**Tomcat Clustering - A Step By Step Guide**

Apache Tomcat is a great performer on its own, but if you're expecting more traffic as your site expands, or are thinking about the best way to provide high availability, you'll be happy to know that Tomcat also shines in a clustered environment. With built-in support for both synchronous and asynchronous in-memory and external session replication, cluster segmentation, and compatibility with all common load balancing solutions, your Tomcat servers are ready for the cluster right out of the box.

In this article, we'll show you how easy it is to set up a simple Tomcat cluster with load balancing and session replication.

This simple step-by-step guide will walk you through every step of the process in plain English, from [installing the load balancer](http://www.mulesoft.com/tomcat-clustering#install-modjk), to [configuring mod\_jk](http://www.mulesoft.com/tomcat-clustering#configure), to [enabling Tomcat's built-in session replication capabilities](http://www.mulesoft.com/tomcat-clustering#session-replication). Along the way, we'll point out common problem areas, to help you avoid configuration mistakes before they happen.

**A Simple Explanation of Clustering**

Although clustering can seem like a complicated topic, the premise is quite simple. A clustered architecture is used to solve one or more of the following problems:

* A single server cannot handle the high number of incoming requests efficiently
* A stateful application needs a way of preserving session data if its server fails
* A developer requires the capability to make configuration changes or [deploy](http://www.mulesoft.com/tomcat-deploy) updates to their applications without discontinuing service.

A clustered architecture solves these problems using a combination of load balancing, multiple server "workers" to process the balanced load, and some kind of session replication. Depending on the needs of the application, only some of these components may be used, or additional components such as caching and compression engines.

Since this is a how-to guide, we'll stop the general information session here, and move on to setting up a working Tomcat cluster. However, if you're new to clustering, it's probably a good idea to do a little more reading up on the subject.

For more information about the how and why of clustered architecture, check out our [Tomcat Clustering](http://www.mulesoft.com/tomcat-clustering) Basics article for an in-depth look at all the components of a cluster, comparisons of different approaches, and more.

**About Our Example Set-Up**

For the purposes of this tutorial, we'll use a simple clustered configuration:

* Apache HTTPD with mod\_jk (for load balancing)
* 2 Tomcat 6.x instances
* in-memory session replication (via Tomcat's built in functionality)

This configuration was chosen because it is a simple example of a very common clustering configuration used by many Tomcat users. Additionally, Apache HTTPD with mod\_jk or mod\_proxy is currently the default clustering solution recommended in the official Tomcat documentation provided by Apache.

We demonstrate configuration with mod\_jk rather than mod\_proxy for two reasons: it's a little more complex, and requires some additional steps, and it's currently the more mature load balancing [connector](http://www.mulesoft.com/tomcat-connector), with a wider user base in the Tomcat community. Additionally, while new releases of mod\_proxy are tied to Apache HTTPD releases, mod\_jk is developed and released separately from Tomcat, so its features tend to be more current.

Using this tutorial as a basic reference, you should be able to find enough information elsewhere to create more customized or complex configurations. If you are curious about these kinds of situations, or want to know more about how other common approaches to Tomcat clustering stack up against one another, check out our [Tomcat Cluster](http://www.mulesoft.com/tomcat-cluster) basics article for an in-depth guide.

**Setting Up Your Tomcat Cluster**

Let's get started!

**Step 1 - Install Tomcat instances and Apache HTTPD**

If you haven't already installed them, the first thing to do is to download and install the latest stable versions of Apache Tomcat 6.x and Apache HTTPD. You can find the latest stable versions on the [Apache HTTPD](http://httpd.apache.org/) and [Tomcat](http://tomcat.apache.org) project sites.

Depending on how you want to set up your servers for the purposes of this tutorial, you can either install these elements on multiple servers or a single server; the only differences will be in the port and address configuration steps. Simply follow standard conventions for sharing a single machine with multiple network services (don't overlap ports, don't use conflicting names, and so on). If you only want to use this tutorial to test different clustering configurations, the multiple Tomcat instances can live on the same machine, and even share the same base directory, using the CATALINA\_BASE variable, as long as you remember that you should move to a set-up that reflects your actual production environment before doing any benchmark testing.

For the purposes of this tutorial, we'll assume that you understand how to install these components. If you need additional help, don't worry - the Apache HTTPD [installation documentation](http://httpd.apache.org/docs/2.1/install.html) is quite good, and we've created a simple guide to installing Tomcat 6 on multiple platforms, which you can find [here](http://www.mulesoft.com/tomcat-60).

**Step 2 - Download and install mod\_jk**

mod\_jk is the Apache HTTPD module that will be used to provide our cluster with its load balancing and proxy capabilities. It uses the AJP protocol to facilitate fast communication between Tomcat servers and the Apache Web Server that will receive the client requests.

The mod\_jk module is distributed separately from Apache HTTPD as part of the Tomcat project. Binary distributions of the module are available for most major platforms, and can be downloaded [here](http://www.apache.org/dist/tomcat/tomcat-connectors/jk/binaries/). If the version and platform you are looking for is not yet available as a binary distribution, you can build the appropriate version from the source.

Once you have either downloaded and unzipped or built the module, place it in the 'modules' directory of your Apache HTTPD server.

**Step 3 - Configure mod\_jk**

Next, we'll have to set up the mod\_jk module in Apache HTTPD's configuration files. This configuration is a two step process, and can be a little confusing, as mod\_jk does not separate its proxy capabilities from its load balancing capabilities.

First, let's configure the module itself. This is done by adding a few lines to the main Apache HTTPD configuration file, httpd.conf. Take a look at this example configuration (we'll explain what each attribute does in a second):

# Load module

LoadModule jk\_module path/to/apache2/mod\_jk.so

# Specify path to worker configuration file

JkWorkersFile /path/to/apache2/conf/workers.properties

# Configure logging and memory

JkShmFile /path/to/desired/log/location/mod\_jk.shm

JkLogFile /path/to/desired/log/location/mod\_jk.log

JkLogLevel info

# Configure monitoring

JkMount /jkmanager/\* jkstatus

<Location /jkmanager>

Order deny, allow

Deny from all

Allow from localhost

</Location>

# Configure applications

JkMount /webapp-directory/\* LoadBalancer

Here's a quick explanation of the parameters we just configured.

* **LoadModule** - this command makes the mod\_jk module available for use. The extension of the module itself will vary by operating system.
* **JkWorkersFile** - sets the path to the worker configuration file, which we will create in the next step.
* **JkShmFile** - sets the path to the shared memory files for the module. Generally, you'll want to keep this with the logs.
* **JkLogFile** - sets the path to the module log file.
* **JkLogLevel** - sets the level of [logging](http://www.mulesoft.com/tomcat-logging) for the module. The valid values for this attribute, in descending order by verbosity, are "debug", "error" or "info".
* **JkMount** - this is used to map a certain URL pattern to a specific worker configured in the worker configuration file. Here, we use it twice - once to enable /jkmanager as the access URL for jkstatus, a virtual monitoring worker, and once to map all requests we want to be handled by the [cluster](http://www.mulesoft.com/tomcat-cluster) to the "lb" worker, a virtual worker that contains the load balancing capability
* **Location** - this is a [security](http://www.mulesoft.com/tomcat-security) constraint. The settings we have included allow access to the jkmanager only from the localhost (this is a Good Idea).

**Step 4 - Configure the cluster workers**

Now that we've configured the main settings, we will configure the workers. "Workers" is a blanket term used within mod\_jk to refer to both real Tomcat servers that will process requests, and virtual servers included in the module to handle load balancing and monitoring. In other words, rather than creating a separate apparatus to manage load balancing, mod\_jk simply loads an additional virtual worker with load balancing functionality. If this seems confusing to you, you're not alone - this is one area where mod\_jk shows its age compared to mod\_proxy, which keeps all of its configuration in the main httpd.conf file, and doesn't use the concept of virtual workers.

Here's a (very) basic workers.properties configuration example (see below for an explanation of the configuration directives):

# Define worker names

worker.list=jkstatus, LoadBalancer

# Create virtual workers

worker.jkstatus.type=status

worker.loadbalancer.type=lb

# Declare Tomcat server workers 1 through n

worker.worker1.type=ajp13

worker.worker1.host=hostname

worker.worker1.port=8009

# ...

worker.worker[n].type=ajp13

worker.worker[n].port=8010

worker.worker[n].host=hostname

# Associate real workers with virtual LoadBalancer worker

worker.LoadBalancer.balance\_workers=worker1,worker2,…worker[n]

Here's a quick explanation of the directives we just configured:

**Global Directives**

These are directives that apply to the entire configuration. There are only two of these directives, and here, we only use one.

**worker.list** - Allows you to specifically name any workers that should be loaded when the server starts up. These are the only workers to which you can map requests in httpd.conf. This has more uses when using mod\_jk as a proxy server. For our purposes, the two workers we've defined are enough.

**General Worker Directives**

These are directives that pertain specifically to workers, but not to virtual workers. They always take the following form:

worker.[name].[directive]=[value]

Worker names are defined as part of a directive (unless set in worker.list). Subsequent directives using the same name value will apply to the same worker. Names may only contain underscores, dashes, and alphanumeric characters, and are case sensitive.

There is a very long list of worker directives, allowing configuration of everything from session replication partner nodes, to connection timeout values, to weights for use with load balancing algorithms. It's even possible to include workers within multiple nodes, allowing you to do things such as using a very fast server as a pinch hitter to handle spikes in multiple clusters. The extensive control this provides over load balancing scenarios is the reason why using mod\_jk over mod\_proxy is currently worth the extra configuration trouble. As this is a simple tutorial, we won't go into the list here, but you can find the whole thing on the Apache [project site](http://tomcat.apache.org/connectors-doc/reference/workers.html), and it is highly recommended reading.

**worker.[name].type** - This allows you to declare a "type" for a given worker. This type can either refer to a virtual worker type (i.e. "lb" for load balancer worker, "status" for the status worker), or to the protocol that the server should use to communicate with a real worker.

Here, we use the type ajp13, which refers to the latest version of the Apache Jserv Protocol, as well as the "lb" and "status" types, which define the virtual load balancing and status manager workers.

worker.[name].host - this allows you to define the appropriate host for a worker. You can also include port in this entry by separating the host name from the port value with a ":".

**worker.[name].port** - This allows you to set a port number to access the relevant server. This is especially useful if you want to cluster multiple Tomcat instances running on a single server.

**Load Balancer Directives**

The mod\_jk virtual workers each have their own specialized subsets of directives, which provide extra levels of control over their functions. For example, although the "lb" worker uses a load balancing algorithm based on requests and each server's lbfactor to distribute the load by default, mod\_jk actually includes three additional load balancing algorithms, some of which are more appropriate for certain situations, and can be configured with the "method" directive.

As this is a bare-bones example configuration, we haven't configured any non-required directives, but as with the worker directives, the full list is available on the Tomcat Connectors [project site](http://tomcat.apache.org/download-connectors.cgi), and is recommended reading.

**worker.[name].balance\_workers=[name1],[name2],…[name[n]]** - this is the only required load balancer directive, and is used to associate a group of workers with a given load balancer. You can define multiple load balancer names in the global worker list if you will be balancing multiple clusters with a single Apache instance.

If you'd like to learn more about load balancing with mod\_jk, visit the [load balancing how-to article](http://tomcat.apache.org/connectors-doc/generic_howto/loadbalancers.html) on the Apache site.

**Tips and Tricks**

In the interest of simplicity, we'll leave it up to you to explore the other configuration options on your own. However, before we move on, here's two quick tricks that can be real time-savers when you start configuring your real-world clusters.

**Variables**

The mod\_jk worker configuration file supports the use of variables to make adding new clusters or migrating configurations to a new server an easier process. Variables can be used in place of any directive-defined value. It's very simple to define a variable. Simply use the format below, making sure that you do not use a word already associated with a specific function:

[variable\_name]=[value]

Variable names can contain any alphanumeric character, as well as dashes, underscores, and periods, and are not case sensitive. To call a variable, use the following syntax:

worker.[name].directive=$(variable\_name)

For example, you could define a base network address:

mynetwork=193.228.43

...And then use it to configure multiple workers:

worker.worker1.host=$(mynetwork).12

worker.worker2.host=$(mynetwork).13

worker.worker3.host=$(mynetwork).14

**Property Inheritance**

If you are configuring multiple similar workers or clusters, you can use the "reference" directive to cause a worker to inherit any properties of another existing worker for which you have not provided an explicit value for the new worker. References are inherited by hierarchy, so you can even create multiple subclasses of reference worker. Use the following syntax to create a reference:

worker.worker1.reference=worker.WorkerTemplate

This will cause "worker1" to inherit all the properties of the "WorkerTemplate" worker. You can create a template worker simply by defining it in the usual way and excluding it from workers.list and any balanced\_worker lists. A common use for the reference directive is to define a single load balancer, and use inherited values to split its workers across two domains.

**Putting It All Together**

A good combined use of property inheritance and variables would be to comment a section at the top of the mod\_jk configuration section of your httpd.conf file as "Global Settings and Templates", and then use something like this:

# Global Settings

myHost=my/host/name.domain

myOtherHost=my/other/hostname.domain

worker.default.connection\_timeout=1

worker.default.host=$(myHost)

worker.fastserver.lbfactor=4

worker.fastserver.host=$(myOtherHost)

Using this technique, you can create whole cluster profiles simply by referencing these archetypes, and migrate entire configurations to new servers by changing just a few variables.

For a more in-depth look at defining workers, and some more inspiration for tricky configurations, visit the Workers HowTo [[http://tomcat.apache.org/connectors-doc/generic\_howto/workers.html]](http://tomcat.apache.org/connectors-doc/generic_howto/workers.html%5d) article on Apache's website.

**Step 5 - Configure your Tomcat workers**

Now that we've finally gotten the mod\_jk configuration out of the way, it's time to configure our Tomcat instances to support clustering. To make this task a little easier to swallow, we've divided the process into two steps - enabling session replication, and configuring the actual cluster.

**Step 5a - Configure your Tomcat workers - Enabling session replication**

In this example, we'll use simple all-to-all in-[memory](http://www.mulesoft.com/tomcat-memory) session replication. This means every worker in our example cluster will replicate their sessions across every other worker. This is not always the most efficient method of session replication in higher load environments, but you can easily build on the concepts we'll introduce in this section to create a more specific solution when you design clusters for your production environment.

Tomcat provides in-memory session replication through a combination of serializable session attributes, "sticky sessions", which are provided by the load balancer, and specialized components configured in Tomcat's XML configuration files. We'll tackle each of these components one by one.

**Serializable Session Attributes**

In order to use Tomcat's built-in session replication, any session attribute or class that will need to be available in the event of failover must implement java.io.Serializable. This interface allows the [JVM](http://www.mulesoft.com/tomcat-jvm) to convert session objects into bytecode that can be stored in memory and copied to other instances. All JavaBeans are technically required to be serializeable by default, but you should make sure that all your session attributes properly implement the interface.

**Sticky Sessions**

Load balancers use a variety of methods to make sure that requests are sent to the machine that has the most current session data. The easiest of these, and the one we will use for this example, is called "sticky sessions".

A load balancer with sticky sessions enabled, after routing a request to a given worker, will pass all subsequent requests with matching sessionID values to the same worker. In the event that this worker fails, the load balancer will begin routing this request to the next most available server that has access to the failed server's session information. Tomcat's method of in-memory session replication relies on sticky sessions for both normal and failover load balancing situations.

The latest version of mod\_jk enables sticky sessions by default. If you want to be absolutely sure that sticky sessions is enabled for your configuration, you can add the worker.lb.stickysessions=true attribute to your workers.properties file.

**Make Your Applications Distributable**

In the context of clustered Java application servers, the term "distributable" is used to denote an application that allows its information to be distributed to more than one JVM. This is essential for session replication. You can mark your applications as "distributable" in one of two places. First, you can include the <distributable> element in your application's deployment descriptor (WEB-INF/web.xml), like this:

<distributable />

Note that the distributable element is one of the rare "self-closing" XML tags in Tomcat; nest it anywhere inside the enclosing <web-app> elements.

Your other option is to simply add the "distributable" attribute to the relevant application's [Context](http://www.mulesoft.com/tomcat-context) element, as follows:

<Context distributable="true">

Even if you have marked your application as distributable, you may run into problems if you haven't created your web application with clustering in mind. In addition to being declared as such, distributable applications must satisfy the following requirements:

Session attributes must implement java.io.Serializable.

HttpSession.setAttribute() must be called any time changes are made to an object that belongs to a session, so that the session replicator can distribute the changes across the cluster

The sessions must not be so big that they overload the server with traffic when copied.

**Setting jvmRoute**

The jvmRoute attribute of the Engine element allows the load balancer to match requests to the JVM currently responsible for updating the relevant session. It does this by appending the name of the JVM to the JSESSIONID of the request, and matching this against the worker name provided in workers.properites.

In order to configure jvmRoute, make sure that the value of "jvmRoute" for all your Engines is paired with an identically named Worker name entry in mod\_jk's worker.properties configuration file.

**Keeping your Workers in Sync**

When clustering multiple servers, it is important that each worker in the cluster have the same understanding of real world time, as some of Tomcat's clustering features are time-dependent.

In order to eliminate this concern, make sure that all of your servers update their time settings automatically by connecting to the same Network Time Protocol (NTP) service. This is a function of the server OS itself, not of the Tomcat instance. If you are unsure of how to set up this connection, consult the documentation provided by your OS vendor. You can find a list of NTP servers and more information about the service at the [NTP Project Wiki](http://support.ntp.org/bin/view/Servers/WebHome).

**Step 5b - Configure your Clusters**

Finally, you can now configure your cluster to work with Tomcat. This is done in Tomcat's main configuration file, server.xml, which can be found in the $CATALINA\_HOME/conf directory, within a special Cluster element. The cluster element is nested

As not every server [configuration](http://www.mulesoft.com/tomcat-configuration) requires clustering, the Cluster element is commented out by default. Open server.xml and uncomment the following entry:

<!--

<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"/>

-->

The className is the Java class responsible for providing Tomcat's clustering capabilities, which is included with all versions of Tomcat 5.x and later.

In order to configure clustering, Tomcat uses a mixture of cluster-specific elements and standard Tomcat elements nested within a Cluster element. Let's take a look at an example cluster configuration, and go through it element by element to learn how it works.

Note that as the goal of this article is simply to demonstrate an easy cluster configuration, we will not get into much depth as to way in which the various Java components that make up Tomcat's clustering functionality work. However, as you become more familiar with clustering, this is recommended reading, and you can get started on the Apache project site.

**An Example Clustering Configuration**

<Engine name="Catalina" defaultHost="www.mysite.com" jvmRoute="[worker name]">

<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster" channelSendOptions="8">

<Manager className="org.apache.catalina.ha.session.DeltaManager"

expireSessionsOnShutdown="false"

notifyListenersOnReplication="true"/>

<Channel className="org.apache.catalina.tribes.group.GroupChannel">

<Membership className="org.apache.catalina.tribes.membership.McastService"

address="228.0.0.4"

port="45564" frequency="500"

dropTime="3000"/>

<Sender className="org.apache.catalina.tribes.transport.ReplicationTransmitter">

<Transport className="org.apache.catalina.tribes.transport.nio.PooledParallelSender"/>

</Sender>

<Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"

address="auto" port="4000" autoBind="100"

selectorTimeout="5000" maxThreads="6"/>

<Interceptor className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>

<Interceptor className="org.apache.catalina.tribes.group.interceptors.MessageDispatch15Interceptor"/>

</Channel>

<Valve className="org.apache.catalina.ha.tcp.ReplicationValve" filter=""/>

<Valve className="org.apache.catalina.ha.session.JvmRouteBinderValve"/>

<ClusterListener className="org.apache.catalina.ha.session.JvmRouteSessionIDBinderListener"/>

<ClusterListener className="org.apache.catalina.ha.session.ClusterSessionListener"/>

</Cluster>

Let's go through this configuration one Element at a time.

**Engine**

This is the standard Engine element that defines Catalina as the component responsible for processing requests. As we mentioned in Step 5a, to enable session replication, you must set the "jvmRoute" attribute to match the corresponding worker you have configured in mod\_jk's workers.properties file. This value must be unique for every node included in the cluster.

**Cluster**

This is the main Cluster element, within which all other clustering elements are nested. It supports a variety of attributes, but in this simple example, we have only configured one, "channelSendOptions". This attribute sets a flag within Tomcat's clustering class that chooses between different methods of cluster communication. These options are outside the scope of this article, but a safe default setting is "8", which enables asynchronous communication.

**Manager**

This is the standard element that Tomcat uses for session management. When nested inside the Cluster element, it is used to tell Tomcat which cluster-aware session manager should be used for session replication. In this example, we have used the DeltaManager, which provides basic cluster-aware session management, as well as additional capabilities you can use to divide your cluster into multiple groups in the future. The attributes we have configured, "expireSessionsOnShutdown" and "notifyListenersOnReplication", have been configured to prevent a failing node from destroying sessions on other clustered nodes and explicitly notify the ClusterListeners when a session has been updated.

**Channel**

This element communicates with a component of Tomcat's clustering solution called Tribes. This component handles all communication between the clustered nodes. In this example, we have configured Tribes to use multicast communication, although more complicated situations can be configured using single point broadcasting. The Channel element is used to contain a series of other elements that divide cluster communication into simple blocks.

**Membership**

This Tribes-related element defines the address all nodes will use to keep track of one another. The settings we have used here are the Tribes defaults.

**Sender**

This Tribes-related element, in conduction with the Transport element nested inside of it, is used to choose from and configure a number of different implementations of cluster communication. Here, we have used the NIO transport, which generally provides the best [performance](http://www.mulesoft.com/tomcat-performance).

**Receiver**

This Tribes-related element configures a single Receiver component, which receives messages from other nodes' Sender components. The attributes of the element allow you to specify addresses, buffer sizes, thread limits, and more. The settings we have used here allow the nodes to automatically discover one another via an address that Tribes will generate automatically.

**Interceptor**

Interceptor elements are used to make modifications to messages sent between nodes. For example, one of the Interceptor elements we have configured here detects delays that may be preventing a member from updating its table due to timeout, and provides an alternative TCP connection. Tribes includes a number of standard interceptors; to enable any of them, simply add an addition Interceptor element with the appropriate className. Here, we have included only interceptors useful in almost all clustering situations.

**Valve**

Tomcat's standard Valve element can be nested within Cluster elements to provide filtering. The element includes a number of cluster-specific implementations. For example, one of the Valves we have included here can be used to restrict the kinds of files replicated across the cluster. For this example configuration, we have included the most commonly used Valves, with blank attribute values that you can configure as required.

**ClusterListener**

This element listens to all messages sent through by cluster workers, and intercepts those that match their respective implementation's specifications. These elements operate in a very similar manner to Inteceptor elements, except that rather than modifying messages and passing them on to a Receiver, they are the intended recipient of the messages for which they are listening.

Once you have edited your server.xml file, simply restart the server, and you will have a cluster-enabled Tomcat instance up and running! Note that you will need to add this configuration to each Tomcat instance you wish to add to the cluster as a worker, and that each Engine element must have its own unique jvmRoute.