Apache Tomcat 8

Version 8.5.100, Mar 19 2024

Clustering/Session Replication How-To

Important Note

You can also check the configuration reference documentation.

Table of Contents

For the impatient

Simply add

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

to your `<Engine>` or your `<Host>` element to enable clustering.

Using the above configuration will enable all-to-all session replication using the `DeltaManager` to replicate session deltas. By all-to-all, we mean that *every* session gets replicated to *all the other nodes* in the cluster. This works great for smaller clusters, but we don't recommend it for larger clusters — more than 4 nodes or so. Also, when using the DeltaManager, Tomcat will replicate sessions to *all* nodes, *even nodes that don't have the application deployed*.
To get around these problem, you'll want to use the `BackupManager`. The `BackupManager` only replicates the session data to *one* backup node, and only to nodes that have the application deployed. Once you have a simple cluster running with the `DeltaManager`, you will probably want to migrate to the `BackupManager` as you increase the number of nodes in your cluster.

Here are some of the important default values:

1. Multicast address is 228.0.0.4
2. Multicast port is 45564 (the port and the address together determine cluster membership.
3. The IP broadcasted is `java.net.InetAddress.getLocalHost().getHostAddress()` (make sure you don't broadcast 127.0.0.1, this is a common error)
4. The TCP port listening for replication messages is the first available server socket in range `4000-4100`
5. Listener is configured `ClusterSessionListener`
6. Two interceptors are configured `TcpFailureDetector` and `MessageDispatchInterceptor`

The following is the default cluster configuration:

```
        <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"/>
            <Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"
                      address="auto"
                      port="4000"
                      autoBind="100"
                      selectorTimeout="5000"
                      maxThreads="6"/>

            <Sender className="org.apache.catalina.tribes.transport.ReplicationTransmitter">
              <Transport className="org.apache.catalina.tribes.transport.nio.PooledParallelSender"/>
            </Sender>
            <Interceptor className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>
            <Interceptor className="org.apache.catalina.tribes.group.interceptors.MessageDispatchInterceptor"/>
          </Channel>
```

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

        <Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"
                  tempDir="/tmp/war-temp/"
                  deployDir="/tmp/war-deploy/"
                  watchDir="/tmp/war-listen/"
                  watchEnabled="false"/>

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

We will cover this section in more detail later in this document.

Security

The cluster implementation is written on the basis that a secure, trusted network is used for all of the cluster related network traffic. It is not safe to run a cluster on a insecure, untrusted network.

There are many options for providing a secure, trusted network for use by a Tomcat cluster. These include:

- private LAN
- a Virtual Private Network (VPN)
- IPSEC

The EncryptInterceptor provides confidentiality and integrity protection but it does not protect against all risks associated with running a Tomcat cluster on an untrusted network, particularly DoS attacks.

Cluster Basics

To run session replication in your Tomcat 8 container, the following steps should be completed:

- All your session attributes must implement `java.io.Serializable`
- Uncomment the `Cluster` element in server.xml
- If you have defined custom cluster valves, make sure you have the `ReplicationValve` defined as well under the Cluster element in server.xml
- If your Tomcat instances are running on the same machine, make sure the `Receiver.port` attribute is unique for each instance, in most cases Tomcat is smart enough to resolve this on it's own by autodetecting available ports in the range 4000-4100
- Make sure your `web.xml` has the `<distributable/>` element
- If you are using mod_jk, make sure that jvmRoute attribute is set at your Engine `<Engine name="Catalina" jvmRoute="node01" >` and that the jvmRoute attribute value matches your worker name in workers.properties
- Make sure that all nodes have the same time and sync with NTP service!
- Make sure that your loadbalancer is configured for sticky session mode.

Load balancing can be achieved through many techniques, as seen in the Load Balancing chapter.

Note: Remember that your session state is tracked by a cookie, so your URL must look the same from the out side otherwise, a new session will be created.

The Cluster module uses the Tomcat JULI logging framework, so you can configure logging through the regular logging.properties file. To track messages, you can enable logging on the key: `org.apache.catalina.tribes.MESSAGES`

Overview

To enable session replication in Tomcat, three different paths can be followed to achieve the exact same thing:

1. Using session persistence, and saving the session to a shared file system (PersistenceManager + FileStore)
2. Using session persistence, and saving the session to a shared database (PersistenceManager + JDBCStore)
3. Using in-memory-replication, using the SimpleTcpCluster that ships with Tomcat (lib/catalina-tribes.jar + lib/catalina-ha.jar)

Tomcat can perform an all-to-all replication of session state using the `DeltaManager` or perform backup replication to only one node using the `BackupManager`. The all-to-all replication is an algorithm that is only efficient when the clusters are small. For larger clusters, you should use the BackupManager to use a primary-secondary session replication strategy where the session will only be stored at one backup node.
Currently you can use the domain worker attribute (mod_jk > 1.2.8) to build cluster partitions with the potential of having a more scalable cluster solution with the DeltaManager (you'll need to configure the domain interceptor for this). In order to keep the network traffic down in an all-to-all environment, you can split your cluster into smaller groups. This can be easily achieved by using different multicast addresses for the different groups. A very simple setup would look like this:

```
        DNS Round Robin
               |
         Load Balancer
          /          \
      Cluster1      Cluster2
      /     \        /     \
 Tomcat1 Tomcat2  Tomcat3 Tomcat4
```

What is important to mention here, is that session replication is only the beginning of clustering. Another popular concept used to implement clusters is farming, i.e., you deploy your apps only to one server, and the cluster will distribute the deployments across the entire cluster. This is all capabilities that can go into with the FarmWarDeployer (s. cluster example at `server.xml`)

In the next section will go deeper into how session replication works and how to configure it.

Cluster Information

Membership is established using multicast heartbeats. Hence, if you wish to subdivide your clusters, you can do this by changing the multicast IP address or port in the `<Membership>` element.

The heartbeat contains the IP address of the Tomcat node and the TCP port that Tomcat listens to for replication traffic. All data communication happens over TCP.

The `ReplicationValve` is used to find out when the request has been completed and initiate the replication, if any. Data is only replicated if the session has changed (by calling setAttribute or removeAttribute on the session).

One of the most important performance considerations is the synchronous versus asynchronous replication. In a synchronous replication mode the request doesn't return until the replicated session has been sent over the wire and reinstantiated on all the other cluster nodes. Synchronous vs. asynchronous is configured using the `channelSendOptions` flag and is an integer value. The default value for the `SimpleTcpCluster/DeltaManager` combo is 8, which is asynchronous. See the configuration reference for more discussion on the various `channelSendOptions` values.

For convenience, `channelSendOptions` can be set by name(s) rather than integer, which are then translated to their integer value upon startup. The valid option names are: "asynchronous" (alias "async"), "byte_message" (alias "byte"), "multicast", "secure", "synchronized_ack" (alias "sync"), "udp", "use_ack". Use comma to separate multiple names, e.g. pass "async, multicast" for the options `SEND_OPTIONS_ASYNCHRONOUS | SEND_OPTIONS_MULTICAST`.

You can read more on the send flag(overview) or the send flag(javadoc). During async replication, the request is returned before the data has been replicated. async replication yields shorter request times, and synchronous replication guarantees the session to be replicated before the request returns.

Bind session after crash to failover node

If you are using mod_jk and not using sticky sessions or for some reasons sticky session don't work, or you are simply failing over, the session id will need to be modified as it previously contained the worker id of the previous tomcat (as defined by jvmRoute in the Engine element). To solve this, we will use the JvmRouteBinderValve.

The JvmRouteBinderValve rewrites the session id to ensure that the next request will remain sticky (and not fall back to go to random nodes since the worker is no longer available) after a fail over. The valve rewrites the JSESSIONID value in the cookie with the same name. Not having this valve in place, will make it harder to ensure stickiness in case of a failure for the mod_jk module.

Remember, if you are adding your own valves in server.xml then the defaults are no longer valid, make sure that you add in all the appropriate valves as defined by the default.

Hint:
With attribute *sessionIdAttribute* you can change the request attribute name that included the old session id. Default attribute name is *org.apache.catalina.ha.session.JvmRouteOriginalSessionID*.

Trick:
You can enable this mod_jk turnover mode via JMX before you drop a node to all backup nodes! Set enable true on all JvmRouteBinderValve backups, disable worker at mod_jk and then drop node and restart it! Then enable mod_jk Worker and disable JvmRouteBinderValves again. This use case means that only requested session are migrated.

Configuration Example

```
        <Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"
                 channelSendOptions="6">

          <Manager className="org.apache.catalina.ha.session.BackupManager"
                   expireSessionsOnShutdown="false"
                   notifyListenersOnReplication="true"
                   mapSendOptions="6"/>
          <!--
          <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"/>
            <Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"
                      address="auto"
                      port="5000"
                      selectorTimeout="100"
                      maxThreads="6"/>

            <Sender className="org.apache.catalina.tribes.transport.ReplicationTransmitter">
              <Transport className="org.apache.catalina.tribes.transport.nio.PooledParallelSender"/>
            </Sender>
            <Interceptor className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>
            <Interceptor className="org.apache.catalina.tribes.group.interceptors.MessageDispatchInterceptor"/>
            <Interceptor className="org.apache.catalina.tribes.group.interceptors.ThroughputInterceptor"/>
```

```
        </Channel>

        <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"
               filter=".*\.gif|.*\.js|.*\.jpeg|.*\.jpg|.*\.png|.*\.htm|.*\.html|.*\.css|.*\.txt"/>

        <Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"
                  tempDir="/tmp/war-temp/"
                  deployDir="/tmp/war-deploy/"
                  watchDir="/tmp/war-listen/"
                  watchEnabled="false"/>

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

Break it down!!

```
      <Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster"
               channelSendOptions="6">
```

The main element, inside this element all cluster details can be configured. The `channelSendOptions` is the flag that is attached to each message sent by the SimpleTcpCluster class or any objects that are invoking the SimpleTcpCluster.send method. The description of the send flags is available at our javadoc site The `DeltaManager` sends information using the SimpleTcpCluster.send method, while the backup manager sends it itself directly through the channel.
For more info, Please visit the reference documentation

```
        <Manager className="org.apache.catalina.ha.session.BackupManager"
                 expireSessionsOnShutdown="false"
                 notifyListenersOnReplication="true"
                 mapSendOptions="6"/>
        <!--
        <Manager className="org.apache.catalina.ha.session.DeltaManager"
                 expireSessionsOnShutdown="false"
                 notifyListenersOnReplication="true"/>
        -->
```

This is a template for the manager configuration that will be used if no manager is defined in the <Context> element. In Tomcat 5.x each webapp marked distributable had to use the same manager, this is no longer the case since Tomcat you can define a manager class for each webapp, so that you can mix managers in your cluster. Obviously the managers on one node's application has to correspond with the same manager on the same application on the other node. If no manager has been specified for the webapp, and the webapp is marked <distributable/> Tomcat will take this manager configuration and create a manager instance cloning this configuration.
For more info, Please visit the reference documentation

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

The channel element is Tribes, the group communication framework used inside Tomcat. This element encapsulates everything that has to do with communication and membership logic.
For more info, Please visit the reference documentation

```
          <Membership className="org.apache.catalina.tribes.membership.McastService"
                      address="228.0.0.4"
                      port="45564"
                      frequency="500"
                      dropTime="3000"/>
```

Membership is done using multicasting. Please note that Tribes also supports static memberships using the `StaticMembershipInterceptor` if you want to extend your membership to points beyond multicasting. The address attribute is the multicast address used and the port is the multicast port. These two together create the cluster separation. If you want a QA cluster and a production cluster, the easiest config is to have the QA cluster be on a separate multicast address/port combination than the production cluster.
The membership component broadcasts TCP address/port of itself to the other nodes so that communication between nodes can be done over TCP. Please note that the address being broadcasted is the one of the `Receiver.address` attribute.
For more info, Please visit the reference documentation

```
          <Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver"
                    address="auto"
                    port="5000"
                    selectorTimeout="100"
                    maxThreads="6"/>
```

In tribes the logic of sending and receiving data has been broken into two functional components. The Receiver, as the name suggests is responsible for receiving messages. Since the Tribes stack is thread less, (a popular improvement now adopted by other frameworks as well), there is a thread pool in this component that has a maxThreads and minThreads setting.
The address attribute is the host address that will be broadcasted by the membership component to the other nodes.
For more info, Please visit the reference documentation

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

The sender component, as the name indicates is responsible for sending messages to other nodes. The sender has a shell component, the `ReplicationTransmitter` but the real stuff done is done in the sub component, `Transport`. Tribes support having a pool of senders, so that messages can be sent in parallel and if using the NIO sender, you can send messages concurrently as well.

Concurrently means one message to multiple senders at the same time and Parallel means multiple messages to multiple senders at the same time.
For more info, Please visit the reference documentation

```
          <Interceptor className="org.apache.catalina.tribes.group.interceptors.TcpFailureDetector"/>
          <Interceptor className="org.apache.catalina.tribes.group.interceptors.MessageDispatchInterceptor"/>
          <Interceptor className="org.apache.catalina.tribes.group.interceptors.ThroughputInterceptor"/>
        </Channel>
```

Tribes uses a stack to send messages through. Each element in the stack is called an interceptor, and works much like the valves do in the Tomcat servlet container. Using interceptors, logic can be broken into more manageable pieces of code. The interceptors configured above are:
TcpFailureDetector - verifies crashed members through TCP, if multicast packets get dropped, this interceptor protects against false positives, ie the node marked as crashed even though it still is alive and running.
MessageDispatchInterceptor - dispatches messages to a thread (thread pool) to send message asynchronously.
ThroughputInterceptor - prints out simple stats on message traffic.
Please note that the order of interceptors is important. The way they are defined in server.xml is the way they are represented in the channel stack. Think of it as a linked list, with the head being the first most interceptor and the tail the last.
For more info, Please visit the reference documentation

```
        <Valve className="org.apache.catalina.ha.tcp.ReplicationValve"
               filter=".*\.gif|.*\.js|.*\.jpeg|.*\.jpg|.*\.png|.*\.htm|.*\.html|.*\.css|.*\.txt"/>
```

The cluster uses valves to track requests to web applications, we've mentioned the ReplicationValve and the JvmRouteBinderValve above. The <Cluster> element itself is not part of the pipeline in Tomcat, instead the cluster adds the valve to its parent container. If the <Cluster> elements is configured in the <Engine> element, the valves get added to the engine and so on.
For more info, Please visit the reference documentation

```
        <Deployer className="org.apache.catalina.ha.deploy.FarmWarDeployer"
                  tempDir="/tmp/war-temp/"
                  deployDir="/tmp/war-deploy/"
                  watchDir="/tmp/war-listen/"
                  watchEnabled="false"/>
```

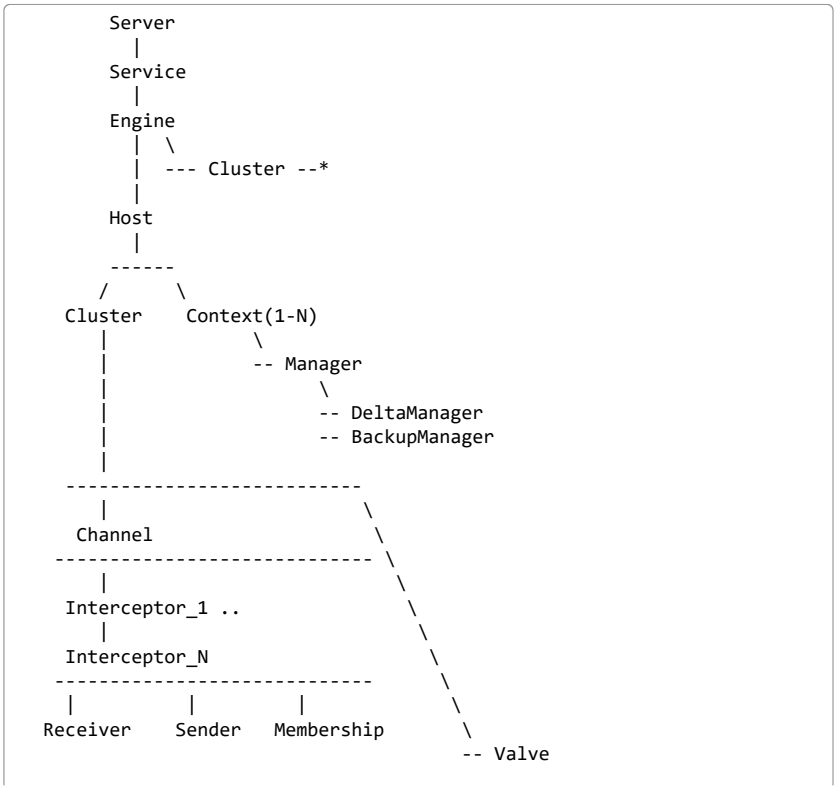
The default tomcat cluster supports farmed deployment, ie, the cluster can deploy and undeploy applications on the other nodes. The state of this component is currently in flux but will be addressed soon. There was a change in the deployment algorithm between Tomcat 5.0 and 5.5 and at that point, the logic of this component changed to where the deploy dir has to match the webapps directory.
For more info, Please visit the reference documentation

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

Since the SimpleTcpCluster itself is a sender and receiver of the Channel object, components can register themselves as listeners to the SimpleTcpCluster. The listener above `ClusterSessionListener` listens for DeltaManager replication messages and applies the deltas to the manager that in turn applies it to the session.
For more info, Please visit the reference documentation

Cluster Architecture

Component Levels:



```
|      \
|       -- ReplicationValve
|       -- JvmRouteBinderValve
|
-- LifecycleListener
|
-- ClusterListener
|      \
|       -- ClusterSessionListener
|
-- Deployer
       \
        -- FarmWarDeployer
```

How it Works

To make it easy to understand how clustering works, we are gonna to take you through a series of scenarios. In this scenario we only plan to use two tomcat instances `TomcatA` and `TomcatB`. We will cover the following sequence of events:

1. `TomcatA` starts up
2. `TomcatB` starts up (Wait the TomcatA start is complete)
3. `TomcatA` receives a request, a session `S1` is created.
4. `TomcatA` crashes
5. `TomcatB` receives a request for session `S1`
6. `TomcatA` starts up
7. `TomcatA` receives a request, invalidate is called on the session (`S1`)
8. `TomcatB` receives a request, for a new session (`S2`)
9. `TomcatA` The session `S2` expires due to inactivity.

Ok, now that we have a good sequence, we will take you through exactly what happens in the session replication code

1. **`TomcatA` starts up**

   Tomcat starts up using the standard start up sequence. When the Host object is created, a cluster object is associated with it. When the contexts are parsed, if the distributable element is in place in the web.xml file, Tomcat asks the Cluster class (in this case `SimpleTcpCluster`) to create a manager for the replicated context. So with clustering enabled, distributable set in web.xml Tomcat will create a `DeltaManager` for that context instead of a `StandardManager`. The cluster class will start up a membership service (multicast) and a replication service (tcp unicast). More on the architecture further down in this document.

2. **`TomcatB` starts up**

   When TomcatB starts up, it follows the same sequence as TomcatA did with one exception. The cluster is started and will establish a membership (TomcatA, TomcatB). TomcatB will now request the session state from a server that already exists in the cluster, in this case TomcatA. TomcatA responds to the request, and before TomcatB starts listening for HTTP requests, the state has been transferred from TomcatA to TomcatB. In case TomcatA doesn't respond, TomcatB will time out after 60 seconds, issue a log entry, and continue starting. The session state gets transferred for each web application that has distributable in its web.xml. (Note: To use session replication efficiently, all your tomcat instances should be configured the same.)

3. **`TomcatA` receives a request, a session `S1` is created.**

   The request coming in to TomcatA is handled exactly the same way as without session replication, until the request is completed, at which time the `ReplicationValve` will intercept the request before the response is returned to the user. At this point it finds that the session has been modified, and it uses TCP to replicate the session to TomcatB. Once the serialized data has been handed off to the operating system's TCP logic, the request returns to the user, back through the valve pipeline. For each request the entire session is replicated, this allows code that modifies attributes in the session without calling setAttribute or removeAttribute to be replicated. A useDirtyFlag configuration parameter can be used to optimize the number of times a session is replicated.

4. **`TomcatA` crashes**

   When TomcatA crashes, TomcatB receives a notification that TomcatA has dropped out of the cluster. TomcatB removes TomcatA from its membership list, and TomcatA will no longer be notified of any changes that occurs in TomcatB. The load balancer will redirect the requests from TomcatA to TomcatB and all the sessions are current.

5. **`TomcatB` receives a request for session `S1`**

   Nothing exciting, TomcatB will process the request as any other request.

6. **`TomcatA` starts up**

   Upon start up, before TomcatA starts taking new request and making itself available to it will follow the start up sequence described above 1) 2). It will join the cluster, contact TomcatB for the current state of all the sessions. And once it receives the session state, it finishes loading and opens its HTTP/mod_jk ports. So no requests will make it to TomcatA until it has received the session state from TomcatB.

7. **`TomcatA` receives a request, invalidate is called on the session (`S1`)**

   The invalidate call is intercepted, and the session is queued with invalidated sessions. When the request is complete, instead of sending out the session that has changed, it sends out an "expire" message to TomcatB and TomcatB will invalidate the session as well.

8. **`TomcatB` receives a request, for a new session (`S2`)**

   Same scenario as in step 3)

9. `TomcatA` The session `S2` expires due to inactivity.

   The invalidate call is intercepted the same way as when a session is invalidated by the user, and the session is queued with invalidated sessions. At this point, the invalidated session will not be replicated across until another request comes through the system and checks the invalid queue.

Phuuuhh! :)

Membership Clustering membership is established using very simple multicast pings. Each Tomcat instance will periodically send out a multicast ping, in the ping message the instance will broadcast its IP and TCP listen port for replication. If an instance has not received such a ping within a given timeframe, the member is considered dead. Very simple, and very effective! Of course, you need to enable multicasting on your system.

TCP Replication Once a multicast ping has been received, the member is added to the cluster Upon the next replication request, the sending instance will use the host and port info and establish a TCP socket. Using this socket it sends over the serialized data. The reason I chose TCP sockets is because it has built in flow control and guaranteed delivery. So I know, when I send some data, it will make it there :)

Distributed locking and pages using frames Tomcat does not keep session instances in sync across the cluster. The implementation of such logic would be to much overhead and cause all kinds of problems. If your client accesses the same session simultaneously using multiple requests, then the last request will override the other sessions in the cluster.

Monitoring your Cluster with JMX

Monitoring is a very important question when you use a cluster. Some of the cluster objects are JMX MBeans

Add the following parameter to your startup script:

```
set CATALINA_OPTS=\
-Dcom.sun.management.jmxremote \
-Dcom.sun.management.jmxremote.port=%my.jmx.port% \
-Dcom.sun.management.jmxremote.ssl=false \
-Dcom.sun.management.jmxremote.authenticate=false
```

List of Cluster Mbeans

Name	Description	MBean ObjectName - Engine	MBean ObjectName - Host
Cluster	The complete cluster element	`type=Cluster`	`type=Cluster,host=${HOST}`
DeltaManager	This manager control the sessions and handle session replication	`type=Manager,context=${APP.CONTEXT.PATH}, host=${HOST}`	`type=Manager,context=${APP.CONTEXT.PATH}, host=${HOST}`
FarmWarDeployer	Manages the process of deploying an application to all nodes in the cluster	Not supported	`type=Cluster, host=${HOST}, component=deployer`
Member	Represents a node in the cluster	type=Cluster, component=member, name=${NODE_NAME}	`type=Cluster, host=${HOST}, component=member, name=${NODE_NAME}`
ReplicationValve	This valve control the replication to the backup nodes	`type=Valve,name=ReplicationValve`	`type=Valve,name=ReplicationValve,host=${HOST}`
JvmRouteBinderValve	This is a cluster fallback valve to change the Session ID to the current tomcat jvmroute.	`type=Valve,name=JvmRouteBinderValve, context=${APP.CONTEXT.PATH}`	`type=Valve,name=JvmRouteBinderValve,host=${HOST}, context=${APP.CONTEXT.PATH}`

FAQ

Please see the clustering section of the FAQ.
