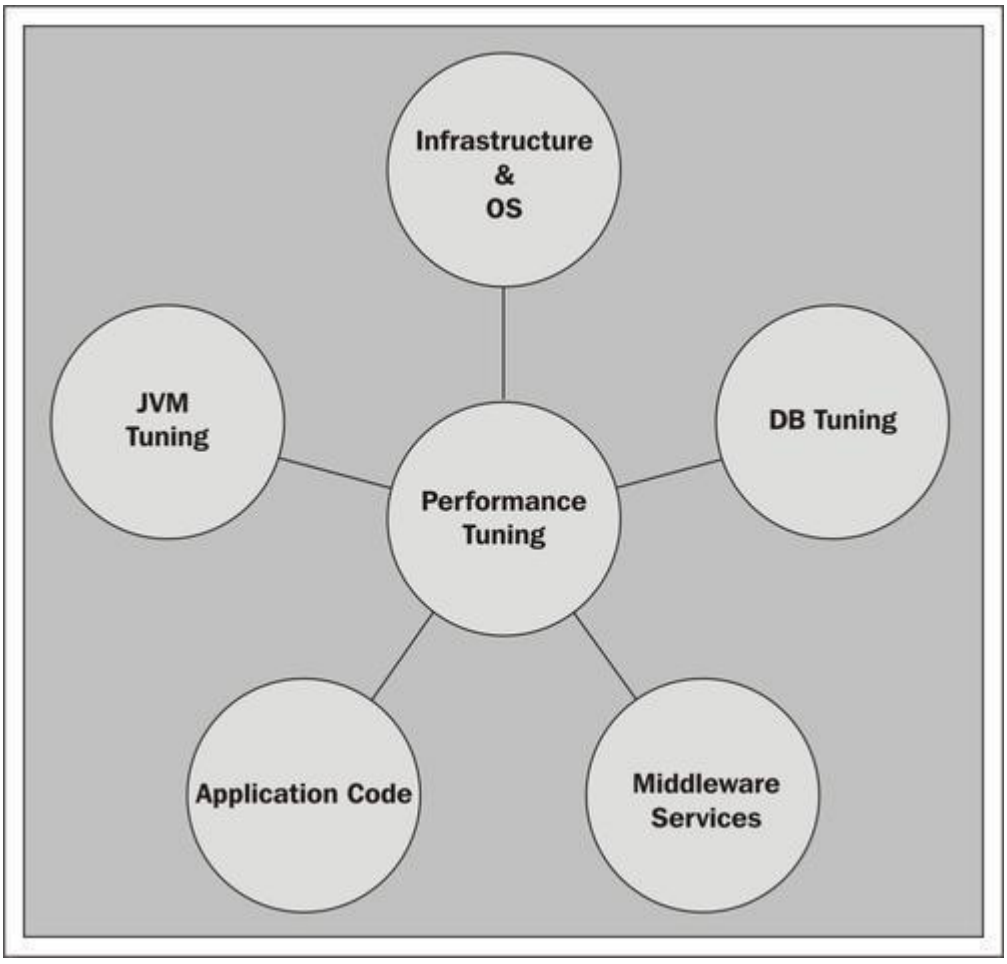
Performance tuning plays a vital role to run a web application without downtime. Also, it helps in improving the performance of Tomcat while running the applications. Tuning of the Tomcat server may vary from application to application. Since every application has its own requirements, it is a very tricky task to tune Tomcat 8 for every application. In this topic, we will discuss tuning of various components of Tomcat, and how they are useful in the server performance. Before we start with the configuration changes, let us quickly discuss why we need to tune Tomcat.

Why we need performance tuning?

People always ask, why do we need to do performance tuning for Tomcat 8 when, by default, Tomcat 8 packages are customized for the production run.

The answer to this question is very subjective; every web application has its own requirement. Some of the applications require high memory while others require less memory but a high GC pause. It varies from application to application and the administrator has to tune Tomcat based on the application's requirement.

Performance tuning is still incomplete with JVM tuning. There are multiple other things which also play a key role in the performance of an application, such as database configuration, OS level setting, and the hardware used for the application. The following figure shows the different types of performance tuning done on Tomcat 8:



There are different aspects, which have an impact on the performance of Tomcat 8, mentioned as follows:

- **Application Code:** If the application code is not developed properly, it may cause performance issues. For example, if the database connection from the application code is not closed properly, then it will create a stale connection in the database, hence, causing the application to run slowly.
- **Database Tuning:** The database can cause major issues in the performance of the application hosted in Tomcat. For example, if the database response is slow, then it sends a delayed response to the application query, hence, causing the application to run slowly in Tomcat 8.
- **JVM Tuning:** Every application has its own memory requirement. If an application (say abc) has a very huge memory requirement and you have allocated less memory, then you might face OOM (out of memory) issues, hence, causing performance issues.
- **Middleware Services:** Middleware services may lead to application connectivity issues with the external interface. For example, nowadays, mobile applications use web services to connect to the server and fetch the application data. In this way, we are only exposing web services to the Internet and not the entire application server.
- **Infrastructure and OS:** Infrastructure issues may also lead to performance degradation. For example, if there is an Internet connectivity issue with the network or there is a packet drop in the network, it causes degraded performance.

Note

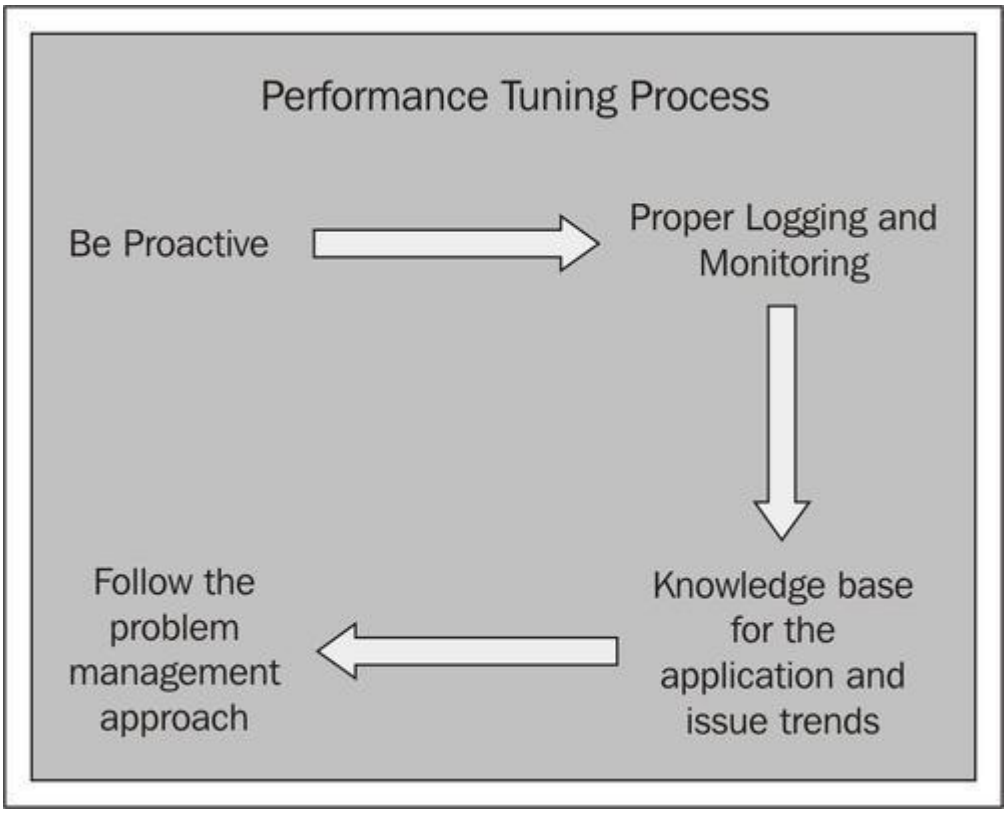
In 70 percent of cases, the reason for bad performance of the application is incorrect code. There are various reasons such as improper closing of loops, connections are not closed, and so on.

How to start performance tuning

Performance tuning starts from the day the application deployment stage begins. One may ask, as the application is only in the development phase, why do we need to do performance tuning now?

At the time of the application development, we are in the state to define the architecture of the application, how the application is going to perform in reality, and how many resources are required for an application. There are no predefined steps for performance tuning. But there are certain thumb rules which all administrators should follow for performance tuning.

The following figure shows the process flow for performance tuning:



- **Be Proactive:** Think about the various ways an application failure may arise in future (bottlenecks for the application). Then prepare a solution for the application to avoid the failure. Keep track of application issues and use the latest profiling tool available in the market.

Note

If you are proactively checking the application and avoid bottlenecks for the application, then around 70 percent of the issues are solved in the production environment. This approach will give your customers great satisfaction when using the application.

- **Proper Logging and Monitoring:** Proper logging should be enabled for Tomcat, this will be really helpful in tracing the potential issues which are about to occur in the production system. We will discuss different methods of enabling logger in Lab 6.

Note

There are many monitoring tools available in the market, which give you a real picture of the application. Major enterprises use these kind of tools for profiling the application. For example, JON(Jboss on Network), CA Wily, Nagios, Panorama, and so on.

- **Knowledge base for the application and issue trends:** It is always recommended to have a knowledge base for the application as the application is running 24x7. Also, we should have a good documentation for the application's support, so that everyone who is working on this application has the same information on the issues.
- **Follow the problem management approach:** Make it good practice to perform the **Root Cause Analysis** (**RCA**) for every issue. This way recurrence of this issue can be avoided in the future.

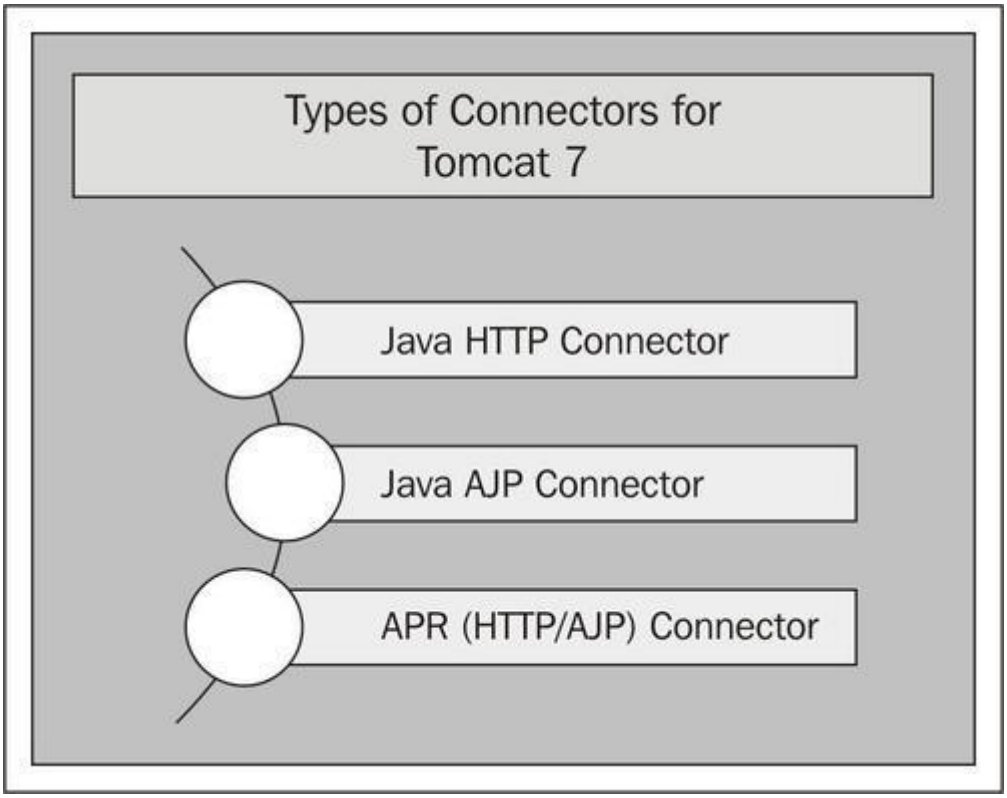
We have discussed the approaches for tuning Tomcat 8. Now, it's time to do real-time performance tuning for Tomcat 8.

Tomcat components tuning

In Tomcat 8, you can do many configurations to improve the server performance, threads tuning, port customization, and JVM tuning. Let's quickly discuss the major components of Tomcat 8 which are important for performance improvement.

Types of connectors for Tomcat 8

Connectors can be defined as a point of intersection where requests are accepted and responses are returned. There are three types of connectors used in Tomcat 8, as shown in the following figure. These connectors are used according to different application requirements. Let's discuss the usability of each connector:



Java HTTP Connector

The Java HTTP Connector is based on the HTTP Protocol, which supports only the HTTP/1.1 protocol. It makes the Tomcat server act as a standalone web server and also enables the functionality to host the JSP/servlet.

Note

For more information on the HTTP Connector, please visit http://tomcat.apache.org/tomcat-7.0-doc/config/http.html.

Java AJP Connector

The Java AJP Connector is based on the **AJP** (**Apache JServ Protocol**) and communicates with the web server through the AJP. This connector is mainly used when you don't want to expose your Java servlet container to the Internet (using a different frontend server) and is also very helpful where SSL is not terminating in Tomcat 8. Some of the examples where the AJP is implemented are, mod_jk, mod _proxy, and so on.

Note

For more information on AJP Connectors, please visit the URL http://tomcat.apache.org/tomcat-7.0-doc/config/ajp.html.

APR (AJP/HTTP) Connector

Apache Portable Runtime (**APR**) is very helpful where scalability, performance, and better collaboration are required with different web servers. It provides additional functionalities such as Open SSL, Shared memory, Unix sockets, and so on. It also enables Java to emerge as a web server technology rather than being absorbed on backend technology.

Note

For more details on the APR Connector, please visit the link http://tomcat.apache.org/tomcat-7.0-doc/apr.html.

Thread optimization for Tomcat 8

Thread tuning plays a major role in Tomcat's performance. In most cases, we have seen that a particular application works very well on other industries, but when we implement the same application, its performance is degraded (application performance issues). The reason being that there may be the chance that we have done an improper tuning of the thread, which may lead to server-degraded performance. Let's discuss the different components of thread tuning thread pools.

The thread pool can be defined as the capacity of the web server to accept the number of connections or requests handled in Tomcat 8. We can configure two types of thread pool; Shared pool and Dedicated pool. These configurations need to be done in `server.xml` placed in `TOMCAT_HOME/conf/server.xml`. Let us discuss these methods and their configurations.

Shared thread pool (shared executor)

It can be defined as a thread pool which is shared among many connectors. For example, if you have a four connector configuration, then you can share the same thread pool for all connectors. Following is the process to configure a shared thread pool:

1. Edit the server.xml and add the definition of the shared thread pool inside the services section. The following highlighted code shows the thread pool:

```
<Executor name="tomcatThreadPool"
namePrefix="catalina-exec-"
maxThreads="150"
minSpareThreads="4"/>
```

2. Once you have defined the shared thread pool, call the reference of the thread setting in the Connector definition for the services section of server.xml, as shown in the following code:

```
<Connector executor="tomcatThreadPool"
port="8080"
protocol="HTTP/1.1"
connectionTimeout="20000"
redirectPort="8443" />
```

We have taken the default example given in `server.xml` to explain the shared pool. In practice, you can define the shared pool based on the business requirement.

Dedicated thread pool

It can be defined as a thread pool which is dedicated to only one connector definition. For example, if the application is expecting a high load, then using a dedicated thread pool is better for the connector, to manage the Tomcat instance smoothly. Following is the process to configure a dedicated thread pool. The highlighted code defines the values of a dedicated connection pool:

1. Edit the server.xml and define the configuration of the dedicated thread pool inside the Connector section. The following highlighted code shows the thread pool:

```
<Connector port="8443" protocol="HTTP/1.1"
SSLEnabled="true"
maxThreads="150"
scheme="https"
secure="true"
clientAuth="false" sslProtocol="TLS" />
```

Shared thread pool versus dedicated thread pool

Let us do a quick comparison between shared thread pool and dedicated thread pool to find out where they can be used. The following table shows the comparison between both the thread pooling methods for Tomcat 7:

Features	Shared thread pool	Dedicated thread pool
Number of users	Less	High
Environment	Development	Production
Performance	Low	Good

maxThreads

maxThreads can be defined as the highest number of requests a server can accept. By default, Tomcat 8 comes with `maxThreads=150`. In a production environment, we have to tune the `maxThreads` based on the server performance.

Let's perform a real time `maxThreads` tuning. Let us assume we have `maxThreads=300` for an application. Now, it's time to determine the impact of the thread tuning on the server. If the `maxThreads` is not configured properly, it may degrade the server performance. How to find the impact on the server?

The solution is to check the CPU utilization for the server. If the CPU utilization is high, then decrease the value of the threads. This means the thread had degraded the server performance, but if the CPU utilization is normal, then increase the value of the threads to accommodate more concurrent users.

Note

While setting the `maxThreads`, you have to consider other resources as well, such as the database connection, network bandwidth, and so on.

maxKeepAlive

It is defined as the number of concurrent TCP connections waiting in Tomcat 8. By default, `maxKeepAlive` is set to `1`, which also means it is disabled.

If `maxKeepAlive = 1` then,

- SSL is not terminated in Tomcat
- Load balancer technique is used
- Concurrent users are more

If `maxKeepAlive > 1` then,

- SSL is terminated in Tomcat
- Concurrent users are less

JVM tuning

Before we start with JVM tuning, we should note that there are various vendors available in the market for JVM. Based on the application requirement, we should select the JDK from the vendor.

Note

Sun JDK is widely used in the IT industries.

Why do we need to tune the JDK for Tomcat?

Tomcat 8 comes with a heap size of 256 MB. Applications today need a large memory to run. In order to run the application, we have to tune the JVM parameter for Tomcat 8. Let's quickly discuss the default JVM configurations for Tomcat. The following steps describe the method to find out the Tomcat **Process ID** (**PID**) and memory values as shown in the next screenshot:

Run the following command on the terminal in Linux:

```
ps -ef |grep java
```

This will return all the Java processes running in the system, with all information such as the PID, where Tomcat is running, and so on:

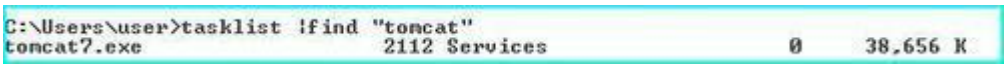

```
root 4306 1 0 14:09 pts/1 00:00:04 /usr/lib/jvm/java-8-openjdk-amd64/bin/java -
Djava.util.logging.config.file=/opt/apache-tomcat-8.5.61/conf/logging.properties -
Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager -
Djava.endorsed.dirs=/opt/apache-tomcat-8.5.61/endorsed -classpath /opt/apache-tomcat-
8.5.61/bin/bootstrap.jar:/opt/apache-tomcat-8.5.61/bin/tomcat-juli.jar -
Dcatalina.base=/opt/apache-tomcat-8.5.61 - Dcatalina.home=/opt/apache-tomcat-8.5.61 -
Djava.io.tmpdir=/opt/apache-tomcat-8.5.61/temp org.apache.catalina.startup.Bootstrap
start
```

In the previous output, `4306` is the PID for the Tomcat process. Now, we know the process ID for Tomcat, let's find out the memory allocated to the Tomcat instance using the command `jmap`.

For checking the process ID for Tomcat on Windows, you have to run the following command:

```
tasklist |find "tomcat"
```

The following screenshot shows the output of the previous command, where `2112` is the process ID for the Tomcat process:



JMAP (Memory Map)

JMAP prints the complete picture for the shared Java Virtual Memory. It is a very good utility for the administrators to check the status of the shared memory. This command provides different options for executing the command. The following table describes the useful options of JMAP:

Options	Description
`-dump`	Dumps the Java heap in `hprof` binary format
`-finalizer info`	Prints information on objects awaiting finalization
`-heap`	Prints a heap summary
`-histo`	Prints a histogram of the heap
`-permstat`	Prints class loader-wise statistics of permanent generation of the Java heap

Note

For more information on the JMAP command, please visit http://docs.oracle.com/javase/6/docs/technotes/tools/share/jmap.html.

Syntax for jmap

The syntax for using the jmap command is `./jmap -heap <process id>` where `<process id>` is the Java process for which we want to check the memory.

In the previous syntax, the `-heap` option prints a heap summary, the GC algorithm used, the heap configuration, and the generation-wise heap usage.

Note

If you want to run the `jmap` command for a 64 bit VM, then use the command `jmap -J-d64 -heap pid`.

In our current scenario, the PID is 4306:

```
[root@localhost bin]# ./jmap -heap 4306
```

The output of the previous command is as follows:

```
Attaching to process ID 4306, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 19.1-b02
using thread-local object allocation.
Mark Sweep Compact GC
```

```
Heap Configuration:
MinHeapFreeRatio = 40
MaxHeapFreeRatio = 70
MaxHeapSize = 268435456 (256.0MB)
NewSize = 1048576 (1.0MB)
MaxNewSize = 4294901760 (4095.9375MB)
OldSize = 4194304 (4.0MB)
NewRatio = 2
SurvivorRatio = 8
PermSize = 12582912 (12.0MB)
MaxPermSize = 67108864 (64.0MB)
Heap Usage:
New Generation (Eden + 1 Survivor Space):
capacity = 5111808 (4.875MB)
used = 3883008 (3.703125MB)
free = 1228800 (1.171875MB)
75.96153846153847% used
Eden Space:
capacity = 4587520 (4.375MB)
used = 3708360 (3.5365676879882812MB)
free = 879160 (0.8384323120117188MB)
80.83583286830357% used
From Space:
capacity = 524288 (0.5MB)
used = 174648 (0.16655731201171875MB)
free = 349640 (0.33344268798828125MB)
33.31146240234375% used
To Space:
capacity = 524288 (0.5MB)
used = 0 (0.0MB)
free = 524288 (0.5MB)
0.0% used
tenured generation:
capacity = 11206656 (10.6875MB)
used = 3280712 (3.1287307739257812MB)
free = 7925944 (7.558769226074219MB)
29.274673908077485% used
Perm Generation:
capacity = 12582912 (12.0MB)
used = 6639016 (6.331459045410156MB)
free = 5943896 (5.668540954589844MB)
52.762158711751304% used
```

The previous command returns the following details:

- The heap configuration for the application (the highlighted code describes the heap configuration)
- The heap utilization for each JVM component
- The algorithm used for the garbage collection

If you see the previous heap allocation, then you can clearly differentiate the memory allocated to Tomcat 8. It is 256 MB, with 12 MB of Perm Generation.

How to increase the heap size in Tomcat 8

To increase the heap size for Tomcat 8, we need to add the `JAVA_OPTS` parameter in `catalina.sh`, which can be found in `TOMCAT_HOME/bin`.

Let us take an example of increasing the max heap size to 512 MB instead 256 MB. Also setting the `Perm Gen = 256 MB`.

```
JAVA_OPTS="-Xms128m -Xmx512m -XX:MaxPermSize=256m"
```

Note

Every change in configuration for the JVM parameter will be in effect after restarting the Tomcat server. You can verify the change done in the JVM parameter by running the `jmap` command.

```
[root@localhost bin]# jmap -heap 21091
```

The output of the previous command is as follows:

```
Attaching to process ID 21091, please wait...
Debugger attached successfully.
Client compiler detected.
JVM version is 19.1-b02
using thread-local object allocation.
Mark Sweep Compact GC
Heap Configuration:
MinHeapFreeRatio = 40
MaxHeapFreeRatio = 70
MaxHeapSize = 536870912 (512.0MB)
NewSize = 1048576 (1.0MB)
MaxNewSize = 4294901760 (4095.9375MB)
OldSize = 4194304 (4.0MB)
NewRatio = 2
SurvivorRatio = 8
PermSize = 12582912 (12.0MB)
MaxPermSize = 268435456 (256.0MB)
```

The highlighted line of code in the previous code snippet is reflecting the value changed after the recycle in the JVM parameter.

We can also define `JRE_HOME, JAVA_OPTS, JAVA_ENDORSED_DIRS, JPDA_TRANSPORT, JPDA_ADDRESS, JPDA_SUSPEND, JPDA_OPTS, LOGGING_CONFIG, LOGGING_MANAGER` parameters in `catalina.sh`.

After doing the real-time implementation of increasing the memory, we will understand the need to tune JVM and how the heap value increases and decreases in Tomcat 8. Another term which people often talk about is **Garbage Collection** (**GC**). JVM tuning is not complete without garbage collection.

Garbage collection

Garbage means waste, but let's find out how this[*waste*] term fits in JVM.

What is garbage in JVM?

Garbage is nothing but an object that resides in the JVM memory and is not currently used by any program.

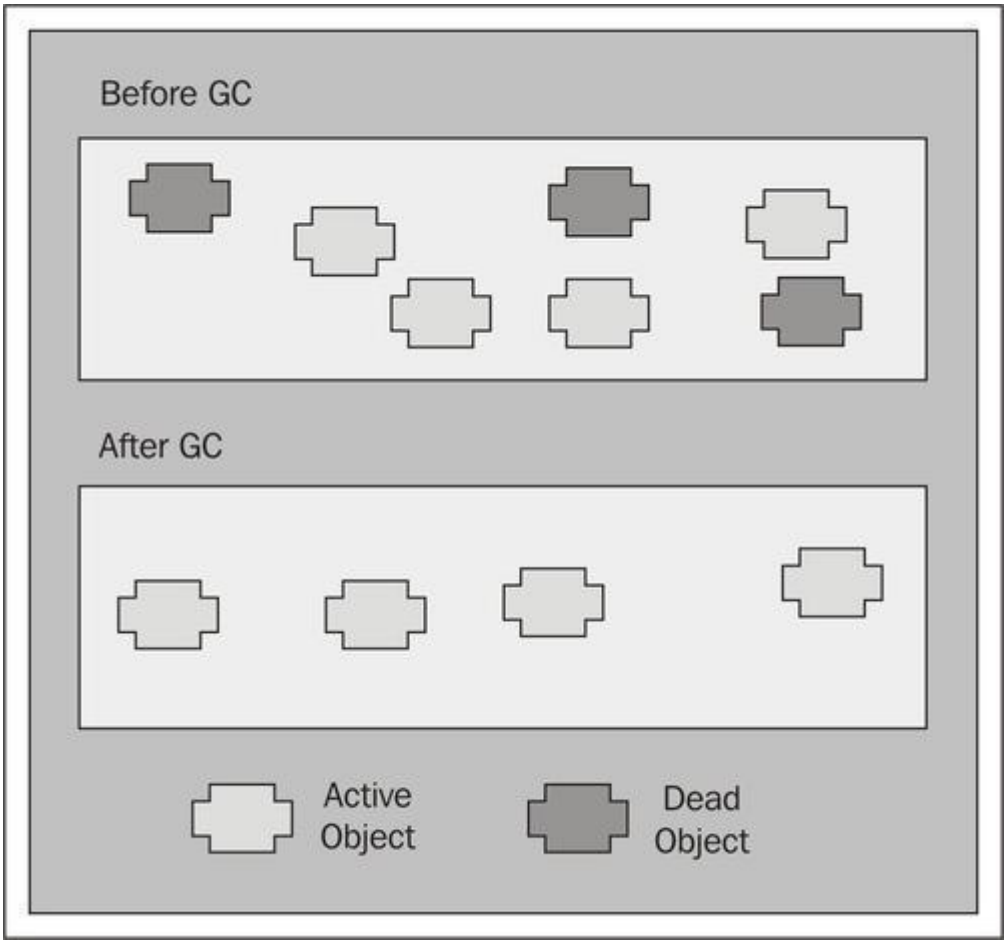
Garbage collector is an algorithm which runs periodically, collects the status of active and inactive objects in the memory, and deletes the inactive objects to release memory.

Following are the facts for garbage collection:

- Garbage collection doesn't work properly with large memory applications
- Garbage collection doesn't consider the fact that some of the objects have a very short life, but some of them remain active for years until the recycle of Tomcat 8

How garbage collection works

When the GC algorithm is called, it collects all the inactive objects present in the memory and, hence, cleans the memory. It can be explained as the opposite of manual memory management:



It removes all inactive objects from the memory and all active threads will remain in the memory. The previous figure shows the state of the memory before and after the GC. There are mainly three types of GC collectors used in the real-time environment:

- Serial collector
- Parallel collector
- Concurrent low pause collector

The following table describes the features of the serial collector:

Features	Serial collector
Process	Single thread is used for GC
GC pause	High
Threading	Single threaded
Application	Small application (data less than 100 MB)
Advantage	There is single thread communication

The following table describes the features of the parallel collector:

Features	Parallel collector
Process	Parallel thread does minor GC
GC pause	Less than Serial
Threading	Multithreaded
Application	Mid-large
Advantage	Used in applications when peak performance is needed

The following table describes the features of the concurrent collector:

Features	Concurrent collector
Process	GC is done concurrently
GC pause	Short pause
Threading	Multithreaded
Application	Mid-large
Advantage	Used in applications when a response is needed

Note

The three algorithms will work in JDK 1.5 or later.

Note

Parallel and concurrent collector algorithms should not be used together.

JVM options

The Java HotSpot VM options are classified in two categories; standard and non-standard options.

Standard options

Standard options are acknowledged by the Java HotSpot VM mentioned in the Java application launcher page for each OS. The following screenshot shows the standard option for the Java application launcher reference page:

```
[root@localhost ~]# java -showversion
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07)
Java HotSpot(TM) Client VM (build 19.1-b02, mixed mode, sharing)

Usage: java [-options] class [args...]
           (to execute a class)
   or  java [-options] -jar jarfile [args...]
           (to execute a jar file)

where options include:
    -d32          use a 32-bit data model if available

    -d64          use a 64-bit data model if available
    -client       to select the "client" VM
    -server       to select the "server" VM
    -hotspot      is a synonym for the "client" VM  [deprecated]
                  The default VM is client.

    -cp <class search path of directories and zip/jar files>
    -classpath <class search path of directories and zip/jar files>
                  A : separated list of directories, JAR archives,
                  and ZIP archives to search for class files.
    -D<name>=<value>
                  set a system property
    -verbose[:class|gc|jni]
                  enable verbose output
    -version      print product version and exit
    -version:<value>
                  require the specified version to run
    -showversion  print product version and continue
    -jre-restrict-search | -jre-no-restrict-search
                  include/exclude user private JREs in the version search
    -? -help      print this help message
    -X            print help on non-standard options
    -ea[:<packagename>...|:<classname>]
    -enableassertions[:<packagename>...|:<classname>]
                  enable assertions
    -da[:<packagename>...|:<classname>]
    -disableassertions[:<packagename>...|:<classname>]
                  disable assertions
    -esa | -enablesystemassertions
                  enable system assertions
    -dsa | -disablesystemassertions
                  disable system assertions
```

Non-standard options

Non-standard options are defined with `-X` or `-XX` options in the JVM. These options are grouped into three categories:

- **Behavioral options:** It changes the basic behavior of the VM.
- **Performance tuning options:** It triggers the performance optimization for the VM. also, these options are very useful for server tuning.
- **Debugging options:** It displays the output and printing information of the VM. In addition to that, it enables tracing in logs (these options are very useful in troubleshooting critical issues).

Note

Options that begin with `-X` are non-standard (not guaranteed to be supported on all VM implementations). Options that are specified with `-XX` are not stable and are not recommended for casual use.

The following table describes very commonly used options for JVM:

Options	Parameter	Description
Behavioral Options	`-XX:+ScavengeBeforeFullGC`	Do young generation GC prior to a full GC
Behavioral Options	`--XX:-UseParallelGC`	Use parallel garbage collection for scavenges
Performance Options	`-XX:MaxNewSize=size`	Maximum size of new generation (in bytes)
Performance Options	`-XX:MaxPermSize=64m`	Size of the Permanent Generation (after exceeding `Xmx` value)
Performance Options	`-Xms`	Minimum heap memory for the startup of Tomcat
Performance Options	`Xmx`	Maximum memory allocated to the instance
Performance Options	`-Xss`	Stack size for the heap
Debugging Options	`-XX:-CITime`	Prints time spent in the JIT Compiler
Debugging Options	`-XX:ErrorFile=./hs_err_pid<pid>.log`	If an error occurs, save the error data to this file
Debugging Options	`-XX:HeapDumpPath=./java_pid<pid>.hprof`	Path to the directory or filename for the heap dump
Debugging Options	`-XX:-HeapDumpOnOutOfMemoryError`	Dump the heap to the file when `java.lang.OutOfMemoryError` is thrown
Options	Parameter	Description

Debugging Options	`-XX:OnError="<cmd args>;<cmd args>"`	Run user-defined commands on fatal error
Debugging Options	`-XX:OnOutOfMemoryError="<cmd args>;`	Run user-defined commands when an OutOfMemoryError is first thrown
Debugging Options	`-XX:-PrintClassHistogram`	Print a histogram of class instances on **Ctrl-Break**

Note

For more information on the JVM non-standard options, please visit the link http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html.

Parameters displayed in the logs for GC

GC prints the output of the garbage collection to the `stdout` stream. At every garbage collection, the following five fields are printed:

```
[%T %B->%A(%C), %D]
```

- `%T:` This is "GC" when the garbage collection is a scavenge, and "Full GC:" is performed, then scavenge collects live objects from the new generation only, whereas a full garbage collection collects objects from all spaces in the Java heap.
- `%B:` It is the size of the Java heap used before the garbage collection, in KB.
- `%A:` It is the size of the Java heap after the garbage collection, in KB.
- `%C:` It is the current capacity of the entire Java heap, in KB.
- `%D:` It is the duration of the collection in seconds.

SurvivorRatio

It is defined as a ratio of eden to the survivor space size. The default value is 8, meaning that eden is 8 times bigger than from and to, each. The syntax for the SurvivorRatio is `-XX:SurvivorRatio=<size>`.

The following are some examples:

```
Xmn / (SurvivorRatio + 2) = size of from and to, each
( Xmn / (SurvivorRatio + 2) ) * SurvivorRatio = eden size
```

OS tuning

Every OS has its own prerequisites to run Tomcat 8 and the system has to be tuned based on the application's requirement, but there are some similarities between each OS. Let's discuss the common module used for optimization of Tomcat 8 for every OS. The OS plays a vital role for increasing the performance. Depending on the hardware, the application's performance will increase or decrease. Some of the points which are very much useful for the application are:

- **Performance characteristics of the 64 bit versus 32 bit VM:** The benefits of using 64 bit VMs are being able to address larger amounts of memory, which comes with a small performance loss in 64 bit VMs versus running the same application on a 32 bit VM. You can allocate more than 4 GB JVM for a memory-intensive application.

Note

In case you are using a 64 bit JVM edition, then you have to add 30 percent more memory as compared to a 32 bit JVM edition.

- **Files size:** Based on the application requirement, the file size is set in the OS. If the application is using a very high number of transactions, then the file limit needs to be increased.
- **Ulimits:** Based on the sessions, a user can increase the limits.

Note

These values are defined in `/etc/sysctl.conf`. If you need to update any of the above parameters then update it in `etc/sysctl.conf`, otherwise all the details will be flushed after the reboot of the OS.

- **Huge page size:** Many applications send a huge page size causing the application to run slow. In this case, you can increase the page size based on the application needs. You can check the page size by running the following command:

```
[root@localhost bin]# cat /proc/meminfo
MemTotal: 1571836 kB
MemFree: 886116 kB
Buffers: 74712 kB
Cached: 430088 kB
SwapCached: 0 kB
Active: 308608 kB
Inactive: 331944 kB
HighTotal: 671680 kB
HighFree: 97708 kB
LowTotal: 900156 kB
LowFree: 788408 kB
SwapTotal: 2040212 kB
SwapFree: 2040212 kB
Dirty: 36 kB
Writeback: 0 kB
AnonPages: 135764 kB
Mapped: 54828 kB
Slab: 33840 kB
PageTables: 3228 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
CommitLimit: 2826128 kB
Committed_AS: 496456 kB
VmallocTotal: 114680 kB
VmallocUsed: 4928 kB
VmallocChunk: 109668 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
Hugepagesize: 4096 kB
```

- It will show you the complete details of the memory. Based on the current utilization, you can increase the value.

Summary

In this lab, we have covered the different ways of performance improvement and techniques in Apache Tomcat 8. We went through step-by-step configuration for connectors, JVM performance tuning, and OS parameter optimization.

In the next lab, we will discuss the various ways of integrating different web servers with Tomcat 8 and solutions for common problems faced in a real-time environment.